

Fast Auxiliary Space Preconditioning

Generated by Doxygen 1.9.2

1 Introduction	1
2 How to obtain FASP	3
3 Building and Installation	5
4 Developers	7
5 Doxygen	9
6 Data Structure Index	11
6.1 Data Structures	11
7 File Index	13
7.1 File List	13
8 Data Structure Documentation	17
8.1 AMG_data Struct Reference	17
8.1.1 Detailed Description	18
8.1.2 Field Documentation	18
8.1.2.1 A	18
8.1.2.2 b	19
8.1.2.3 cfmark	19
8.1.2.4 colors	19
8.1.2.5 cycle_type	19
8.1.2.6 ic	19
8.1.2.7 icmap	20
8.1.2.8 ILU_levels	20
8.1.2.9 LU	20
8.1.2.10 max_levels	20
8.1.2.11 mumps	20
8.1.2.12 near_kernel_basis	21
8.1.2.13 near_kernel_dim	21
8.1.2.14 num_levels	21
8.1.2.15 Numeric	21
8.1.2.16 P	21
8.1.2.17 pdata	22
8.1.2.18 R	22
8.1.2.19 Schwarz	22
8.1.2.20 SWZ_levels	22
8.1.2.21 w	22
8.1.2.22 weight	23

8.1.2.23 x	23
8.2 AMG_data_bsr Struct Reference	23
8.2.1 Detailed Description	24
8.2.2 Field Documentation	25
8.2.2.1 A	25
8.2.2.2 A_nk	25
8.2.2.3 Ac	25
8.2.2.4 b	25
8.2.2.5 cfmark	26
8.2.2.6 diaginv	26
8.2.2.7 diaginv_SS	26
8.2.2.8 ILU_levels	26
8.2.2.9 LU	26
8.2.2.10 max_levels	27
8.2.2.11 mumps	27
8.2.2.12 near_kernel_basis	27
8.2.2.13 near_kernel_dim	27
8.2.2.14 num_levels	27
8.2.2.15 Numeric	28
8.2.2.16 P	28
8.2.2.17 P_nk	28
8.2.2.18 pdata	28
8.2.2.19 PP	28
8.2.2.20 PP_LU	29
8.2.2.21 pw	29
8.2.2.22 R	29
8.2.2.23 R_nk	29
8.2.2.24 SS	29
8.2.2.25 sw	30
8.2.2.26 w	30
8.2.2.27 x	30
8.3 AMG_param Struct Reference	30
8.3.1 Detailed Description	33
8.3.2 Field Documentation	33
8.3.2.1 aggregation_type	33
8.3.2.2 aggressive_level	33
8.3.2.3 aggressive_path	33
8.3.2.4 AMG_type	34
8.3.2.5 amli_coef	34

8.3.2.6 amli_degree	34
8.3.2.7 coarse_dof	34
8.3.2.8 coarse_scaling	34
8.3.2.9 coarse_solver	35
8.3.2.10 coarsening_type	35
8.3.2.11 cycle_type	35
8.3.2.12 ILU_droptol	35
8.3.2.13 ILU_levels	35
8.3.2.14 ILU_lfil	36
8.3.2.15 ILU_permtol	36
8.3.2.16 ILU_relax	36
8.3.2.17 ILU_type	36
8.3.2.18 interpolation_type	36
8.3.2.19 max_aggregation	37
8.3.2.20 max_levels	37
8.3.2.21 max_row_sum	37
8.3.2.22 maxit	37
8.3.2.23 nl_amli_krylov_type	37
8.3.2.24 pair_number	38
8.3.2.25 polynomial_degree	38
8.3.2.26 postsmooth_iter	38
8.3.2.27 presmooth_iter	38
8.3.2.28 print_level	38
8.3.2.29 quality_bound	39
8.3.2.30 relaxation	39
8.3.2.31 smooth_filter	39
8.3.2.32 smooth_order	39
8.3.2.33 smooth_restriction	39
8.3.2.34 smoother	40
8.3.2.35 strong_coupled	40
8.3.2.36 strong_threshold	40
8.3.2.37 SWZ_blksolver	40
8.3.2.38 SWZ_levels	40
8.3.2.39 SWZ_maxlvl	41
8.3.2.40 SWZ_mmsize	41
8.3.2.41 SWZ_type	41
8.3.2.42 tentative_smooth	41
8.3.2.43 tol	41
8.3.2.44 truncation_threshold	42

8.4 block_dvector Struct Reference	42
8.4.1 Detailed Description	42
8.4.2 Field Documentation	42
8.4.2.1 blocks	42
8.4.2.2 brow	43
8.5 block_ivector Struct Reference	43
8.5.1 Detailed Description	43
8.5.2 Field Documentation	43
8.5.2.1 blocks	44
8.5.2.2 brow	44
8.6 dBLCmat Struct Reference	44
8.6.1 Detailed Description	44
8.6.2 Field Documentation	45
8.6.2.1 bcol	45
8.6.2.2 blocks	45
8.6.2.3 brow	45
8.7 dBSRmat Struct Reference	45
8.7.1 Detailed Description	46
8.7.2 Field Documentation	46
8.7.2.1 COL	46
8.7.2.2 IA	47
8.7.2.3 JA	47
8.7.2.4 nb	47
8.7.2.5 NNZ	47
8.7.2.6 ROW	47
8.7.2.7 storage_manner	48
8.7.2.8 val	48
8.8 dCOOmat Struct Reference	48
8.8.1 Detailed Description	49
8.8.2 Field Documentation	49
8.8.2.1 col	49
8.8.2.2 colind	49
8.8.2.3 nnz	49
8.8.2.4 row	50
8.8.2.5 rowind	50
8.8.2.6 val	50
8.9 dCSRLmat Struct Reference	50
8.9.1 Detailed Description	51
8.9.2 Field Documentation	51

8.9.2.1 col	51
8.9.2.2 dif	52
8.9.2.3 index	52
8.9.2.4 ja	52
8.9.2.5 nnz	52
8.9.2.6 nz_diff	52
8.9.2.7 row	53
8.9.2.8 start	53
8.9.2.9 val	53
8.10 dCSRmat Struct Reference	53
8.10.1 Detailed Description	54
8.10.2 Field Documentation	54
8.10.2.1 col	54
8.10.2.2 IA	55
8.10.2.3 JA	55
8.10.2.4 nnz	55
8.10.2.5 row	55
8.10.2.6 val	55
8.11 ddenmat Struct Reference	56
8.11.1 Detailed Description	56
8.11.2 Field Documentation	56
8.11.2.1 col	56
8.11.2.2 row	56
8.11.2.3 val	57
8.12 dSTRmat Struct Reference	57
8.12.1 Detailed Description	58
8.12.2 Field Documentation	58
8.12.2.1 diag	58
8.12.2.2 nband	58
8.12.2.3 nc	58
8.12.2.4 ngrid	59
8.12.2.5 nx	59
8.12.2.6 nxy	59
8.12.2.7 ny	59
8.12.2.8 nz	59
8.12.2.9 offdiag	60
8.12.2.10 offsets	60
8.13 dvector Struct Reference	60
8.13.1 Detailed Description	60

8.13.2 Field Documentation	60
8.13.2.1 row	61
8.13.2.2 val	61
8.14 grid2d Struct Reference	61
8.14.1 Detailed Description	62
8.14.2 Field Documentation	62
8.14.2.1 e	62
8.14.2.2 edges	62
8.14.2.3 ediri	62
8.14.2.4 efather	63
8.14.2.5 p	63
8.14.2.6 pdiri	63
8.14.2.7 pfather	63
8.14.2.8 s	63
8.14.2.9 t	64
8.14.2.10 tfather	64
8.14.2.11 triangles	64
8.14.2.12 vertices	64
8.15 iBLCmat Struct Reference	64
8.15.1 Detailed Description	65
8.15.2 Field Documentation	65
8.15.2.1 bcol	65
8.15.2.2 blocks	65
8.15.2.3 brow	66
8.16 iCOOmat Struct Reference	66
8.16.1 Detailed Description	66
8.16.2 Field Documentation	67
8.16.2.1 col	67
8.16.2.2 I	67
8.16.2.3 J	67
8.16.2.4 nnz	67
8.16.2.5 row	68
8.16.2.6 val	68
8.17 iCSRmat Struct Reference	68
8.17.1 Detailed Description	69
8.17.2 Field Documentation	69
8.17.2.1 col	69
8.17.2.2 IA	69
8.17.2.3 JA	69

8.17.2.4 nnz	70
8.17.2.5 row	70
8.17.2.6 val	70
8.18 idenmat Struct Reference	70
8.18.1 Detailed Description	71
8.18.2 Field Documentation	71
8.18.2.1 col	71
8.18.2.2 row	71
8.18.2.3 val	71
8.19 ILU_data Struct Reference	72
8.19.1 Detailed Description	73
8.19.2 Field Documentation	73
8.19.2.1 A	73
8.19.2.2 col	73
8.19.2.3 ic	73
8.19.2.4 icmap	73
8.19.2.5 ijlu	74
8.19.2.6 ilevL	74
8.19.2.7 ilevU	74
8.19.2.8 iperm	74
8.19.2.9 jlevL	74
8.19.2.10 jlevU	75
8.19.2.11 luval	75
8.19.2.12 nb	75
8.19.2.13 ncolors	75
8.19.2.14 nlevL	75
8.19.2.15 nlevU	76
8.19.2.16 nwork	76
8.19.2.17 nzlu	76
8.19.2.18 row	76
8.19.2.19 type	76
8.19.2.20 uptr	77
8.19.2.21 work	77
8.20 ILU_param Struct Reference	77
8.20.1 Detailed Description	78
8.20.2 Field Documentation	78
8.20.2.1 ILU_droptol	78
8.20.2.2 ILU_lfil	78
8.20.2.3 ILU_permtol	78

8.20.2.4 ILU_relax	78
8.20.2.5 ILU_type	79
8.20.2.6 print_level	79
8.21 input_param Struct Reference	79
8.21.1 Detailed Description	80
8.21.2 Field Documentation	81
8.21.2.1 AMG_aggregation_type	81
8.21.2.2 AMG_aggressive_level	81
8.21.2.3 AMG_aggressive_path	81
8.21.2.4 AMG_amli_degree	81
8.21.2.5 AMG_coarse_dof	82
8.21.2.6 AMG_coarse_scaling	82
8.21.2.7 AMG_coarse_solver	82
8.21.2.8 AMG_coarsening_type	82
8.21.2.9 AMG_cycle_type	82
8.21.2.10 AMG_ILU_levels	83
8.21.2.11 AMG_interpolation_type	83
8.21.2.12 AMG_levels	83
8.21.2.13 AMG_max_aggregation	83
8.21.2.14 AMG_max_row_sum	83
8.21.2.15 AMG_maxit	84
8.21.2.16 AMG_nl_amli_krylov_type	84
8.21.2.17 AMG_pair_number	84
8.21.2.18 AMG_polynomial_degree	84
8.21.2.19 AMG_postsmooth_iter	84
8.21.2.20 AMG_presmooth_iter	85
8.21.2.21 AMG_quality_bound	85
8.21.2.22 AMG_relaxation	85
8.21.2.23 AMG_smooth_filter	85
8.21.2.24 AMG_smooth_order	85
8.21.2.25 AMG_smooth_restriction	86
8.21.2.26 AMG_smoother	86
8.21.2.27 AMG_strong_coupled	86
8.21.2.28 AMG_strong_threshold	86
8.21.2.29 AMG_SWZ_levels	86
8.21.2.30 AMG_tentative_smooth	87
8.21.2.31 AMG_tol	87
8.21.2.32 AMG_truncation_threshold	87
8.21.2.33 AMG_type	87

8.21.2.34 decoup_type	87
8.21.2.35 ILU_droptol	88
8.21.2.36 ILU_lfil	88
8.21.2.37 ILU_permtol	88
8.21.2.38 ILU_relax	88
8.21.2.39 ILU_type	88
8.21.2.40 inifile	89
8.21.2.41 itsolver_maxit	89
8.21.2.42 itsolver_tol	89
8.21.2.43 output_type	89
8.21.2.44 precondition_type	89
8.21.2.45 print_level	90
8.21.2.46 problem_num	90
8.21.2.47 restart	90
8.21.2.48 solver_type	90
8.21.2.49 stop_type	90
8.21.2.50 SWZ_blksolver	91
8.21.2.51 SWZ_maxlvl	91
8.21.2.52 SWZ_mmsize	91
8.21.2.53 SWZ_type	91
8.21.2.54 workdir	91
8.22 ITS_param Struct Reference	92
8.22.1 Detailed Description	92
8.22.2 Field Documentation	92
8.22.2.1 decoup_type	92
8.22.2.2 itsolver_type	92
8.22.2.3 maxit	93
8.22.2.4 precondition_type	93
8.22.2.5 print_level	93
8.22.2.6 restart	93
8.22.2.7 stop_type	93
8.22.2.8 tol	94
8.23 ivector Struct Reference	94
8.23.1 Detailed Description	94
8.23.2 Field Documentation	94
8.23.2.1 row	94
8.23.2.2 val	95
8.24 Mumps_data Struct Reference	95
8.24.1 Detailed Description	95

8.24.2 Field Documentation	95
8.24.2.1 job	95
8.25 mxv_matfree Struct Reference	96
8.25.1 Detailed Description	96
8.25.2 Field Documentation	96
8.25.2.1 data	96
8.25.2.2 fct	96
8.26 Pardiso_data Struct Reference	97
8.26.1 Detailed Description	97
8.26.2 Field Documentation	97
8.26.2.1 pt	97
8.27 precondition Struct Reference	97
8.27.1 Detailed Description	98
8.27.2 Field Documentation	98
8.27.2.1 data	98
8.27.2.2 fct	98
8.28 precondition_data Struct Reference	99
8.28.1 Detailed Description	100
8.28.2 Field Documentation	100
8.28.2.1 A	100
8.28.2.2 A_nk	100
8.28.2.3 AMG_type	101
8.28.2.4 amli_coef	101
8.28.2.5 amli_degree	101
8.28.2.6 coarse_scaling	101
8.28.2.7 coarse_solver	101
8.28.2.8 coarsening_type	102
8.28.2.9 cycle_type	102
8.28.2.10 LU	102
8.28.2.11 max_levels	102
8.28.2.12 maxit	102
8.28.2.13 mgl_data	103
8.28.2.14 nl_amli_krylov_type	103
8.28.2.15 P_nk	103
8.28.2.16 polynomial_degree	103
8.28.2.17 postsmooth_iter	103
8.28.2.18 presmooth_iter	104
8.28.2.19 print_level	104
8.28.2.20 r	104

8.28.2.21 R_nk	104
8.28.2.22 relaxation	104
8.28.2.23 smooth_order	105
8.28.2.24 smoother	105
8.28.2.25 tentative_smooth	105
8.28.2.26 tol	105
8.28.2.27 w	105
8.29 precondition_data_blc Struct Reference	106
8.29.1 Detailed Description	106
8.29.2 Field Documentation	106
8.29.2.1 A_diag	106
8.29.2.2 Ablc	106
8.29.2.3 amgparam	107
8.29.2.4 LU_diag	107
8.29.2.5 mgl	107
8.29.2.6 r	107
8.30 precondition_data_bsr Struct Reference	107
8.30.1 Detailed Description	109
8.30.2 Field Documentation	109
8.30.2.1 A	109
8.30.2.2 A_nk	109
8.30.2.3 AMG_type	110
8.30.2.4 amli_coef	110
8.30.2.5 amli_degree	110
8.30.2.6 coarse_scaling	110
8.30.2.7 coarse_solver	110
8.30.2.8 coarsening_type	111
8.30.2.9 cycle_type	111
8.30.2.10 LU	111
8.30.2.11 max_levels	111
8.30.2.12 maxit	111
8.30.2.13 mgl_data	112
8.30.2.14 nl_amli_krylov_type	112
8.30.2.15 P_nk	112
8.30.2.16 postsmooth_iter	112
8.30.2.17 pres_mgl_data	112
8.30.2.18 presmooth_iter	113
8.30.2.19 print_level	113
8.30.2.20 r	113

8.30.2.21 <code>R_nk</code>	113
8.30.2.22 <code>relaxation</code>	113
8.30.2.23 <code>smooth_order</code>	114
8.30.2.24 <code>smoother</code>	114
8.30.2.25 <code>tentative_smooth</code>	114
8.30.2.26 <code>tol</code>	114
8.30.2.27 <code>w</code>	114
8.31 <code>precond_data_str</code> Struct Reference	115
8.31.1 Detailed Description	116
8.31.2 Field Documentation	116
8.31.2.1 <code>A</code>	116
8.31.2.2 <code>A_str</code>	116
8.31.2.3 <code>AMG_type</code>	117
8.31.2.4 <code>coarse_scaling</code>	117
8.31.2.5 <code>coarsening_type</code>	117
8.31.2.6 <code>cycle_type</code>	117
8.31.2.7 <code>diaginv</code>	117
8.31.2.8 <code>diaginvS</code>	118
8.31.2.9 <code>LU</code>	118
8.31.2.10 <code>max_levels</code>	118
8.31.2.11 <code>maxit</code>	118
8.31.2.12 <code>mgl_data</code>	118
8.31.2.13 <code>neigh</code>	119
8.31.2.14 <code>order</code>	119
8.31.2.15 <code>pivot</code>	119
8.31.2.16 <code>pivotS</code>	119
8.31.2.17 <code>postsmooth_iter</code>	119
8.31.2.18 <code>presmooth_iter</code>	120
8.31.2.19 <code>print_level</code>	120
8.31.2.20 <code>r</code>	120
8.31.2.21 <code>relaxation</code>	120
8.31.2.22 <code>scaled</code>	120
8.31.2.23 <code>smoother</code>	121
8.31.2.24 <code>SS_str</code>	121
8.31.2.25 <code>tol</code>	121
8.31.2.26 <code>w</code>	121
8.32 <code>precond_data_sweeping</code> Struct Reference	121
8.32.1 Detailed Description	122
8.32.2 Field Documentation	122

8.32.2.1 A	122
8.32.2.2 Ai	123
8.32.2.3 local_A	123
8.32.2.4 local_index	123
8.32.2.5 local_LU	123
8.32.2.6 NumLayers	123
8.32.2.7 r	124
8.32.2.8 w	124
8.33 precondition_diag_bsr Struct Reference	124
8.33.1 Detailed Description	124
8.33.2 Field Documentation	125
8.33.2.1 diag	125
8.33.2.2 nb	125
8.34 precondition_diag_str Struct Reference	125
8.34.1 Detailed Description	126
8.34.2 Field Documentation	126
8.34.2.1 diag	126
8.34.2.2 nc	126
8.35 SWZ_data Struct Reference	126
8.35.1 Detailed Description	127
8.35.2 Field Documentation	128
8.35.2.1 A	128
8.35.2.2 al	128
8.35.2.3 au	128
8.35.2.4 blk_data	128
8.35.2.5 blk_solver	129
8.35.2.6 iblock	129
8.35.2.7 jblock	129
8.35.2.8 mask	129
8.35.2.9 maxa	129
8.35.2.10 maxbs	130
8.35.2.11 memt	130
8.35.2.12 mumps	130
8.35.2.13 nblk	130
8.35.2.14 rhsloc	130
8.35.2.15 rhsloc1	131
8.35.2.16 SWZ_type	131
8.35.2.17 swzparam	131
8.35.2.18 xloc1	131

8.36 SWZ_param Struct Reference	131
8.36.1 Detailed Description	132
8.36.2 Field Documentation	132
8.36.2.1 print_level	132
8.36.2.2 SWZ_blksolver	132
8.36.2.3 SWZ_maxlvl	133
8.36.2.4 SWZ_mmsize	133
8.36.2.5 SWZ_type	133
9 File Documentation	135
9.1 doxygen.h File Reference	135
9.1.1 Detailed Description	135
9.2 doxygen.h	135
9.3 XtrMumps.c File Reference	135
9.3.1 Detailed Description	136
9.3.2 Macro Definition Documentation	136
9.3.2.1 ICNTL	136
9.3.3 Function Documentation	136
9.3.3.1 fasp_solver_mumps()	136
9.3.3.2 fasp_solver_mumps_steps()	137
9.4 XtrMumps.c	137
9.5 XtrPardiso.c File Reference	143
9.5.1 Detailed Description	143
9.5.2 Function Documentation	144
9.5.2.1 fasp_solver_pardiso()	144
9.6 XtrPardiso.c	144
9.7 XtrSamg.c File Reference	147
9.7.1 Detailed Description	148
9.7.2 Function Documentation	148
9.7.2.1 dCSRmat2SAMGInput()	148
9.7.2.2 dvector2SAMGInput()	149
9.8 XtrSamg.c	149
9.9 XtrSuperlu.c File Reference	150
9.9.1 Detailed Description	151
9.9.2 Function Documentation	151
9.9.2.1 fasp_solver_superlu()	151
9.10 XtrSuperlu.c	152
9.11 XtrUmfpack.c File Reference	153
9.11.1 Detailed Description	153

9.11.2 Function Documentation	153
9.11.2.1 fasp_solver_umfpack()	153
9.12 XtrUmfpack.c	154
9.13 fasp.h File Reference	156
9.13.1 Detailed Description	158
9.13.2 Macro Definition Documentation	158
9.13.2.1 __FASP_HEADER__	159
9.13.2.2 ABS	159
9.13.2.3 DIAGONAL_PREF	159
9.13.2.4 DLMALLOC	159
9.13.2.5 FASP_VERSION	159
9.13.2.6 GE	159
9.13.2.7 GT	159
9.13.2.8 INT	160
9.13.2.9 ISNAN	160
9.13.2.10 LE	160
9.13.2.11 LONG	160
9.13.2.12 LONGLONG	160
9.13.2.13 LONGREAL	160
9.13.2.14 LS	160
9.13.2.15 MAX	161
9.13.2.16 MIN	161
9.13.2.17 MULTI_COLOR_ORDER	161
9.13.2.18 NEDMALLOC	161
9.13.2.19 PUT_INT	161
9.13.2.20 PUT_REAL	161
9.13.2.21 REAL	161
9.13.2.22 RS_C1	162
9.13.2.23 SHORT	162
9.13.2.24 STRLEN	162
9.13.3 Typedef Documentation	162
9.13.3.1 dCOOmat	162
9.13.3.2 dCSRLmat	162
9.13.3.3 dCSRmat	162
9.13.3.4 ddenmat	162
9.13.3.5 dSTRmat	162
9.13.3.6 dvector	163
9.13.3.7 iCOOmat	163
9.13.3.8 iCSRmat	163

9.13.3.9 idenmat	163
9.13.3.10 ivector	163
9.14 fasp.h	163
9.15 fasp_block.h File Reference	172
9.15.1 Detailed Description	173
9.15.2 Macro Definition Documentation	173
9.15.2.1 __FASPBLOCK_HEADER__	173
9.15.3 Typedef Documentation	174
9.15.3.1 block_dvector	174
9.15.3.2 block_ivecator	174
9.15.3.3 dBLCmat	174
9.15.3.4 dBSRmat	174
9.15.3.5 iBLCmat	174
9.16 fasp_block.h	174
9.17 fasp_const.h File Reference	177
9.17.1 Detailed Description	180
9.17.2 Macro Definition Documentation	181
9.17.2.1 AMLI_CYCLE	181
9.17.2.2 ASCEND	181
9.17.2.3 BIGREAL	181
9.17.2.4 CF_ORDER	181
9.17.2.5 CGPT	181
9.17.2.6 CLASSIC_AMG	181
9.17.2.7 COARSE_AC	182
9.17.2.8 COARSE_CR	182
9.17.2.9 COARSE_MIS	182
9.17.2.10 COARSE_RS	182
9.17.2.11 COARSE_RSP	182
9.17.2.12 CPFIRST	182
9.17.2.13 DESCEND	182
9.17.2.14 ERROR_ALLOC_MEM	182
9.17.2.15 ERROR_AMG_COARSE_TYPE	183
9.17.2.16 ERROR_AMG_COARSEING	183
9.17.2.17 ERROR_AMG_INTERP_TYPE	183
9.17.2.18 ERROR_AMG_SETUP	183
9.17.2.19 ERROR_AMG_SMOOTH_TYPE	183
9.17.2.20 ERROR_DATA_STRUCTURE	183
9.17.2.21 ERROR_DATA_ZERODIAG	183
9.17.2.22 ERROR_DUMMY_VAR	183

9.17.2.23 ERROR_INPUT_PAR	184
9.17.2.24 ERROR_LIC_TYPE	184
9.17.2.25 ERROR_MAT_SIZE	184
9.17.2.26 ERROR_MISC	184
9.17.2.27 ERROR_NUM_BLOCKS	184
9.17.2.28 ERROR_OPEN_FILE	184
9.17.2.29 ERROR_QUAD_DIM	184
9.17.2.30 ERROR_QUAD_TYPE	184
9.17.2.31 ERROR_READ_FILE	185
9.17.2.32 ERROR_REGRESS	185
9.17.2.33 ERROR_SOLVER_EXIT	185
9.17.2.34 ERROR_SOLVER_ILUSETUP	185
9.17.2.35 ERROR_SOLVER_MAXIT	185
9.17.2.36 ERROR_SOLVER_MISC	185
9.17.2.37 ERROR_SOLVER_PRECTYPE	185
9.17.2.38 ERROR_SOLVER_SOLSTAG	185
9.17.2.39 ERROR_SOLVER_STAG	186
9.17.2.40 ERROR_SOLVER_TOLSMALL	186
9.17.2.41 ERROR_SOLVER_TYPE	186
9.17.2.42 ERROR_UNKNOWN	186
9.17.2.43 ERROR_WRONG_FILE	186
9.17.2.44 FALSE	186
9.17.2.45 FASP_SUCCESS	186
9.17.2.46 FGPT	186
9.17.2.47 FPFIRST	187
9.17.2.48 FPNA_RATIO	187
9.17.2.49 G0PT	187
9.17.2.50 ILUK	187
9.17.2.51 ILUt	187
9.17.2.52 ILUtp	187
9.17.2.53 INTERP_DIR	187
9.17.2.54 INTERP_ENG	188
9.17.2.55 INTERP_EXT	188
9.17.2.56 INTERP_STD	188
9.17.2.57 ISPT	188
9.17.2.58 MAT_bBSR	188
9.17.2.59 MAT_bCSR	188
9.17.2.60 MAT_BLC	188
9.17.2.61 MAT_BSR	188

9.17.2.62 MAT_bSTR	189
9.17.2.63 MAT_CSR	189
9.17.2.64 MAT_CSRL	189
9.17.2.65 MAT_FREE	189
9.17.2.66 MAT_STR	189
9.17.2.67 MAT_SymCSR	189
9.17.2.68 MAX_AMG_LVL	189
9.17.2.69 MAX_CRATE	189
9.17.2.70 MAX_REFINE_LVL	190
9.17.2.71 MAX_RESTART	190
9.17.2.72 MAX_STAG	190
9.17.2.73 MIN_CDOF	190
9.17.2.74 MIN_CRATE	190
9.17.2.75 NL_AMLI_CYCLE	190
9.17.2.76 NO_ORDER	190
9.17.2.77 NPAIR	190
9.17.2.78 OFF	191
9.17.2.79 ON	191
9.17.2.80 OPENMP_HOLDS	191
9.17.2.81 PAIRWISE	191
9.17.2.82 PREC_AMG	191
9.17.2.83 PREC_DIAG	191
9.17.2.84 PREC_FMG	191
9.17.2.85 PREC_ILU	191
9.17.2.86 PREC_NULL	192
9.17.2.87 PREC_SCHWARZ	192
9.17.2.88 PRINT_ALL	192
9.17.2.89 PRINT_MIN	192
9.17.2.90 PRINT_MORE	192
9.17.2.91 PRINT_MOST	192
9.17.2.92 PRINT_NONE	192
9.17.2.93 PRINT_SOME	193
9.17.2.94 SA_AMG	193
9.17.2.95 SCHWARZ_BACKWARD	193
9.17.2.96 SCHWARZ_FORWARD	193
9.17.2.97 SCHWARZ_SYMMETRIC	193
9.17.2.98 SMALLREAL	193
9.17.2.99 SMALLREAL2	193
9.17.2.100 SMOOTHER_BLKOIL	193

9.17.2.101 SMOOTHER_CG	194
9.17.2.102 SMOOTHER_GS	194
9.17.2.103 SMOOTHER_GSOR	194
9.17.2.104 SMOOTHER_JACOBI	194
9.17.2.105 SMOOTHER_L1DIAG	194
9.17.2.106 SMOOTHER_POLY	194
9.17.2.107 SMOOTHER_SGS	194
9.17.2.108 SMOOTHER_SGSOR	195
9.17.2.109 SMOOTHER_SOR	195
9.17.2.110 SMOOTHER_SPETEN	195
9.17.2.111 SMOOTHER_SSOR	195
9.17.2.112 SOLVER_AMG	195
9.17.2.113 SOLVER_BiCGstab	195
9.17.2.114 SOLVER_CG	195
9.17.2.115 SOLVER_DEFAULT	195
9.17.2.116 SOLVER_FMG	196
9.17.2.117 SOLVER_GCG	196
9.17.2.118 SOLVER_GCR	196
9.17.2.119 SOLVER_GMRES	196
9.17.2.120 SOLVER_MinRes	196
9.17.2.121 SOLVER_MUMPS	196
9.17.2.122 SOLVER_PARDISO	196
9.17.2.123 SOLVER_SBiCGstab	196
9.17.2.124 SOLVER_SCG	197
9.17.2.125 SOLVER_SGCG	197
9.17.2.126 SOLVER_SGMRES	197
9.17.2.127 SOLVER_SMinRes	197
9.17.2.128 SOLVER_SUPERLU	197
9.17.2.129 SOLVER_SVFGMRES	197
9.17.2.130 SOLVER_SVGMRES	197
9.17.2.131 SOLVER_UMFPACK	197
9.17.2.132 SOLVER_VFGMRES	198
9.17.2.133 SOLVER_VGMRES	198
9.17.2.134 SPAIR	198
9.17.2.135 STAG_RATIO	198
9.17.2.136 STOP_MOD_REL_RES	198
9.17.2.137 STOP_REL_PRECRES	198
9.17.2.138 STOP_REL_RES	198
9.17.2.139 TRUE	198

9.17.2.140 UA_AMG	199
9.17.2.141 UNPT	199
9.17.2.142 USERDEFINED	199
9.17.2.143 V_CYCLE	199
9.17.2.144 VMB	199
9.17.2.145 VW_CYCLE	199
9.17.2.146 W_CYCLE	199
9.17.2.147 WV_CYCLE	200
9.18 fasp_const.h	200
9.19 fasp_grid.h File Reference	202
9.19.1 Detailed Description	202
9.19.2 Macro Definition Documentation	202
9.19.2.1 __FASPGRID_HEADER__	202
9.19.3 Typedef Documentation	203
9.19.3.1 grid2d	203
9.19.3.2 pcgrid2d	203
9.19.3.3 pgrid2d	203
9.20 fasp_grid.h	203
9.21 AuxArray.c File Reference	203
9.21.1 Detailed Description	204
9.21.2 Function Documentation	204
9.21.2.1 fasp_darray_cp()	204
9.21.2.2 fasp_darray_set()	204
9.21.2.3 fasp_iarray_cp()	205
9.21.2.4 fasp_iarray_set()	205
9.22 AuxArray.c	207
9.23 AuxConvert.c File Reference	209
9.23.1 Detailed Description	209
9.23.2 Function Documentation	209
9.23.2.1 fasp_aux_bbyteToldouble()	209
9.23.2.2 fasp_aux_change_endian4()	210
9.23.2.3 fasp_aux_change_endian8()	210
9.24 AuxConvert.c	211
9.25 AuxGivens.c File Reference	211
9.25.1 Detailed Description	212
9.25.2 Function Documentation	212
9.25.2.1 fasp_aux_givens()	212
9.26 AuxGivens.c	212
9.27 AuxGraphics.c File Reference	213

9.27.1 Detailed Description	214
9.27.2 Function Documentation	214
9.27.2.1 fasp_dbsr_plot()	214
9.27.2.2 fasp_dbsr_subplot()	214
9.27.2.3 fasp_dcsr_plot()	215
9.27.2.4 fasp_dcsr_subplot()	216
9.27.2.5 fasp_grid2d_plot()	216
9.28 AuxGraphics.c	217
9.29 AuxInput.c File Reference	223
9.29.1 Detailed Description	224
9.29.2 Function Documentation	224
9.29.2.1 fasp_param_check()	224
9.29.2.2 fasp_param_input()	224
9.30 AuxInput.c	226
9.31 AuxMemory.c File Reference	236
9.31.1 Detailed Description	236
9.31.2 Function Documentation	237
9.31.2.1 fasp_mem_calloc()	237
9.31.2.2 fasp_mem_free()	237
9.31.2.3 fasp_mem_iludata_check()	238
9.31.2.4 fasp_mem_realloc()	238
9.31.2.5 fasp_mem_usage()	238
9.31.3 Variable Documentation	239
9.31.3.1 Million	239
9.32 AuxMemory.c	239
9.33 AuxMessage.c File Reference	241
9.33.1 Detailed Description	241
9.33.2 Function Documentation	242
9.33.2.1 fasp_amgcomplexity()	242
9.33.2.2 fasp_amgcomplexity_bsr()	242
9.33.2.3 fasp_chkerr()	242
9.33.2.4 fasp_cputime()	244
9.33.2.5 fasp_itinfo()	244
9.33.2.6 fasp_message()	245
9.34 AuxMessage.c	245
9.35 AuxParam.c File Reference	248
9.35.1 Detailed Description	249
9.35.2 Function Documentation	249
9.35.2.1 fasp_param_amg_init()	249

9.35.2.2 fasp_param_amg_print()	250
9.35.2.3 fasp_param_amg_set()	250
9.35.2.4 fasp_param_amg_to_prec()	251
9.35.2.5 fasp_param_amg_to_precbsr()	251
9.35.2.6 fasp_param_ilu_init()	251
9.35.2.7 fasp_param_ilu_print()	252
9.35.2.8 fasp_param_ilu_set()	252
9.35.2.9 fasp_param_init()	253
9.35.2.10 fasp_param_input_init()	253
9.35.2.11 fasp_param_prec_to_amg()	254
9.35.2.12 fasp_param_precbsr_to_amg()	254
9.35.2.13 fasp_param_set()	254
9.35.2.14 fasp_param_solver_init()	255
9.35.2.15 fasp_param_solver_print()	255
9.35.2.16 fasp_param_solver_set()	256
9.35.2.17 fasp_param_swz_init()	256
9.35.2.18 fasp_param_swz_print()	257
9.35.2.19 fasp_param_swz_set()	257
9.36 AuxParam.c	257
9.37 AuxSort.c File Reference	267
9.37.1 Detailed Description	268
9.37.2 Function Documentation	268
9.37.2.1 fasp_aux_BiSearch()	268
9.37.2.2 fasp_aux_dQuickSort()	269
9.37.2.3 fasp_aux_dQuickSortIndex()	269
9.37.2.4 fasp_aux_iQuickSort()	270
9.37.2.5 fasp_aux_iQuickSortIndex()	270
9.37.2.6 fasp_aux_merge()	271
9.37.2.7 fasp_aux_msort()	272
9.37.2.8 fasp_aux_unique()	272
9.38 AuxSort.c	273
9.39 AuxThreads.c File Reference	276
9.39.1 Detailed Description	276
9.39.2 Function Documentation	276
9.39.2.1 fasp_get_start_end()	276
9.39.2.2 fasp_set_gs_threads()	277
9.39.3 Variable Documentation	277
9.39.3.1 THDs_AMG_GS	277
9.39.3.2 THDs_CPR_gGS	278

9.39.3.3 THDs_CPR_IGS	278
9.40 AuxThreads.c	278
9.41 AuxTiming.c File Reference	279
9.41.1 Detailed Description	280
9.41.2 Function Documentation	280
9.41.2.1 fasp_gettime()	280
9.42 AuxTiming.c	280
9.43 AuxVector.c File Reference	281
9.43.1 Detailed Description	281
9.43.2 Function Documentation	282
9.43.2.1 fasp_dvec_alloc()	282
9.43.2.2 fasp_dvec_cp()	282
9.43.2.3 fasp_dvec_create()	282
9.43.2.4 fasp_dvec_free()	283
9.43.2.5 fasp_dvec_isnan()	283
9.43.2.6 fasp_dvec_maxdiff()	284
9.43.2.7 fasp_dvec_rand()	284
9.43.2.8 fasp_dvec_set()	285
9.43.2.9 fasp_dvec_symdiagscale()	286
9.43.2.10 fasp_ivec_alloc()	286
9.43.2.11 fasp_ivec_create()	286
9.43.2.12 fasp_ivec_free()	287
9.43.2.13 fasp_ivec_set()	287
9.44 AuxVector.c	288
9.45 BlaArray.c File Reference	291
9.45.1 Detailed Description	292
9.45.2 Function Documentation	292
9.45.2.1 fasp_blas_darray_ax()	292
9.45.2.2 fasp_blas_darray_axpby()	293
9.45.2.3 fasp_blas_darray_axpy()	293
9.45.2.4 fasp_blas_darray_axpy_nc2()	294
9.45.2.5 fasp_blas_darray_axpy_nc3()	294
9.45.2.6 fasp_blas_darray_axpy_nc5()	295
9.45.2.7 fasp_blas_darray_axpy_nc7()	295
9.45.2.8 fasp_blas_darray_axpyz()	296
9.45.2.9 fasp_blas_darray_axpyz_nc2()	296
9.45.2.10 fasp_blas_darray_axpyz_nc3()	297
9.45.2.11 fasp_blas_darray_axpyz_nc5()	297
9.45.2.12 fasp_blas_darray_axpyz_nc7()	298

9.45.2.13 fasp_blas_darray_dotprod()	298
9.45.2.14 fasp_blas_darray_norm1()	299
9.45.2.15 fasp_blas_darray_norm2()	299
9.45.2.16 fasp_blas_darray_norminf()	300
9.46 BlaArray.c	300
9.47 BlaEigen.c File Reference	307
9.47.1 Detailed Description	307
9.47.2 Function Documentation	307
9.47.2.1 fasp_dcsr_maxeig()	308
9.48 BlaEigen.c	308
9.49 BlaFormat.c File Reference	309
9.49.1 Detailed Description	309
9.49.2 Function Documentation	310
9.49.2.1 fasp_format_dblc_dcsr()	310
9.49.2.2 fasp_format_dbsr_dcoo()	310
9.49.2.3 fasp_format_dbsr_dcsr()	311
9.49.2.4 fasp_format_dcoo_dcsr()	311
9.49.2.5 fasp_format_dcsr_dbsr()	312
9.49.2.6 fasp_format_dcsr_dcoo()	312
9.49.2.7 fasp_format_dcsrl_dcsr()	313
9.49.2.8 fasp_format_dstr_dbsr()	313
9.49.2.9 fasp_format_dstr_dcsr()	313
9.50 BlaFormat.c	314
9.51 BlalLU.c File Reference	325
9.51.1 Detailed Description	325
9.51.2 Function Documentation	326
9.51.2.1 fasp_iluk()	326
9.51.2.2 fasp_ilut()	327
9.51.2.3 fasp_ilutp()	328
9.51.2.4 fasp_symbfactor()	329
9.52 BlalLU.c	331
9.53 BlalLUSetupBSR.c File Reference	353
9.53.1 Detailed Description	353
9.53.2 Function Documentation	354
9.53.2.1 fasp_ilu_dbsr_setup()	354
9.53.2.2 fasp_ilu_dbsr_setup_levsch_omp()	354
9.53.2.3 fasp_ilu_dbsr_setup_levsch_step()	355
9.53.2.4 fasp_ilu_dbsr_setup_mc_omp()	356
9.53.2.5 fasp_ilu_dbsr_setup_omp()	356

9.53.2.6 fasp_ilu_dbsr_setup_step()	357
9.53.2.7 mulcol_independ_set()	358
9.53.2.8 topologic_sort_ILU()	358
9.54 BlalLUSetupBSR.c	358
9.55 BlalLUSetupCSR.c File Reference	379
9.55.1 Detailed Description	379
9.55.2 Function Documentation	380
9.55.2.1 fasp_ilu_dcsr_setup()	380
9.56 BlalLUSetupCSR.c	380
9.57 BlalLUSetupSTR.c File Reference	382
9.57.1 Detailed Description	382
9.57.2 Function Documentation	383
9.57.2.1 fasp_ilu_dstr_setup0()	383
9.57.2.2 fasp_ilu_dstr_setup1()	383
9.58 BlalLUSetupSTR.c	384
9.59 BlalO.c File Reference	399
9.59.1 Detailed Description	401
9.59.2 Function Documentation	401
9.59.2.1 fasp_dbsr_print()	401
9.59.2.2 fasp_dbsr_read()	402
9.59.2.3 fasp_dbsr_write()	403
9.59.2.4 fasp_dbsr_write_coo()	403
9.59.2.5 fasp_dcoo_print()	403
9.59.2.6 fasp_dcoo_read()	404
9.59.2.7 fasp_dcoo_read1()	404
9.59.2.8 fasp_dcoo_shift_read()	405
9.59.2.9 fasp_dcoo_write()	406
9.59.2.10 fasp_dcoovec_bin_read()	406
9.59.2.11 fasp_dcsr_print()	407
9.59.2.12 fasp_dcsr_read()	407
9.59.2.13 fasp_dcsr_write_coo()	408
9.59.2.14 fasp_dcsr_write_mtx()	408
9.59.2.15 fasp_dcsrvec_read1()	409
9.59.2.16 fasp_dcsrvec_read2()	409
9.59.2.17 fasp_dcsrvec_write1()	410
9.59.2.18 fasp_dcsrvec_write2()	411
9.59.2.19 fasp_dmtx_read()	412
9.59.2.20 fasp_dmtxsym_read()	412
9.59.2.21 fasp_dstr_print()	413

9.59.2.22 fasp_dstr_read()	413
9.59.2.23 fasp_dstr_write()	414
9.59.2.24 fasp_dvec_print()	414
9.59.2.25 fasp_dvec_read()	415
9.59.2.26 fasp_dvec_write()	415
9.59.2.27 fasp_dvecind_read()	416
9.59.2.28 fasp_dvecind_write()	416
9.59.2.29 fasp_hb_read()	417
9.59.2.30 fasp_ivec_print()	417
9.59.2.31 fasp_ivec_read()	418
9.59.2.32 fasp_ivec_write()	418
9.59.2.33 fasp_ivecind_read()	419
9.59.2.34 fasp_matrix_read()	420
9.59.2.35 fasp_matrix_read_bin()	420
9.59.2.36 fasp_matrix_write()	421
9.59.2.37 fasp_vector_read()	422
9.59.2.38 fasp_vector_write()	423
9.59.3 Variable Documentation	424
9.59.3.1 dlength	424
9.59.3.2 ilength	424
9.60 BlaiO.c	424
9.61 BlaOrderingCSR.c File Reference	446
9.61.1 Detailed Description	446
9.61.2 Function Documentation	446
9.61.2.1 fasp_dcsr_CMK_order()	446
9.61.2.2 fasp_dcsr_RCMK_order()	447
9.62 BlaOrderingCSR.c	447
9.63 BlaschwarzSetup.c File Reference	449
9.63.1 Detailed Description	449
9.63.2 Function Documentation	449
9.63.2.1 fasp_dcsr_swz_backward()	450
9.63.2.2 fasp_dcsr_swz_forward()	450
9.63.2.3 fasp_swz_dcsr_setup()	451
9.64 BlaschwarzSetup.c	451
9.65 BlasSmallMat.c File Reference	457
9.65.1 Detailed Description	459
9.65.2 Function Documentation	459
9.65.2.1 fasp_blas_smat_aAxpby()	459
9.65.2.2 fasp_blas_smat_add()	460

9.65.2.3 fasp_blas_smat_axm()	460
9.65.2.4 fasp_blas_smat_mul()	461
9.65.2.5 fasp_blas_smat_mul_nc2()	462
9.65.2.6 fasp_blas_smat_mul_nc3()	462
9.65.2.7 fasp_blas_smat_mul_nc4()	462
9.65.2.8 fasp_blas_smat_mul_nc5()	463
9.65.2.9 fasp_blas_smat_mul_nc7()	463
9.65.2.10 fasp_blas_smat_m xv()	464
9.65.2.11 fasp_blas_smat_m xv_nc2()	465
9.65.2.12 fasp_blas_smat_m xv_nc3()	465
9.65.2.13 fasp_blas_smat_m xv_nc4()	465
9.65.2.14 fasp_blas_smat_m xv_nc5()	466
9.65.2.15 fasp_blas_smat_m xv_nc7()	466
9.65.2.16 fasp_blas_smat_ymAx()	467
9.65.2.17 fasp_blas_smat_ymAx_nc2()	467
9.65.2.18 fasp_blas_smat_ymAx_nc3()	468
9.65.2.19 fasp_blas_smat_ymAx_nc4()	468
9.65.2.20 fasp_blas_smat_ymAx_nc5()	470
9.65.2.21 fasp_blas_smat_ymAx_nc7()	470
9.65.2.22 fasp_blas_smat_ypAx()	472
9.65.2.23 fasp_blas_smat_ypAx_nc2()	472
9.65.2.24 fasp_blas_smat_ypAx_nc3()	474
9.65.2.25 fasp_blas_smat_ypAx_nc4()	474
9.65.2.26 fasp_blas_smat_ypAx_nc5()	475
9.65.2.27 fasp_blas_smat_ypAx_nc7()	475
9.66 BlaSmallMat.c	476
9.67 BlaSmallMatInv.c File Reference	485
9.67.1 Detailed Description	486
9.67.2 Macro Definition Documentation	486
9.67.2.1 SWAP	486
9.67.3 Function Documentation	486
9.67.3.1 fasp_smat_identity()	486
9.67.3.2 fasp_smat_identity_nc2()	487
9.67.3.3 fasp_smat_identity_nc3()	487
9.67.3.4 fasp_smat_identity_nc5()	487
9.67.3.5 fasp_smat_identity_nc7()	488
9.67.3.6 fasp_smat_inv()	488
9.67.3.7 fasp_smat_inv_nc()	489
9.67.3.8 fasp_smat_inv_nc2()	489

9.67.3.9 fasp_smat_inv_nc3()	489
9.67.3.10 fasp_smat_inv_nc4()	490
9.67.3.11 fasp_smat_inv_nc5()	490
9.67.3.12 fasp_smat_inv_nc7()	490
9.67.3.13 fasp_smat_invp_nc()	491
9.67.3.14 fasp_smat_Linf()	491
9.68 BlaSmallMatInv.c	492
9.69 BlaSmallMatLU.c File Reference	501
9.69.1 Detailed Description	502
9.69.2 Function Documentation	502
9.69.2.1 fasp_smat_lu_decomp()	502
9.69.2.2 fasp_smat_lu_solve()	503
9.70 BlaSmallMatLU.c	503
9.71 BlaSparseBLC.c File Reference	505
9.71.1 Detailed Description	505
9.71.2 Function Documentation	505
9.71.2.1 fasp_dblc_free()	505
9.72 BlaSparseBLC.c	506
9.73 BlaSparseBSR.c File Reference	506
9.73.1 Detailed Description	507
9.73.2 Function Documentation	507
9.73.2.1 fasp_dbsr_alloc()	508
9.73.2.2 fasp_dbsr_cp()	508
9.73.2.3 fasp_dbsr_create()	509
9.73.2.4 fasp_dbsr_diaginv()	509
9.73.2.5 fasp_dbsr_diaginv2()	510
9.73.2.6 fasp_dbsr_diaginv3()	510
9.73.2.7 fasp_dbsr_diaginv4()	511
9.73.2.8 fasp_dbsr_diagLU()	511
9.73.2.9 fasp_dbsr_diagLU2()	512
9.73.2.10 fasp_dbsr_diagpref()	512
9.73.2.11 fasp_dbsr_free()	514
9.73.2.12 fasp_dbsr_getblk()	514
9.73.2.13 fasp_dbsr_getdiag()	515
9.73.2.14 fasp_dbsr_getdiaginv()	516
9.73.2.15 fasp_dbsr_merge_col()	516
9.73.2.16 fasp_dbsr_perm()	517
9.73.2.17 fasp_dbsr_trans()	517
9.74 BlaSparseBSR.c	518

9.75 BlaSparseCheck.c File Reference	542
9.75.1 Detailed Description	543
9.75.2 Function Documentation	543
9.75.2.1 fasp_check_dCSRmat()	543
9.75.2.2 fasp_check_diagdom()	543
9.75.2.3 fasp_check_diagpos()	544
9.75.2.4 fasp_check_diagzero()	544
9.75.2.5 fasp_check_iCSRmat()	545
9.75.2.6 fasp_check_ordering()	545
9.75.2.7 fasp_check_symm()	545
9.76 BlaSparseCheck.c	547
9.77 BlaSparseCOO.c File Reference	551
9.77.1 Detailed Description	551
9.77.2 Function Documentation	551
9.77.2.1 fasp_dcoo_alloc()	551
9.77.2.2 fasp_dcoo_create()	552
9.77.2.3 fasp_dcoo_free()	552
9.77.2.4 fasp_dcoo_shift()	553
9.78 BlaSparseCOO.c	553
9.79 BlaSparseCSR.c File Reference	554
9.79.1 Detailed Description	556
9.79.2 Function Documentation	556
9.79.2.1 dCSRmat_Multicoloring()	556
9.79.2.2 dCSRmat_Multicoloring_Strong_Coupled()	557
9.79.2.3 dCSRmat_Multicoloring_Theta()	557
9.79.2.4 fasp_dcsr_alloc()	558
9.79.2.5 fasp_dcsr_bandwidth()	558
9.79.2.6 fasp_dcsr_compress()	559
9.79.2.7 fasp_dcsr_compress_inplace()	559
9.79.2.8 fasp_dcsr_cp()	560
9.79.2.9 fasp_dcsr_create()	560
9.79.2.10 fasp_dcsr_diagpref()	561
9.79.2.11 fasp_dcsr_free()	561
9.79.2.12 fasp_dcsr_getblk()	562
9.79.2.13 fasp_dcsr_getcol()	562
9.79.2.14 fasp_dcsr_getdiag()	563
9.79.2.15 fasp_dcsr_multicoloring()	563
9.79.2.16 fasp_dcsr_perm()	564
9.79.2.17 fasp_dcsr_permz()	564

9.79.2.18 fasp_dcsr_regdiag()	565
9.79.2.19 fasp_dcsr_shift()	565
9.79.2.20 fasp_dcsr_sort()	566
9.79.2.21 fasp_dcsr_sortz()	566
9.79.2.22 fasp_dcsr_symdiagscale()	567
9.79.2.23 fasp_dcsr_sympart()	567
9.79.2.24 fasp_dcsr_trans()	568
9.79.2.25 fasp_dcsr_transpose()	568
9.79.2.26 fasp_dcsr_transz()	569
9.79.2.27 fasp_icsr_cp()	569
9.79.2.28 fasp_icsr_create()	570
9.79.2.29 fasp_icsr_free()	570
9.79.2.30 fasp_icsr_trans()	571
9.79.2.31 fasp_smoother_dcsr_gs_multicolor()	571
9.80 BlaSparseCSR.c	571
9.81 BlaSparseCSRL.c File Reference	593
9.81.1 Detailed Description	593
9.81.2 Function Documentation	594
9.81.2.1 fasp_dcsrl_create()	594
9.81.2.2 fasp_dcsrl_free()	594
9.82 BlaSparseCSRL.c	594
9.83 BlaSparseSTR.c File Reference	595
9.83.1 Detailed Description	595
9.83.2 Function Documentation	596
9.83.2.1 fasp_dstr_alloc()	596
9.83.2.2 fasp_dstr_cp()	596
9.83.2.3 fasp_dstr_create()	597
9.83.2.4 fasp_dstr_free()	598
9.84 BlaSparseSTR.c	598
9.85 BlaSparseUtil.c File Reference	599
9.85.1 Detailed Description	600
9.85.2 Function Documentation	601
9.85.2.1 fasp_sparse_aat_()	601
9.85.2.2 fasp_sparse_abyb_()	601
9.85.2.3 fasp_sparse_abybms_()	602
9.85.2.4 fasp_sparse_aplbms_()	603
9.85.2.5 fasp_sparse_aplusb_()	603
9.85.2.6 fasp_sparse_iit_()	604
9.85.2.7 fasp_sparse_mis_()	604

9.85.2.8 fasp_sparse_rapcmp_()	605
9.85.2.9 fasp_sparse_rapms_()	606
9.85.2.10 fasp_sparse_wta_()	606
9.85.2.11 fasp_sparse_wtams_()	607
9.85.2.12 fasp_sparse_ytx_()	608
9.85.2.13 fasp_sparse_ytxbig_()	608
9.86 BlaSparseUtil.c	609
9.87 BlaSpmvBLC.c File Reference	618
9.87.1 Detailed Description	618
9.87.2 Function Documentation	618
9.87.2.1 fasp_blas_dblc_aApy()	618
9.87.2.2 fasp_blas_dblc_mxv()	619
9.87.2.3 fasp_blas_ldblc_aApy()	619
9.88 BlaSpmvBLC.c	620
9.89 BlaSpmvBSR.c File Reference	624
9.89.1 Detailed Description	625
9.89.2 Function Documentation	625
9.89.2.1 fasp_blas_dbsr_aApyby()	625
9.89.2.2 fasp_blas_dbsr_aApy()	626
9.89.2.3 fasp_blas_dbsr_aApy_agg()	626
9.89.2.4 fasp_blas_dbsr_axm()	627
9.89.2.5 fasp_blas_dbsr_mxm()	627
9.89.2.6 fasp_blas_dbsr_mxv()	628
9.89.2.7 fasp_blas_dbsr_mxv_agg()	628
9.89.2.8 fasp_blas_dbsr_rap()	629
9.89.2.9 fasp_blas_dbsr_rap1()	630
9.89.2.10 fasp_blas_dbsr_rap_agg()	630
9.90 BlaSpmvBSR.c	631
9.91 BlaSpmvCSR.c File Reference	696
9.91.1 Detailed Description	697
9.91.2 Function Documentation	698
9.91.2.1 fasp_blas_dcsr_aApy()	698
9.91.2.2 fasp_blas_dcsr_aApy_agg()	698
9.91.2.3 fasp_blas_dcsr_add()	699
9.91.2.4 fasp_blas_dcsr_axm()	699
9.91.2.5 fasp_blas_dcsr_mxm()	700
9.91.2.6 fasp_blas_dcsr_mxv()	700
9.91.2.7 fasp_blas_dcsr_mxv_agg()	701
9.91.2.8 fasp_blas_dcsr_ptap()	701

9.91.2.9 fasp_blas_dcsr_rap()	702
9.91.2.10 fasp_blas_dcsr_rap2()	702
9.91.2.11 fasp_blas_dcsr_rap4()	703
9.91.2.12 fasp_blas_dcsr_rap_agg()	704
9.91.2.13 fasp_blas_dcsr_rap_agg1()	704
9.91.2.14 fasp_blas_dcsr_vmv()	705
9.91.2.15 fasp_blas_ldcsr_aApy()	705
9.91.3 Variable Documentation	706
9.91.3.1 total_alloc_count	706
9.91.3.2 total_alloc_mem	706
9.92 BlaSpmvCSR.c	706
9.93 BlaSpmvCSR.c File Reference	730
9.93.1 Detailed Description	730
9.93.2 Function Documentation	731
9.93.2.1 fasp_blas_dcsr_mxv()	731
9.94 BlaSpmvCSR.c	731
9.95 BlaSpmvSTR.c File Reference	733
9.95.1 Detailed Description	733
9.95.2 Function Documentation	733
9.95.2.1 fasp_blas_dstr_aApy()	733
9.95.2.2 fasp_blas_dstr_diagscale()	734
9.95.2.3 fasp_blas_dstr_mxv()	734
9.96 BlaSpmvSTR.c	735
9.97 BlaVector.c File Reference	754
9.97.1 Detailed Description	755
9.97.2 Function Documentation	755
9.97.2.1 fasp_blas_dvec_axpy()	755
9.97.2.2 fasp_blas_dvec_axpyz()	756
9.97.2.3 fasp_blas_dvec_dotprod()	756
9.97.2.4 fasp_blas_dvec_norm1()	757
9.97.2.5 fasp_blas_dvec_norm2()	757
9.97.2.6 fasp_blas_dvec_norminf()	757
9.97.2.7 fasp_blas_dvec_reterr()	758
9.98 BlaVector.c	758
9.99 ltrSmootherBSR.c File Reference	761
9.99.1 Detailed Description	762
9.99.2 Function Documentation	762
9.99.2.1 fasp_smoother_dbsr_gs()	763
9.99.2.2 fasp_smoother_dbsr_gs1()	763

9.99.2.3 fasp_smoother_dbsr_gs_ascend()	764
9.99.2.4 fasp_smoother_dbsr_gs_ascend1()	764
9.99.2.5 fasp_smoother_dbsr_gs_descend()	765
9.99.2.6 fasp_smoother_dbsr_gs_descend1()	765
9.99.2.7 fasp_smoother_dbsr_gs_order1()	766
9.99.2.8 fasp_smoother_dbsr_gs_order2()	766
9.99.2.9 fasp_smoother_dbsr_ilu()	767
9.99.2.10 fasp_smoother_dbsr_jacobi()	768
9.99.2.11 fasp_smoother_dbsr_jacobi1()	768
9.99.2.12 fasp_smoother_dbsr_jacobi_setup()	769
9.99.2.13 fasp_smoother_dbsr_sor()	769
9.99.2.14 fasp_smoother_dbsr_sor1()	770
9.99.2.15 fasp_smoother_dbsr_sor_ascend()	770
9.99.2.16 fasp_smoother_dbsr_sor_descend()	771
9.99.2.17 fasp_smoother_dbsr_sor_order()	771
9.99.3 Variable Documentation	772
9.99.3.1 ilu_solve_time	772
9.100 ltrSmootherBSR.c	772
9.101 ltrSmootherCSR.c File Reference	789
9.101.1 Detailed Description	790
9.101.2 Function Documentation	790
9.101.2.1 fasp_smoother_dcsr_gs()	790
9.101.2.2 fasp_smoother_dcsr_gs_cf()	791
9.101.2.3 fasp_smoother_dcsr_ilu()	792
9.101.2.4 fasp_smoother_dcsr_jacobi()	792
9.101.2.5 fasp_smoother_dcsr_kaczmarz()	793
9.101.2.6 fasp_smoother_dcsr_L1diag()	794
9.101.2.7 fasp_smoother_dcsr_sgs()	794
9.101.2.8 fasp_smoother_dcsr_sor()	795
9.101.2.9 fasp_smoother_dcsr_sor_cf()	796
9.102 ltrSmootherCSR.c	796
9.103 ltrSmootherCSRcr.c File Reference	813
9.103.1 Detailed Description	813
9.103.2 Function Documentation	813
9.103.2.1 fasp_smoother_dcsr_gscr()	813
9.104 ltrSmootherCSRcr.c	814
9.105 ltrSmootherCSRpoly.c File Reference	815
9.105.1 Detailed Description	815
9.105.2 Function Documentation	816

9.105.2.1 fasp_smoother_dcsr_poly()	816
9.105.2.2 fasp_smoother_dcsr_poly_old()	816
9.106 ltrSmootherCSRpoly.c	817
9.107 ltrSmootherSTR.c File Reference	823
9.107.1 Detailed Description	824
9.107.2 Function Documentation	824
9.107.2.1 fasp_generate_diaginv_block()	824
9.107.2.2 fasp_smoother_dstr_gs()	825
9.107.2.3 fasp_smoother_dstr_gs1()	825
9.107.2.4 fasp_smoother_dstr_gs_ascend()	826
9.107.2.5 fasp_smoother_dstr_gs_cf()	827
9.107.2.6 fasp_smoother_dstr_gs_descend()	827
9.107.2.7 fasp_smoother_dstr_gs_order()	828
9.107.2.8 fasp_smoother_dstr_jacobi()	828
9.107.2.9 fasp_smoother_dstr_jacobi1()	829
9.107.2.10 fasp_smoother_dstr_sor()	829
9.107.2.11 fasp_smoother_dstr_sor1()	830
9.107.2.12 fasp_smoother_dstr_sor_ascend()	830
9.107.2.13 fasp_smoother_dstr_sor_cf()	831
9.107.2.14 fasp_smoother_dstr_sor_descend()	832
9.107.2.15 fasp_smoother_dstr_sor_order()	832
9.107.2.16 fasp_smoother_dstr_swz()	833
9.108 ltrSmootherSTR.c	833
9.109 KryPbcgs.c File Reference	852
9.109.1 Detailed Description	853
9.109.2 Function Documentation	853
9.109.2.1 fasp_solver_dblc_pbcgs()	853
9.109.2.2 fasp_solver_dbsr_pbcgs()	854
9.109.2.3 fasp_solver_dcsr_pbcgs()	855
9.109.2.4 fasp_solver_dstr_pbcgs()	856
9.109.2.5 fasp_solver_pbcgs()	856
9.110 KryPbcgs.c	857
9.111 KryPcg.c File Reference	876
9.111.1 Detailed Description	877
9.111.2 Function Documentation	878
9.111.2.1 fasp_solver_dblc_pcg()	878
9.111.2.2 fasp_solver_dbsr_pcg()	879
9.111.2.3 fasp_solver_dcsr_pcg()	880
9.111.2.4 fasp_solver_dstr_pcg()	880

9.111.2.5 fasp_solver_pcg()	881
9.112 KryPcg.c	882
9.113 KryPgcg.c File Reference	899
9.113.1 Detailed Description	899
9.113.2 Function Documentation	899
9.113.2.1 fasp_solver_dcsr_pgcg()	899
9.113.2.2 fasp_solver_pgcg()	900
9.114 KryPgcg.c	901
9.115 KryPgcr.c File Reference	904
9.115.1 Detailed Description	905
9.115.2 Function Documentation	905
9.115.2.1 fasp_solver_dblc_pgcr()	905
9.115.2.2 fasp_solver_dcsr_pgcr()	906
9.116 KryPgcr.c	907
9.117 KryPgmres.c File Reference	911
9.117.1 Detailed Description	912
9.117.2 Function Documentation	912
9.117.2.1 fasp_solver_dblc_pgmres()	912
9.117.2.2 fasp_solver_dbsr_pgmres()	913
9.117.2.3 fasp_solver_dcsr_pgmres()	914
9.117.2.4 fasp_solver_dstr_pgmres()	915
9.117.2.5 fasp_solver_pgmres()	916
9.118 KryPgmres.c	916
9.119 KryPminres.c File Reference	935
9.119.1 Detailed Description	936
9.119.2 Function Documentation	936
9.119.2.1 fasp_solver_dblc_pminres()	936
9.119.2.2 fasp_solver_dcsr_pminres()	937
9.119.2.3 fasp_solver_dstr_pminres()	938
9.119.2.4 fasp_solver_pminres()	938
9.120 KryPminres.c	939
9.121 KryPvfgmres.c File Reference	958
9.121.1 Detailed Description	959
9.121.2 Function Documentation	959
9.121.2.1 fasp_solver_dblc_pvfgmres()	959
9.121.2.2 fasp_solver_dbsr_pvfgmres()	960
9.121.2.3 fasp_solver_dcsr_pvfgmres()	961
9.121.2.4 fasp_solver_pvfgmres()	962
9.122 KryPvfgmres.c	963

9.123 KryPvgmres.c File Reference	977
9.123.1 Detailed Description	978
9.123.2 Function Documentation	978
9.123.2.1 fasp_solver_dblc_pvgmres()	978
9.123.2.2 fasp_solver_dbsr_pvgmres()	979
9.123.2.3 fasp_solver_dcsr_pvgmres()	980
9.123.2.4 fasp_solver_dstr_pvgmres()	981
9.123.2.5 fasp_solver_pvgmres()	981
9.124 KryPvgmres.c	982
9.125 KrySPbcgs.c File Reference	1002
9.125.1 Detailed Description	1002
9.125.2 Function Documentation	1003
9.125.2.1 fasp_solver_dblc_spbcgs()	1003
9.125.2.2 fasp_solver_dbsr_spbcgs()	1004
9.125.2.3 fasp_solver_dcsr_spbcgs()	1004
9.125.2.4 fasp_solver_dstr_spbcgs()	1005
9.126 KrySPbcgs.c	1006
9.127 KrySPcg.c File Reference	1024
9.127.1 Detailed Description	1025
9.127.2 Function Documentation	1025
9.127.2.1 fasp_solver_dblc_spcg()	1025
9.127.2.2 fasp_solver_dcsr_spcg()	1026
9.127.2.3 fasp_solver_dstr_spcg()	1027
9.128 KrySPcg.c	1027
9.129 KrySPgmres.c File Reference	1039
9.129.1 Detailed Description	1040
9.129.2 Function Documentation	1040
9.129.2.1 fasp_solver_dblc_spgmres()	1040
9.129.2.2 fasp_solver_dbsr_spgmres()	1041
9.129.2.3 fasp_solver_dcsr_spgmres()	1042
9.129.2.4 fasp_solver_dstr_spgmres()	1043
9.130 KrySPgmres.c	1043
9.131 KrySPminres.c File Reference	1059
9.131.1 Detailed Description	1060
9.131.2 Function Documentation	1060
9.131.2.1 fasp_solver_dblc_spmminres()	1060
9.131.2.2 fasp_solver_dcsr_spmminres()	1061
9.131.2.3 fasp_solver_dstr_spmminres()	1062
9.132 KrySPminres.c	1062

9.133 KrySPvgmres.c File Reference	1079
9.133.1 Detailed Description	1079
9.133.2 Function Documentation	1080
9.133.2.1 fasp_solver_dblc_spvgmres()	1080
9.133.2.2 fasp_solver_dbsr_spvgmres()	1080
9.133.2.3 fasp_solver_dcsr_spvgmres()	1081
9.133.2.4 fasp_solver_dstr_spvgmres()	1082
9.134 KrySPvgmres.c	1083
9.135 PreAMGCoarsenCR.c File Reference	1101
9.135.1 Detailed Description	1101
9.135.2 Macro Definition Documentation	1101
9.135.2.1 AMG_COARSEN_CR	1101
9.135.3 Function Documentation	1101
9.135.3.1 fasp_amg_coarsening_cr()	1102
9.136 PreAMGCoarsenCR.c	1102
9.137 PreAMGCoarsenRS.c File Reference	1106
9.137.1 Detailed Description	1107
9.137.2 Function Documentation	1107
9.137.2.1 fasp_amg_coarsening_rs()	1107
9.138 PreAMGCoarsenRS.c	1108
9.139 PreAMGInterp.c File Reference	1130
9.139.1 Detailed Description	1130
9.139.2 Function Documentation	1131
9.139.2.1 fasp_amg_interp()	1131
9.140 PreAMGInterp.c	1131
9.141 PreAMGInterpEM.c File Reference	1141
9.141.1 Detailed Description	1141
9.141.2 Function Documentation	1142
9.141.2.1 fasp_amg_interp_em()	1142
9.142 PreAMGInterpEM.c	1142
9.143 PreAMGSetupCR.c File Reference	1151
9.143.1 Detailed Description	1151
9.143.2 Function Documentation	1151
9.143.2.1 fasp_amg_setup_cr()	1151
9.144 PreAMGSetupCR.c	1152
9.145 PreAMGSetupRS.c File Reference	1153
9.145.1 Detailed Description	1153
9.145.2 Function Documentation	1154
9.145.2.1 fasp_amg_setup_rs()	1154

9.146 PreAMGSetupRS.c	1154
9.147 PreAMGSetupSA.c File Reference	1158
9.147.1 Detailed Description	1158
9.147.2 Function Documentation	1159
9.147.2.1 fasp_amg_setup_sa()	1159
9.148 PreAMGSetupSA.c	1159
9.149 PreAMGSetupSABSR.c File Reference	1168
9.149.1 Detailed Description	1169
9.149.2 Function Documentation	1169
9.149.2.1 fasp_amg_setup_sa_bsr()	1169
9.150 PreAMGSetupSABSR.c	1170
9.151 PreAMGSetupUA.c File Reference	1175
9.151.1 Detailed Description	1175
9.151.2 Function Documentation	1175
9.151.2.1 fasp_amg_setup_ua()	1175
9.152 PreAMGSetupUA.c	1176
9.153 PreAMGSetupUABSR.c File Reference	1181
9.153.1 Detailed Description	1181
9.153.2 Function Documentation	1181
9.153.2.1 fasp_amg_setup_ua_bsr()	1181
9.154 PreAMGSetupUABSR.c	1182
9.155 PreBLC.c File Reference	1186
9.155.1 Detailed Description	1187
9.155.2 Function Documentation	1187
9.155.2.1 fasp_precond_dblc_diag_3()	1187
9.155.2.2 fasp_precond_dblc_diag_3_amg()	1187
9.155.2.3 fasp_precond_dblc_diag_4()	1188
9.155.2.4 fasp_precond_dblc_lower_3()	1188
9.155.2.5 fasp_precond_dblc_lower_3_amg()	1190
9.155.2.6 fasp_precond_dblc_lower_4()	1190
9.155.2.7 fasp_precond_dblc_SGS_3()	1192
9.155.2.8 fasp_precond_dblc_SGS_3_amg()	1192
9.155.2.9 fasp_precond_dblc_sweeping()	1194
9.155.2.10 fasp_precond_dblc_upper_3()	1194
9.155.2.11 fasp_precond_dblc_upper_3_amg()	1196
9.156 PreBLC.c	1196
9.157 PreBSR.c File Reference	1208
9.157.1 Detailed Description	1209
9.157.2 Function Documentation	1209

9.157.2.1 fasp_precond_dbsr_amg()	1209
9.157.2.2 fasp_precond_dbsr_amg_nk()	1209
9.157.2.3 fasp_precond_dbsr_diag()	1210
9.157.2.4 fasp_precond_dbsr_diag_nc2()	1210
9.157.2.5 fasp_precond_dbsr_diag_nc3()	1211
9.157.2.6 fasp_precond_dbsr_diag_nc5()	1211
9.157.2.7 fasp_precond_dbsr_diag_nc7()	1212
9.157.2.8 fasp_precond_dbsr_ilu()	1213
9.157.2.9 fasp_precond_dbsr_ilu_ls_omp()	1213
9.157.2.10 fasp_precond_dbsr_ilu_mc_omp()	1214
9.157.2.11 fasp_precond_dbsr_namli()	1214
9.158 PreBSR.c	1215
9.159 PreCSR.c File Reference	1227
9.159.1 Detailed Description	1228
9.159.2 Function Documentation	1228
9.159.2.1 fasp_precond_amg()	1228
9.159.2.2 fasp_precond_amg_nk()	1228
9.159.2.3 fasp_precond_amli()	1229
9.159.2.4 fasp_precond_diag()	1229
9.159.2.5 fasp_precond_famg()	1230
9.159.2.6 fasp_precond_ilu()	1230
9.159.2.7 fasp_precond_ilu_backward()	1231
9.159.2.8 fasp_precond_ilu_forward()	1231
9.159.2.9 fasp_precond_namli()	1232
9.159.2.10 fasp_precond_setup()	1232
9.159.2.11 fasp_precond_swz()	1233
9.160 PreCSR.c	1233
9.161 PreDataInit.c File Reference	1239
9.161.1 Detailed Description	1239
9.161.2 Function Documentation	1240
9.161.2.1 fasp_amg_data_bsr_create()	1240
9.161.2.2 fasp_amg_data_bsr_free()	1240
9.161.2.3 fasp_amg_data_create()	1241
9.161.2.4 fasp_amg_data_free()	1241
9.161.2.5 fasp_ilu_data_create()	1242
9.161.2.6 fasp_ilu_data_free()	1242
9.161.2.7 fasp_precond_data_init()	1243
9.161.2.8 fasp_swz_data_free()	1243
9.162 PreDataInit.c	1243

9.163 PreMGCycle.c File Reference	1247
9.163.1 Detailed Description	1247
9.163.2 Function Documentation	1247
9.163.2.1 fasp_solver_mgcycle()	1247
9.163.2.2 fasp_solver_mgcycle_bsr()	1248
9.164 PreMGCycle.c	1248
9.165 PreMGCycleFull.c File Reference	1254
9.165.1 Detailed Description	1255
9.165.2 Function Documentation	1255
9.165.2.1 fasp_solver_fmgcycle()	1255
9.166 PreMGCycleFull.c	1256
9.167 PreMGRecur.c File Reference	1260
9.167.1 Detailed Description	1260
9.167.2 Function Documentation	1260
9.167.2.1 fasp_solver_mgrecur()	1260
9.168 PreMGRecur.c	1261
9.169 PreMGRecurAMLI.c File Reference	1263
9.169.1 Detailed Description	1263
9.169.2 Function Documentation	1263
9.169.2.1 fasp_amg_amli_coef()	1264
9.169.2.2 fasp_solver_amli()	1264
9.169.2.3 fasp_solver_namli()	1265
9.169.2.4 fasp_solver_namli_bsr()	1265
9.170 PreMGRecurAMLI.c	1266
9.171 PreMGSolve.c File Reference	1274
9.171.1 Detailed Description	1275
9.171.2 Function Documentation	1275
9.171.2.1 fasp_amg_solve()	1275
9.171.2.2 fasp_amg_solve_amli()	1276
9.171.2.3 fasp_amg_solve_namli()	1276
9.171.2.4 fasp_famg_solve()	1277
9.172 PreMGSolve.c	1277
9.173 PreSTR.c File Reference	1281
9.173.1 Detailed Description	1281
9.173.2 Function Documentation	1282
9.173.2.1 fasp_precond_dstr_blockgs()	1282
9.173.2.2 fasp_precond_dstr_diag()	1282
9.173.2.3 fasp_precond_dstr_ilu0()	1283
9.173.2.4 fasp_precond_dstr_ilu0_backward()	1283

9.173.2.5 fasp_precond_dstr_ilu0_forward()	1284
9.173.2.6 fasp_precond_dstr_ilu1()	1284
9.173.2.7 fasp_precond_dstr_ilu1_backward()	1284
9.173.2.8 fasp_precond_dstr_ilu1_forward()	1285
9.174 PreSTR.c	1285
9.175 SolAMG.c File Reference	1307
9.175.1 Detailed Description	1307
9.175.2 Function Documentation	1307
9.175.2.1 fasp_solver_amg()	1307
9.176 SolAMG.c	1308
9.177 SolBLC.c File Reference	1309
9.177.1 Detailed Description	1310
9.177.2 Function Documentation	1310
9.177.2.1 fasp_solver_dblc_itsolver()	1310
9.177.2.2 fasp_solver_dblc_krylov()	1311
9.177.2.3 fasp_solver_dblc_krylov_block3()	1311
9.177.2.4 fasp_solver_dblc_krylov_block4()	1312
9.177.2.5 fasp_solver_dblc_krylov_sweeping()	1313
9.178 SolBLC.c	1314
9.179 SolBSR.c File Reference	1319
9.179.1 Detailed Description	1320
9.179.2 Function Documentation	1320
9.179.2.1 fasp_solver_dbsr_itsolver()	1320
9.179.2.2 fasp_solver_dbsr_krylov()	1321
9.179.2.3 fasp_solver_dbsr_krylov_amg()	1322
9.179.2.4 fasp_solver_dbsr_krylov_amg_nk()	1322
9.179.2.5 fasp_solver_dbsr_krylov_diag()	1323
9.179.2.6 fasp_solver_dbsr_krylov_ilu()	1324
9.179.2.7 fasp_solver_dbsr_krylov_nk_amg()	1324
9.180 SolBSR.c	1325
9.181 SolCSR.c File Reference	1333
9.181.1 Detailed Description	1333
9.181.2 Function Documentation	1334
9.181.2.1 fasp_solver_dcsr_itsolver()	1334
9.181.2.2 fasp_solver_dcsr_itsolver_s()	1334
9.181.2.3 fasp_solver_dcsr_krylov()	1335
9.181.2.4 fasp_solver_dcsr_krylov_amg()	1336
9.181.2.5 fasp_solver_dcsr_krylov_amg_nk()	1336
9.181.2.6 fasp_solver_dcsr_krylov_diag()	1337

9.181.2.7 fasp_solver_dcsr_krylov_ilu()	1338
9.181.2.8 fasp_solver_dcsr_krylov_ilu_M()	1338
9.181.2.9 fasp_solver_dcsr_krylov_s()	1339
9.181.2.10 fasp_solver_dcsr_krylov_swz()	1339
9.182 SolCSR.c	1340
9.183 SolFAMG.c File Reference	1348
9.183.1 Detailed Description	1349
9.183.2 Function Documentation	1349
9.183.2.1 fasp_solver_famg()	1349
9.184 SolFAMG.c	1349
9.185 SolGMGPoisson.c File Reference	1351
9.185.1 Detailed Description	1351
9.185.2 Function Documentation	1352
9.185.2.1 fasp_poisson_fgmg1d()	1352
9.185.2.2 fasp_poisson_fgmg2d()	1352
9.185.2.3 fasp_poisson_fgmg3d()	1353
9.185.2.4 fasp_poisson_gmg1d()	1354
9.185.2.5 fasp_poisson_gmg2d()	1354
9.185.2.6 fasp_poisson_gmg3d()	1355
9.185.2.7 fasp_poisson_gmgcg1d()	1356
9.185.2.8 fasp_poisson_gmgcg2d()	1356
9.185.2.9 fasp_poisson_gmgcg3d()	1357
9.186 SolGMGPoisson.c	1358
9.187 SolMatFree.c File Reference	1369
9.187.1 Detailed Description	1369
9.187.2 Function Documentation	1369
9.187.2.1 fasp_solver_itsolver()	1369
9.187.2.2 fasp_solver_krylov()	1370
9.187.2.3 fasp_solver_matfree_init()	1371
9.188 SolMatFree.c	1371
9.189 SolSTR.c File Reference	1373
9.189.1 Detailed Description	1374
9.189.2 Function Documentation	1374
9.189.2.1 fasp_solver_dstr_itsolver()	1374
9.189.2.2 fasp_solver_dstr_krylov()	1375
9.189.2.3 fasp_solver_dstr_krylov_blockgs()	1375
9.189.2.4 fasp_solver_dstr_krylov_diag()	1376
9.189.2.5 fasp_solver_dstr_krylov_ilu()	1377
9.190 SolSTR.c	1377

9.191 SolWrapper.c File Reference	1381
9.191.1 Detailed Description	1382
9.191.2 Function Documentation	1382
9.191.2.1 fasp_fwrapper_dbsr_krylov_amg_()	1382
9.191.2.2 fasp_fwrapper_dbsr_krylov_ilu_()	1383
9.191.2.3 fasp_fwrapper_dcsr_amg_()	1383
9.191.2.4 fasp_fwrapper_dcsr_krylov_amg_()	1384
9.191.2.5 fasp_fwrapper_dcsr_krylov_ilu_()	1385
9.191.2.6 fasp_fwrapper_dcsr_pardiso_()	1386
9.192 SolWrapper.c	1386
Index	1391

Chapter 1

Introduction

Over the last few decades, researchers have expended significant effort on developing efficient iterative methods for solving discretized partial differential equations (PDEs). Though these efforts have yielded many mathematically optimal solvers such as the multigrid method, the unfortunate reality is that multigrid methods have not been much used in practical applications. This marked gap between theory and practice is mainly due to the fragility of traditional multigrid (MG) methodology and the complexity of its implementation. We aim to develop techniques and the corresponding software that will narrow this gap, specifically by developing mathematically optimal solvers that are robust and easy to use in practice.

We believe that there is no one-size-for-all solution method for discrete linear systems from different applications. And, efficient iterative solvers can be constructed by taking the properties of PDEs and discretizations into account. In this project, we plan to construct a pool of discrete problems arising from partial differential equations (PDEs) or PDE systems and efficient linear solvers for these problems. We mainly utilize the methodology of Auxiliary Space Preconditioning (ASP) to construct efficient linear solvers. Due to this reason, this software package is called Fast Auxiliary Space Preconditioning or FASP for short.

The levels of abstraction are designed as follows:

- Level 0 (Aux*.c): Auxiliary functions (timing, memory, threading, ...)
- Level 1 (Bla*.c): Basic linear algebra subroutines (SpMV, RAP, ILU, SWZ, ...)
- Level 2 (Itr*.c): Iterative methods and smoothers (Jacobi, GS, SOR, Poly, ...)
- Level 3 (Kry*.c): Krylov iterative methods (CG, BiCGstab, MinRes, GMRES, ...)
- Level 4 (Pre*.c): Preconditioners (GMG, AMG, FAMG, ...)
- Level 5 (Sol*.c): User interface for FASP solvers (Solvers, wrappers, ...)
- Level x (Xtr*.c): Interface to external packages (Mumps, Umfpack, ...)

FASP contains the kernel part and several applications (ranging from fluid dynamics to reservoir simulation). The kernel part is open-source and licensed under GNU Lesser General Public License or LGPL version 3.0 or later. Some of the applications contain contributions from and owned partially by other parties.

For the moment, FASP is under alpha testing. If you wish to obtain a current version of FASP or you have any questions, feel free to contact us at faspdev@gmail.com.

This software distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

Chapter 2

How to obtain FASP

The most updated version of FASP can be downloaded from

<http://www.multigrid.org/fasp/download/faspsolver.zip>

We use Git as our main version control tool. Git is easy to use and it is available at all OS platforms. For people who is interested in the developer version, you can obtain the FASP package with Git:

```
$ git clone git@github.com:FaspDevTeam/faspsolver.git
```

will give you the developer version of the FASP package.

Chapter 3

Building and Installation

This is a simple instruction on building and testing. For more details, please refer to the README files and the short [User's Guide](#) in "faspolver/doc/".

To compile, you need a Fortran and a C compiler. First, you can type in the "faspolver/" root directory:

```
$ mkdir Build; cd Build; cmake ..
```

which will config the environment automatically. And, then, you can need to type:

```
$ make install
```

which will make the FASP shared static library and install to PREFIX/. By default, FASP libraries and executables will be installed in the FASP home directory "faspolver/".

There is a simple GUI tool for building and installing FASP included in the package. You need Tcl/Tk support in your computer. You may call this GUI by run in the root directory:

```
$ wish fasp_install.tcl
```

If you need to see the detailed usage of "make" or need any help, please type:

```
$ make help
```

After installation, tutorial examples can be found in "tutorial/".

Chapter 4

Developers

Project leader:

- Xu, Jinchao (Penn State University, USA)

Project coordinator:

- Zhang, Chensong (Chinese Academy of Sciences, China)

Current active developers (in alphabetic order):

- Feng, Chunsheng (Xiangtan University, China)
- Zhang, Chensong (Chinese Academy of Sciences, China)

With contributions from (in alphabetic order):

- Brannick, James (Penn State University, USA)
- Chen, Long (University of California, Irvine, USA)
- Hu, Xiaozhe (Tufts University, USA)
- Huang, Feiteng (Sichuan University, China)
- Huang, Xuehai (Shanghai Jiaotong University, China)
- Li, Zheng (Xiangtan University, China)
- Qiao, Changhe (Penn State University, USA)
- Shu, Shi (Xiangtan University, China)
- Sun, Pengtao (University of Nevada, Las Vegas, USA)
- Yang, Kai (Penn State University, USA)

- Yue, Xiaoqiang (Xiangtan University, China)
- Wang, Lu (LLNL, USA)
- Wang, Ziteng (University of Alabama, USA)
- Zhang, Shiquan (Sichuan University, China)
- Zhang, Shuo (Chinese Academy of Sciences, China)
- Zhang, Hongxuan (Penn State Univeristy, USA)
- Zhang, Weifeng (Kunming University of Science and Technology, China)
- Zhou, Zhiyang (Xiangtan University, China)

Chapter 5

Doxygen

We use Doxygen as our automatically documentation generator which will make our future maintainance minimized. You can obtain the software (Windows, Linux and OS X) as well as its manual on the official website

<http://www.doxygen.org>

For an ordinary user, Doxygen is completely trivial to use. We only need to use some special marker in the usual comment as we put in c-files.

Chapter 6

Data Structure Index

6.1 Data Structures

Here are the data structures with brief descriptions:

AMG_data	Data for AMG methods	17
AMG_data_bsr	Data for multigrid levels in dBSRmat format	23
AMG_param	Parameters for AMG methods	30
block_dvector	Block REAL vector structure	42
block_ivector	Block INT vector structure	43
dBLCmat	Block REAL CSR matrix format	44
dBSRmat	Block sparse row storage matrix of REAL type	45
dCOOmat	Sparse matrix of REAL type in COO (IJ) format	48
dCSRLmat	Sparse matrix of REAL type in CSRL format	50
dCSRmat	Sparse matrix of REAL type in CSR format	53
ddenmat	Dense matrix of REAL type	56
dSTRmat	Structure matrix of REAL type	57
dvector	Vector with n entries of REAL type	60
grid2d	Two dimensional grid data structure	61
iBLCmat	Block INT CSR matrix format	64
iCOOmat	Sparse matrix of INT type in COO (IJ) format	66

iCSRmat	Sparse matrix of INT type in CSR format	68
idenmat	Dense matrix of INT type	70
ILU_data	Data for ILU setup	72
ILU_param	Parameters for ILU	77
input_param	Input parameters	79
ITS_param	Parameters for iterative solvers	92
ivector	Vector with n entries of INT type	94
Mumps_data	Data for MUMPS interface	95
mxv_matfree	Matrix-vector multiplication, replace the actual matrix	96
Pardiso_data	Data for Intel MKL PARDISO interface	97
precond	Preconditioner data and action	97
precond_data	Data for preconditioners	99
precond_data_blc	Data for block preconditioners in dBLCmat format	106
precond_data_bsr	Data for preconditioners in dBSRmat format	107
precond_data_str	Data for preconditioners in dSTRmat format	115
precond_data_sweeping	Data for sweeping preconditioner	121
precond_diag_bsr	Data for diagonal preconditioners in dBSRmat format	124
precond_diag_str	Data for diagonal preconditioners in dSTRmat format	125
SWZ_data	Data for Schwarz methods	126
SWZ_param	Parameters for Schwarz method	131

Chapter 7

File Index

7.1 File List

Here is a list of all documented files with brief descriptions:

doxygen.h	Main page for Doygen documentation	135
XtrMumps.c	Interface to MUMPS direct solvers	135
XtrPardiso.c	Interface to Intel MKL PARDISO direct solvers	143
XtrSamg.c	Interface to SAMG solvers	147
XtrSuperlu.c	Interface to SuperLU direct solvers	150
XtrUmfpack.c	Interface to UMFPACK direct solvers	153
fasp.h	Main header file for the FASP project	156
fasp_block.h	Header file for FASP block matrices	172
fasp_const.h	Definition of FASP constants, including messages, solver types, etc	177
fasp_grid.h	Header file for FASP grid	202
AuxArray.c	Simple array operations – init, set, copy, etc	203
AuxConvert.c	Utilities for encoding format conversion	209
AuxGivens.c	Givens transformation	211
AuxGraphics.c	Graphical output for CSR matrix	213
AuxInput.c	Read and check input parameters	223
AuxMemory.c	Memory allocation and deallocation subroutines	236

AuxMessage.c	
Output some useful messages	241
AuxParam.c	
Initialize, set, or print input data and parameters	248
AuxSort.c	
Array sorting/merging and removing duplicated integers	267
AuxThreads.c	
Get and set number of threads and assign work load for each thread	276
AuxTiming.c	
Timing subroutines	279
AuxVector.c	
Simple vector operations – init, set, copy, etc	281
BlaArray.c	
BLAS1 operations for arrays	291
BlaEigen.c	
Computing the extreme eigenvalues	307
BlaFormat.c	
Subroutines for matrix format conversion	309
BlaLU.c	
Incomplete LU decomposition: ILUK, ILUt, ILUtp	325
BlaLUSetupBSR.c	
Setup incomplete LU decomposition for dBSRmat matrices	353
BlaLUSetupCSR.c	
Setup incomplete LU decomposition for dCSRmat matrices	379
BlaLUSetupSTR.c	
Setup incomplete LU decomposition for dSTRmat matrices	382
BlaIO.c	
Matrix/vector input/output subroutines	399
BlaOrderingCSR.c	
Generating ordering using algebraic information	446
BlaSchwarzSetup.c	
Setup phase for the Schwarz methods	449
BlaSmallMat.c	
BLAS operations for <i>small</i> dense matrices	457
BlaSmallMatInv.c	
Find inversion of <i>small</i> dense matrices in row-major format	485
BlaSmallMatLU.c	
LU decomposition and direct solver for small dense matrices	501
BlaSparseBLC.c	
Sparse matrix block operations	505
BlaSparseBSR.c	
Sparse matrix operations for dBSRmat matrices	506
BlaSparseCheck.c	
Check properties of sparse matrices	542
BlaSparseCOO.c	
Sparse matrix operations for dCOOmat matrices	551
BlaSparseCSR.c	
Sparse matrix operations for dCSRmat matrices	554
BlaSparseCSRL.c	
Sparse matrix operations for dCSRLmat matrices	593
BlaSparseSTR.c	
Sparse matrix operations for dSTRmat matrices	595
BlaSparseUtil.c	
Routines for sparse matrix operations	599

BlaSpmvBLC.c	Linear algebraic operations for dBLCmat matrices	618
BlaSpmvBSR.c	Linear algebraic operations for dBSRmat matrices	624
BlaSpmvCSR.c	Linear algebraic operations for dCSRmat matrices	696
BlaSpmvCSRL.c	Linear algebraic operations for dCSRLmat matrices	730
BlaSpmvSTR.c	Linear algebraic operations for dSTRmat matrices	733
BlaVector.c	BLAS1 operations for vectors	754
ltrSmootherBSR.c	Smoothers for dBSRmat matrices	761
ltrSmootherCSR.c	Smoothers for dCSRmat matrices	789
ltrSmootherCSRcr.c	Smoothers for dCSRmat matrices using compatible relaxation	813
ltrSmootherCSRpoly.c	Smoothers for dCSRmat matrices using poly. approx. to A^{-1}	815
ltrSmootherSTR.c	Smoothers for dSTRmat matrices	823
KryPbcgs.c	Krylov subspace methods – Preconditioned BiCGstab	852
KryPcg.c	Krylov subspace methods – Preconditioned CG	876
KryPgcg.c	Krylov subspace methods – Preconditioned generalized CG	899
KryPgcr.c	Krylov subspace methods – Preconditioned GCR	904
KryPgmres.c	Krylov subspace methods – Right-preconditioned GMRes	911
KryPminres.c	Krylov subspace methods – Preconditioned minimal residual	935
KryPvfgmres.c	Krylov subspace methods – Preconditioned variable-restarting FGMRes	958
KryPvgmres.c	Krylov subspace methods – Preconditioned variable-restart GMRes	977
KrySPbcgs.c	Krylov subspace methods – Preconditioned BiCGstab with safety net	1002
KrySPcg.c	Krylov subspace methods – Preconditioned CG with safety net	1024
KrySPgmres.c	Krylov subspace methods – Preconditioned GMRes with safety net	1039
KrySPminres.c	Krylov subspace methods – Preconditioned MINRES with safety net	1059
KrySPvgmres.c	Krylov subspace methods – Preconditioned variable-restart GMRes with safety net	1079
PreAMGCoarsenCR.c	Coarsening with Brannick-Falgout strategy	1101
PreAMGCoarsenRS.c	Coarsening with a modified Ruge-Stuben strategy	1106
PreAMGInterp.c	Direct and standard interpolations for classical AMG	1130

PreAMGInterpEM.c	
Interpolation operators for AMG based on energy-min	1141
PreAMGSetupCR.c	
Brannick-Falgout compatible relaxation based AMG: SETUP phase	1151
PreAMGSetupRS.c	
Ruge-Stuben AMG: SETUP phase	1153
PreAMGSetupSA.c	
Smoothed aggregation AMG: SETUP phase	1158
PreAMGSetupSABSR.c	
Smoothed aggregation AMG: SETUP phase (for BSR matrices)	1168
PreAMGSetupUA.c	
Unsmoothed aggregation AMG: SETUP phase	1175
PreAMGSetupUABSR.c	
Unsmoothed aggregation AMG: SETUP phase (for BSR matrices)	1181
PreBLC.c	
Preconditioners for dBLCMat matrices	1186
PreBSR.c	
Preconditioners for dBSRmat matrices	1208
PreCSR.c	
Preconditioners for dCSRmat matrices	1227
PreDataInit.c	
Initialize important data structures	1239
PreMGCycle.c	
Abstract multigrid cycle – non-recursive version	1247
PreMGCycleFull.c	
Abstract non-recursive full multigrid cycle	1254
PreMGRecur.c	
Abstract multigrid cycle – recursive version	1260
PreMGRecurAMLI.c	
Abstract AMLI multilevel iteration – recursive version	1263
PreMGsolve.c	
Algebraic multigrid iterations: SOLVE phase	1274
PreSTR.c	
Preconditioners for dSTRmat matrices	1281
SolAMG.c	
AMG method as an iterative solver	1307
SolBLC.c	
Iterative solvers for dBLCMat matrices	1309
SolBSR.c	
Iterative solvers for dBSRmat matrices	1319
SolCSR.c	
Iterative solvers for dCSRmat matrices	1333
SolFAMG.c	
Full AMG method as an iterative solver	1348
SolGMGPoisson.c	
GMG method as an iterative solver for Poisson Problem	1351
SolMatFree.c	
Iterative solvers using MatFree spmv operations	1369
SolISTR.c	
Iterative solvers for dSTRmat matrices	1373
SolWrapper.c	
Wrappers for accessing functions by advanced users	1381

Chapter 8

Data Structure Documentation

8.1 AMG_data Struct Reference

Data for AMG methods.

```
#include <fasp.h>
```

Data Fields

- [SHORT max_levels](#)
max number of levels
- [SHORT num_levels](#)
number of levels in use \leq max_levels
- [dCSRmat A](#)
pointer to the matrix at level level_num
- [dCSRmat R](#)
restriction operator at level level_num
- [dCSRmat P](#)
prolongation operator at level level_num
- [dvector b](#)
pointer to the right-hand side at level level_num
- [dvector x](#)
pointer to the iterative solution at level level_num
- `void *` [Numeric](#)
pointer to the numerical factorization from UMFPACK
- [Pardiso_data pdata](#)
data for Intel MKL PARDISO
- [ivector cfmark](#)
pointer to the CF marker at level level_num
- [INT ILU_levels](#)
number of levels use ILU smoother

- [ILU_data LU](#)
ILU matrix for ILU smoother.
- [INT near_kernel_dim](#)
dimension of the near kernel for SAMG
- [REAL ** near_kernel_basis](#)
basis of near kernel space for SAMG
- [INT SWZ_levels](#)
number of levels use Schwarz smoother
- [SWZ_data Schwarz](#)
data of Schwarz smoother
- [dvector w](#)
temporary work space
- [Mumps_data mumps](#)
data for MUMPS
- [INT cycle_type](#)
cycle type
- [INT * ic](#)
indices for different colors
- [INT * icmap](#)
mapping from vertex to color
- [INT colors](#)
number of colors
- [REAL weight](#)
weight for smoother

8.1.1 Detailed Description

Data for AMG methods.

Note

This is needed for the AMG solver/preconditioner.

Definition at line [790](#) of file [fasp.h](#).

8.1.2 Field Documentation

8.1.2.1 A

[dCSRmat](#) A

pointer to the matrix at level `level_num`

Definition at line [803](#) of file [fasp.h](#).

8.1.2.2 b

`dvector` b

pointer to the right-hand side at level level_num

Definition at line 812 of file [fasp.h](#).

8.1.2.3 cfmark

`ivector` cfmark

pointer to the CF marker at level level_num

Definition at line 826 of file [fasp.h](#).

8.1.2.4 colors

`INT` colors

number of colors

Definition at line 864 of file [fasp.h](#).

8.1.2.5 cycle_type

`INT` cycle_type

cycle type

Definition at line 855 of file [fasp.h](#).

8.1.2.6 ic

`INT*` ic

indices for different colors

Definition at line 858 of file [fasp.h](#).

8.1.2.7 icmap

`INT* icmap`

mapping from vertex to color

Definition at line 861 of file [fasp.h](#).

8.1.2.8 ILU_levels

`INT ILU_levels`

number of levels use ILU smoother

Definition at line 829 of file [fasp.h](#).

8.1.2.9 LU

`ILU_data LU`

ILU matrix for ILU smoother.

Definition at line 832 of file [fasp.h](#).

8.1.2.10 max_levels

`SHORT max_levels`

max number of levels

Definition at line 795 of file [fasp.h](#).

8.1.2.11 mumps

`Mumps_data mumps`

data for MUMPS

Definition at line 852 of file [fasp.h](#).

8.1.2.12 near_kernel_basis

`REAL** near_kernel_basis`

basis of near kernel space for SAMG

Definition at line 838 of file [fasp.h](#).

8.1.2.13 near_kernel_dim

`INT near_kernel_dim`

dimension of the near kernel for SAMG

Definition at line 835 of file [fasp.h](#).

8.1.2.14 num_levels

`SHORT num_levels`

number of levels in use \leq max_levels

Definition at line 798 of file [fasp.h](#).

8.1.2.15 Numeric

`void* Numeric`

pointer to the numerical factorization from UMFPACK

Definition at line 820 of file [fasp.h](#).

8.1.2.16 P

`dCSRmat P`

prolongation operator at level level_num

Definition at line 809 of file [fasp.h](#).

8.1.2.17 pdata

`Pardiso_data` pdata

data for Intel MKL PARDISO

Definition at line 823 of file [fasp.h](#).

8.1.2.18 R

`dCSRmat` R

restriction operator at level level_num

Definition at line 806 of file [fasp.h](#).

8.1.2.19 Schwarz

`SWZ_data` Schwarz

data of Schwarz smoother

Definition at line 846 of file [fasp.h](#).

8.1.2.20 SWZ_levels

`INT` SWZ_levels

number of levels use Schwarz smoother

Definition at line 843 of file [fasp.h](#).

8.1.2.21 w

`dvector` w

temporary work space

Definition at line 849 of file [fasp.h](#).

8.1.2.22 weight

`REAL` weight

weight for smoother

Definition at line 867 of file [fasp.h](#).

8.1.2.23 x

`dvector` x

pointer to the iterative solution at level level_num

Definition at line 815 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.2 AMG_data_bsr Struct Reference

Data for multigrid levels in [dBSRmat](#) format.

```
#include <fasp_block.h>
```

Data Fields

- [INT](#) max_levels
max number of levels
- [INT](#) num_levels
number of levels in use <= max_levels
- [dBSRmat](#) A
pointer to the matrix at level level_num
- [dBSRmat](#) R
restriction operator at level level_num
- [dBSRmat](#) P
prolongation operator at level level_num
- [dvector](#) b
pointer to the right-hand side at level level_num
- [dvector](#) x
pointer to the iterative solution at level level_num
- [dvector](#) diaginv

- pointer to the diagonal inverse at level level_num*
- [dCSRmat Ac](#)
 - pointer to the matrix at level level_num (csr format)*
- `void * Numeric`
 - pointer to the numerical dactorization from UMFPACK*
- [Pardiso_data pdata](#)
 - data for Intel MKL PARDISO*
- [dCSRmat PP](#)
 - pointer to the pressure block (only for reservoir simulation)*
- `REAL * pw`
 - pointer to the auxiliary vectors for pressure block*
- [dBSRmat SS](#)
 - pointer to the saturation block (only for reservoir simulation)*
- `REAL * sw`
 - pointer to the auxiliary vectors for saturation block*
- [dvector diagin_v_SS](#)
 - pointer to the diagonal inverse of the saturation block at level level_num*
- [ILU_data PP_LU](#)
 - ILU data for pressure block.*
- [ivector cfmark](#)
 - pointer to the CF marker at level level_num*
- `INT ILU_levels`
 - number of levels use ILU smoother*
- [ILU_data LU](#)
 - ILU matrix for ILU smoother.*
- `INT near_kernel_dim`
 - dimension of the near kernel for SAMG*
- `REAL ** near_kernel_basis`
 - basis of near kernel space for SAMG*
- `dCSRmat * A_nk`
 - Matrix data for near kernal.*
- `dCSRmat * P_nk`
 - Prolongation for near kernal.*
- `dCSRmat * R_nk`
 - Resriction for near kernal.*
- [dvector w](#)
 - temporary work space*
- [Mumps_data mumps](#)
 - data for MUMPS*

8.2.1 Detailed Description

Data for multigrid levels in [dBSRmat](#) format.

Note

This structure is needed for the AMG solver/preconditioner in BSR format

Definition at line 146 of file [fasp_block.h](#).

8.2.2 Field Documentation

8.2.2.1 A

`dBSRmat` A

pointer to the matrix at level level_num

Definition at line 155 of file `fasp_block.h`.

8.2.2.2 A_nk

`dCSRmat*` A_nk

Matrix data for near kernal.

Definition at line 218 of file `fasp_block.h`.

8.2.2.3 Ac

`dCSRmat` Ac

pointer to the matrix at level level_num (csr format)

Definition at line 173 of file `fasp_block.h`.

8.2.2.4 b

`dvector` b

pointer to the right-hand side at level level_num

Definition at line 164 of file `fasp_block.h`.

8.2.2.5 cfmark

`ivector` cfmark

pointer to the CF marker at level level_num

Definition at line 200 of file [fasp_block.h](#).

8.2.2.6 diaginv

`dvector` diaginv

pointer to the diagonal inverse at level level_num

Definition at line 170 of file [fasp_block.h](#).

8.2.2.7 diaginv_SS

`dvector` diaginv_SS

pointer to the diagonal inverse of the saturation block at level level_num

Definition at line 194 of file [fasp_block.h](#).

8.2.2.8 ILU_levels

`INT` ILU_levels

number of levels use ILU smoother

Definition at line 203 of file [fasp_block.h](#).

8.2.2.9 LU

`ILU_data` LU

ILU matrix for ILU smoother.

Definition at line 206 of file [fasp_block.h](#).

8.2.2.10 max_levels

`INT max_levels`

max number of levels

Definition at line 149 of file [fasp_block.h](#).

8.2.2.11 mumps

`Mumps_data mumps`

data for MUMPS

Definition at line 231 of file [fasp_block.h](#).

8.2.2.12 near_kernel_basis

`REAL** near_kernel_basis`

basis of near kernel space for SAMG

Definition at line 212 of file [fasp_block.h](#).

8.2.2.13 near_kernel_dim

`INT near_kernel_dim`

dimension of the near kernel for SAMG

Definition at line 209 of file [fasp_block.h](#).

8.2.2.14 num_levels

`INT num_levels`

number of levels in use \leq max_levels

Definition at line 152 of file [fasp_block.h](#).

8.2.2.15 Numeric

`void* Numeric`

pointer to the numerical dactorization from UMFPACK

Definition at line 176 of file [fasp_block.h](#).

8.2.2.16 P

`dBSRmat P`

prolongation operator at level `level_num`

Definition at line 161 of file [fasp_block.h](#).

8.2.2.17 P_nk

`dCSRmat* P_nk`

Prolongation for near kernal.

Definition at line 221 of file [fasp_block.h](#).

8.2.2.18 pdata

`Pardiso_data pdata`

data for Intel MKL PARDISO

Definition at line 179 of file [fasp_block.h](#).

8.2.2.19 PP

`dCSRmat PP`

pointer to the pressure block (only for reservoir simulation)

Definition at line 182 of file [fasp_block.h](#).

8.2.2.20 PP_LU

`ILU_data` PP_LU

ILU data for pressure block.

Definition at line 197 of file `fasp_block.h`.

8.2.2.21 pw

`REAL*` pw

pointer to the auxiliary vectors for pressure block

Definition at line 185 of file `fasp_block.h`.

8.2.2.22 R

`dBSRmat` R

restriction operator at level level_num

Definition at line 158 of file `fasp_block.h`.

8.2.2.23 R_nk

`dCSRmat*` R_nk

Resriction for near kernal.

Definition at line 224 of file `fasp_block.h`.

8.2.2.24 SS

`dBSRmat` SS

pointer to the saturation block (only for reservoir simulation)

Definition at line 188 of file `fasp_block.h`.

8.2.2.25 **sw**

`REAL*` `sw`

pointer to the auxiliary vectors for saturation block

Definition at line 191 of file [fasp_block.h](#).

8.2.2.26 **w**

`dvector` `w`

temporary work space

Definition at line 228 of file [fasp_block.h](#).

8.2.2.27 **x**

`dvector` `x`

pointer to the iterative solution at level `level_num`

Definition at line 167 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.3 AMG_param Struct Reference

Parameters for AMG methods.

```
#include <fasp.h>
```

Data Fields

- [SHORT AMG_type](#)
type of AMG method
- [SHORT print_level](#)
print level for AMG
- [INT maxit](#)
max number of iterations of AMG
- [REAL tol](#)
stopping tolerance for AMG solver
- [SHORT max_levels](#)
max number of levels of AMG
- [INT coarse_dof](#)
max number of coarsest level DOF
- [SHORT cycle_type](#)
type of AMG cycle
- [REAL quality_bound](#)
quality threshold for pairwise aggregation
- [SHORT smoother](#)
smoother type
- [SHORT smooth_order](#)
smoother order
- [SHORT presmooth_iter](#)
number of presmoother
- [SHORT postsmooth_iter](#)
number of postsmoother
- [REAL relaxation](#)
relaxation parameter for Jacobi and SOR smoother
- [SHORT polynomial_degree](#)
degree of the polynomial smoother
- [SHORT coarse_solver](#)
coarse solver type
- [SHORT coarse_scaling](#)
switch of scaling of the coarse grid correction
- [SHORT aml_i_degree](#)
degree of the polynomial used by AMLI cycle
- [REAL * aml_i_coef](#)
coefficients of the polynomial used by AMLI cycle
- [SHORT nl_aml_i_krylov_type](#)
type of Krylov method used by Nonlinear AMLI cycle
- [SHORT coarsening_type](#)
coarsening type
- [SHORT aggregation_type](#)
aggregation type
- [SHORT interpolation_type](#)
interpolation type
- [REAL strong_threshold](#)

- strong connection threshold for coarsening*
- [REAL max_row_sum](#)
 - maximal row sum parameter*
- [REAL truncation_threshold](#)
 - truncation threshold*
- [INT aggressive_level](#)
 - number of levels use aggressive coarsening*
- [INT aggressive_path](#)
 - number of paths use to determine strongly coupled C points*
- [INT pair_number](#)
 - number of pairwise matchings*
- [REAL strong_coupled](#)
 - strong coupled threshold for aggregate*
- [INT max_aggregation](#)
 - max size of each aggregate*
- [REAL tentative_smooth](#)
 - relaxation parameter for smoothing the tentative prolongation*
- [SHORT smooth_filter](#)
 - switch for filtered matrix used for smoothing the tentative prolongation*
- [SHORT smooth_restriction](#)
 - smooth the restriction for SA methods or not*
- [SHORT ILU_levels](#)
 - number of levels use ILU smoother*
- [SHORT ILU_type](#)
 - ILU type for smoothing.*
- [INT ILU_ifil](#)
 - level of fill-in for ILUs and ILUk*
- [REAL ILU_droptol](#)
 - drop tolerance for ILU_t*
- [REAL ILU_relax](#)
 - relaxation for ILUs*
- [REAL ILU_permtol](#)
 - permuted if $\text{permtol} * |a(i,j)| > |a(i,i)|$*
- [INT SWZ_levels](#)
 - number of levels use Schwarz smoother*
- [INT SWZ_mmsize](#)
 - maximal block size*
- [INT SWZ_maxlvl](#)
 - maximal levels*
- [INT SWZ_type](#)
 - type of Schwarz method*
- [INT SWZ_blksolver](#)
 - type of Schwarz block solver*

8.3.1 Detailed Description

Parameters for AMG methods.

Note

This is needed for the AMG solver/preconditioner.

Definition at line 447 of file [fasp.h](#).

8.3.2 Field Documentation

8.3.2.1 aggregation_type

`SHORT aggregation_type`

aggregation type

Definition at line 510 of file [fasp.h](#).

8.3.2.2 aggressive_level

`INT aggressive_level`

number of levels use aggressive coarsening

Definition at line 525 of file [fasp.h](#).

8.3.2.3 aggressive_path

`INT aggressive_path`

number of paths use to determine strongly coupled C points

Definition at line 528 of file [fasp.h](#).

8.3.2.4 AMG_type

`SHORT` AMG_type

type of AMG method

Definition at line 450 of file [fasp.h](#).

8.3.2.5 amli_coef

`REAL*` amli_coef

coefficients of the polynomial used by AMLI cycle

Definition at line 501 of file [fasp.h](#).

8.3.2.6 amli_degree

`SHORT` amli_degree

degree of the polynomial used by AMLI cycle

Definition at line 498 of file [fasp.h](#).

8.3.2.7 coarse_dof

`INT` coarse_dof

max number of coarsest level DOF

Definition at line 465 of file [fasp.h](#).

8.3.2.8 coarse_scaling

`SHORT` coarse_scaling

switch of scaling of the coarse grid correction

Definition at line 495 of file [fasp.h](#).

8.3.2.9 coarse_solver

`SHORT coarse_solver`

coarse solver type

Definition at line 492 of file [fasp.h](#).

8.3.2.10 coarsening_type

`SHORT coarsening_type`

coarsening type

Definition at line 507 of file [fasp.h](#).

8.3.2.11 cycle_type

`SHORT cycle_type`

type of AMG cycle

Definition at line 468 of file [fasp.h](#).

8.3.2.12 ILU_droptol

`REAL ILU_droptol`

drop tolerance for ILU_t

Definition at line 558 of file [fasp.h](#).

8.3.2.13 ILU_levels

`SHORT ILU_levels`

number of levels use ILU smoother

Definition at line 549 of file [fasp.h](#).

8.3.2.14 ILU_lfil

`INT ILU_lfil`

level of fill-in for ILUs and ILUk

Definition at line 555 of file [fasp.h](#).

8.3.2.15 ILU_permtol

`REAL ILU_permtol`

permuted if $\text{permtol} * |a(i,j)| > |a(i,i)|$

Definition at line 564 of file [fasp.h](#).

8.3.2.16 ILU_relax

`REAL ILU_relax`

relaxation for ILUs

Definition at line 561 of file [fasp.h](#).

8.3.2.17 ILU_type

`SHORT ILU_type`

ILU type for smoothing.

Definition at line 552 of file [fasp.h](#).

8.3.2.18 interpolation_type

`SHORT interpolation_type`

interpolation type

Definition at line 513 of file [fasp.h](#).

8.3.2.19 max_aggregation

`INT max_aggregation`

max size of each aggregate

Definition at line 537 of file [fasp.h](#).

8.3.2.20 max_levels

`SHORT max_levels`

max number of levels of AMG

Definition at line 462 of file [fasp.h](#).

8.3.2.21 max_row_sum

`REAL max_row_sum`

maximal row sum parameter

Definition at line 519 of file [fasp.h](#).

8.3.2.22 maxit

`INT maxit`

max number of iterations of AMG

Definition at line 456 of file [fasp.h](#).

8.3.2.23 nl_amli_krylov_type

`SHORT nl_amli_krylov_type`

type of Krylov method used by Nonlinear AMLI cycle

Definition at line 504 of file [fasp.h](#).

8.3.2.24 `pair_number`

`INT pair_number`

number of pairwise matchings

Definition at line [531](#) of file [fasp.h](#).

8.3.2.25 `polynomial_degree`

`SHORT polynomial_degree`

degree of the polynomial smoother

Definition at line [489](#) of file [fasp.h](#).

8.3.2.26 `postsmooth_iter`

`SHORT postsmooth_iter`

number of postsmoothers

Definition at line [483](#) of file [fasp.h](#).

8.3.2.27 `presmooth_iter`

`SHORT presmooth_iter`

number of presmoothers

Definition at line [480](#) of file [fasp.h](#).

8.3.2.28 `print_level`

`SHORT print_level`

print level for AMG

Definition at line [453](#) of file [fasp.h](#).

8.3.2.29 quality_bound

`REAL` quality_bound

quality threshold for pairwise aggregation

Definition at line 471 of file [fasp.h](#).

8.3.2.30 relaxation

`REAL` relaxation

relaxation parameter for Jacobi and SOR smoother

Definition at line 486 of file [fasp.h](#).

8.3.2.31 smooth_filter

`SHORT` smooth_filter

switch for filtered matrix used for smoothing the tentative prolongation

Definition at line 543 of file [fasp.h](#).

8.3.2.32 smooth_order

`SHORT` smooth_order

smoother order

Definition at line 477 of file [fasp.h](#).

8.3.2.33 smooth_restriction

`SHORT` smooth_restriction

smooth the restriction for SA methods or not

Definition at line 546 of file [fasp.h](#).

8.3.2.34 smoother

`SHORT` smoother

smoother type

Definition at line 474 of file [fasp.h](#).

8.3.2.35 strong_coupled

`REAL` strong_coupled

strong coupled threshold for aggregate

Definition at line 534 of file [fasp.h](#).

8.3.2.36 strong_threshold

`REAL` strong_threshold

strong connection threshold for coarsening

Definition at line 516 of file [fasp.h](#).

8.3.2.37 SWZ_blksolver

`INT` SWZ_blksolver

type of Schwarz block solver

Definition at line 579 of file [fasp.h](#).

8.3.2.38 SWZ_levels

`INT` SWZ_levels

number of levels use Schwarz smoother

Definition at line 567 of file [fasp.h](#).

8.3.2.39 SWZ_maxlvl

`INT SWZ_maxlvl`

maximal levels

Definition at line 573 of file [fasp.h](#).

8.3.2.40 SWZ_mmsize

`INT SWZ_mmsize`

maximal block size

Definition at line 570 of file [fasp.h](#).

8.3.2.41 SWZ_type

`INT SWZ_type`

type of Schwarz method

Definition at line 576 of file [fasp.h](#).

8.3.2.42 tentative_smooth

`REAL tentative_smooth`

relaxation parameter for smoothing the tentative prolongation

Definition at line 540 of file [fasp.h](#).

8.3.2.43 tol

`REAL tol`

stopping tolerance for AMG solver

Definition at line 459 of file [fasp.h](#).

8.3.2.44 truncation_threshold

`REAL truncation_threshold`

truncation threshold

Definition at line 522 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.4 block_dvector Struct Reference

Block REAL vector structure.

```
#include <fasp_block.h>
```

Data Fields

- [INT brow](#)
row number of blocks in A, m
- [dvector ** blocks](#)
blocks of dvector, point to blocks[brow]

8.4.1 Detailed Description

Block REAL vector structure.

Definition at line 110 of file [fasp_block.h](#).

8.4.2 Field Documentation

8.4.2.1 blocks

`dvector** blocks`

blocks of dvector, point to blocks[brow]

Definition at line 116 of file [fasp_block.h](#).

8.4.2.2 brow

`INT brow`

row number of blocks in A, m

Definition at line 113 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.5 block_ivector Struct Reference

Block INT vector structure.

```
#include <fasp_block.h>
```

Data Fields

- `INT brow`
row number of blocks in A, m
- `ivector ** blocks`
blocks of dvector, point to blocks[brow]

8.5.1 Detailed Description

Block INT vector structure.

Note

The starting index of A is 0.

Definition at line 126 of file [fasp_block.h](#).

8.5.2 Field Documentation

8.5.2.1 blocks

`ivector** blocks`

blocks of dvector, point to blocks[brow]

Definition at line 132 of file [fasp_block.h](#).

8.5.2.2 brow

`INT brow`

row number of blocks in A, m

Definition at line 129 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.6 dBLCmat Struct Reference

Block REAL CSR matrix format.

```
#include <fasp_block.h>
```

Data Fields

- [INT brow](#)
row number of blocks in A, m
- [INT bcol](#)
column number of blocks A, n
- [dCSRmat** blocks](#)
blocks of [dCSRmat](#), point to blocks[brow][bcol]

8.6.1 Detailed Description

Block REAL CSR matrix format.

Note

The starting index of A is 0.

Definition at line 74 of file [fasp_block.h](#).

8.6.2 Field Documentation

8.6.2.1 bcol

`INT bcol`

column number of blocks A, n

Definition at line 80 of file [fasp_block.h](#).

8.6.2.2 blocks

`dCSRmat** blocks`

blocks of [dCSRmat](#), point to blocks[brow][bcol]

Definition at line 83 of file [fasp_block.h](#).

8.6.2.3 brow

`INT brow`

row number of blocks in A, m

Definition at line 77 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.7 dBSRmat Struct Reference

Block sparse row storage matrix of REAL type.

```
#include <fasp_block.h>
```

Data Fields

- `INT ROW`
number of rows of sub-blocks in matrix A, M
- `INT COL`
number of cols of sub-blocks in matrix A, N
- `INT NNZ`
number of nonzero sub-blocks in matrix A, NNZ
- `INT nb`
dimension of each sub-block
- `INT storage_manner`
storage manner for each sub-block
- `REAL * val`
- `INT * IA`
integer array of row pointers, the size is ROW+1
- `INT * JA`

8.7.1 Detailed Description

Block sparse row storage matrix of REAL type.

Note

This data structure is adapted from the Intel MKL library. Refer to: <http://software.intel.com/sites/products/documentation/hpc/mkl/lin/index.htm>

Some of the following entries are capitalized to stress that they are for blocks!

Definition at line 34 of file `fasp_block.h`.

8.7.2 Field Documentation

8.7.2.1 COL

`INT COL`

number of cols of sub-blocks in matrix A, N

Definition at line 40 of file `fasp_block.h`.

8.7.2.2 IA

`INT*` IA

integer array of row pointers, the size is ROW+1

Definition at line 60 of file [fasp_block.h](#).

8.7.2.3 JA

`INT*` JA

Element i of the integer array columns is the number of the column in the block matrix that contains the i-th non-zero block. The size is NNZ.

Definition at line 64 of file [fasp_block.h](#).

8.7.2.4 nb

`INT` nb

dimension of each sub-block

Definition at line 46 of file [fasp_block.h](#).

8.7.2.5 NNZ

`INT` NNZ

number of nonzero sub-blocks in matrix A, NNZ

Definition at line 43 of file [fasp_block.h](#).

8.7.2.6 ROW

`INT` ROW

number of rows of sub-blocks in matrix A, M

Definition at line 37 of file [fasp_block.h](#).

8.7.2.7 storage_manner

`INT storage_manner`

storage manner for each sub-block

Definition at line 49 of file [fasp_block.h](#).

8.7.2.8 val

`REAL* val`

A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each nonzero block elements are stored in row-major order and the size is (NNZ*nb*nb).

Definition at line 57 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.8 dCOOmat Struct Reference

Sparse matrix of REAL type in COO (IJ) format.

```
#include <fasp.h>
```

Data Fields

- `INT row`
row number of matrix A, m
- `INT col`
column of matrix A, n
- `INT nnz`
number of nonzero entries
- `INT * rowind`
integer array of row indices, the size is nnz
- `INT * colind`
integer array of column indices, the size is nnz
- `REAL * val`
nonzero entries of A

8.8.1 Detailed Description

Sparse matrix of REAL type in COO (IJ) format.

Coordinate Format (I,J,A)

Note

The starting index of A is 0.

Change I to rowind, J to colind. To avoid with complex.h confliction on I.

Definition at line 213 of file [fasp.h](#).

8.8.2 Field Documentation

8.8.2.1 col

`INT col`

column of matrix A, n

Definition at line 219 of file [fasp.h](#).

8.8.2.2 colind

`INT* colind`

integer array of column indices, the size is nnz

Definition at line 228 of file [fasp.h](#).

8.8.2.3 nnz

`INT nnz`

number of nonzero entries

Definition at line 222 of file [fasp.h](#).

8.8.2.4 row

`INT row`

row number of matrix A, m

Definition at line 216 of file [fasp.h](#).

8.8.2.5 rowind

`INT* rowind`

integer array of row indices, the size is nnz

Definition at line 225 of file [fasp.h](#).

8.8.2.6 val

`REAL* val`

nonzero entries of A

Definition at line 231 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.9 dCSRLmat Struct Reference

Sparse matrix of REAL type in CSRL format.

```
#include <fasp.h>
```


Data Fields

- [INT row](#)
number of rows
- [INT col](#)
number of cols
- [INT nnz](#)
number of nonzero entries
- [INT dif](#)
number of different values in i-th row, i=0:nrows-1
- [INT * nz_diff](#)
nz_diff[i]: the i-th different value in 'nzrow'
- [INT * index](#)
row index of the matrix (length-grouped): rows with same nnz are together
- [INT * start](#)
j in {start[i],...,start[i+1]-1} means nz_diff[i] nnz in index[j]-row
- [INT * ja](#)
column indices of all the nonzeros
- [REAL * val](#)
values of all the nonzero entries

8.9.1 Detailed Description

Sparse matrix of REAL type in CSRL format.

Definition at line 269 of file [fasp.h](#).

8.9.2 Field Documentation

8.9.2.1 col

[INT col](#)

number of cols

Definition at line 275 of file [fasp.h](#).

8.9.2.2 dif

`INT dif`

number of different values in i-th row, i=0:nrows-1

Definition at line 281 of file [fasp.h](#).

8.9.2.3 index

`INT* index`

row index of the matrix (length-grouped): rows with same nnz are together

Definition at line 287 of file [fasp.h](#).

8.9.2.4 ja

`INT* ja`

column indices of all the nonzeros

Definition at line 293 of file [fasp.h](#).

8.9.2.5 nnz

`INT nnz`

number of nonzero entries

Definition at line 278 of file [fasp.h](#).

8.9.2.6 nz_diff

`INT* nz_diff`

`nz_diff[i]`: the i-th different value in 'nzrow'

Definition at line 284 of file [fasp.h](#).

8.9.2.7 row

`INT row`

number of rows

Definition at line 272 of file [fasp.h](#).

8.9.2.8 start

`INT* start`

j in $\{start[i], \dots, start[i+1]-1\}$ means $nz_diff[i]$ nnz in $index[j]$ -row

Definition at line 290 of file [fasp.h](#).

8.9.2.9 val

`REAL* val`

values of all the nonzero entries

Definition at line 296 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.10 dCSRmat Struct Reference

Sparse matrix of REAL type in CSR format.

```
#include <fasp.h>
```

Data Fields

- `INT row`
row number of matrix A, m
- `INT col`
column of matrix A, n
- `INT nnz`
number of nonzero entries
- `INT * IA`
integer array of row pointers, the size is m+1
- `INT * JA`
integer array of column indexes, the size is nnz
- `REAL * val`
nonzero entries of A

8.10.1 Detailed Description

Sparse matrix of REAL type in CSR format.

CSR Format (IA,JA,A) in REAL

Note

The starting index of A is 0.

Definition at line 143 of file `fasp.h`.

8.10.2 Field Documentation

8.10.2.1 `col`

`INT col`

column of matrix A, n

Definition at line 149 of file `fasp.h`.

8.10.2.2 IA

`INT*` IA

integer array of row pointers, the size is m+1

Definition at line 155 of file [fasp.h](#).

8.10.2.3 JA

`INT*` JA

integer array of column indexes, the size is nnz

Definition at line 158 of file [fasp.h](#).

8.10.2.4 nnz

`INT` nnz

number of nonzero entries

Definition at line 152 of file [fasp.h](#).

8.10.2.5 row

`INT` row

row number of matrix A, m

Definition at line 146 of file [fasp.h](#).

8.10.2.6 val

`REAL*` val

nonzero entries of A

Definition at line 161 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.11 ddenmat Struct Reference

Dense matrix of REAL type.

```
#include <fasp.h>
```

Data Fields

- [INT row](#)
number of rows
- [INT col](#)
number of columns
- [REAL ** val](#)
actual matrix entries

8.11.1 Detailed Description

Dense matrix of REAL type.

A dense REAL matrix

Definition at line 103 of file [fasp.h](#).

8.11.2 Field Documentation

8.11.2.1 col

[INT col](#)

number of columns

Definition at line 109 of file [fasp.h](#).

8.11.2.2 row

[INT row](#)

number of rows

Definition at line 106 of file [fasp.h](#).

8.11.2.3 val

`REAL** val`

actual matrix entries

Definition at line 112 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.12 dSTRmat Struct Reference

Structure matrix of REAL type.

```
#include <fasp.h>
```

Data Fields

- `INT nx`
number of grids in x direction
- `INT ny`
number of grids in y direction
- `INT nz`
number of grids in z direction
- `INT nxy`
number of grids on x-y plane
- `INT nc`
size of each block (number of components)
- `INT ngrid`
number of grids
- `REAL * diag`
*diagonal entries (length is $ngrid * (nc^2)$)*
- `INT nband`
number of off-diag bands
- `INT * offsets`
offsets of the off-diagonals (length is nband)
- `REAL ** offdiag`
*off-diagonal entries (dimension is $nband * [(ngrid - |offsets|) * nc^2]$)*

8.12.1 Detailed Description

Structure matrix of REAL type.

Note

Every nc^2 entries of the array `diag` and `off-diag[i]` store one block: For 2D matrix, the recommended offsets is `[-1,1,-nx,nx]`; For 3D matrix, the recommended offsets is `[-1,1,-nx,nx,-nxy,nxy]`.

Definition at line 308 of file [fasp.h](#).

8.12.2 Field Documentation

8.12.2.1 `diag`

`REAL* diag`

diagonal entries (length is $ngrid*(nc^2)$)

Definition at line 329 of file [fasp.h](#).

8.12.2.2 `nband`

`INT nband`

number of off-diag bands

Definition at line 332 of file [fasp.h](#).

8.12.2.3 `nc`

`INT nc`

size of each block (number of components)

Definition at line 323 of file [fasp.h](#).

8.12.2.4 ngrid

`INT ngrid`

number of grids

Definition at line 326 of file [fasp.h](#).

8.12.2.5 nx

`INT nx`

number of grids in x direction

Definition at line 311 of file [fasp.h](#).

8.12.2.6 nxy

`INT nxy`

number of grids on x-y plane

Definition at line 320 of file [fasp.h](#).

8.12.2.7 ny

`INT ny`

number of grids in y direction

Definition at line 314 of file [fasp.h](#).

8.12.2.8 nz

`INT nz`

number of grids in z direction

Definition at line 317 of file [fasp.h](#).

8.12.2.9 offdiag

`REAL** offdiag`

off-diagonal entries (dimension is $\text{nband} * [(\text{ngrid} - |\text{offsets}|) * \text{nc}^2]$)

Definition at line 338 of file [fasp.h](#).

8.12.2.10 offsets

`INT* offsets`

offsets of the off-diagonals (length is nband)

Definition at line 335 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.13 dvector Struct Reference

Vector with n entries of REAL type.

```
#include <fasp.h>
```

Data Fields

- `INT row`
number of rows
- `REAL * val`
actual vector entries

8.13.1 Detailed Description

Vector with n entries of REAL type.

Definition at line 346 of file [fasp.h](#).

8.13.2 Field Documentation

8.13.2.1 row

`INT row`

number of rows

Definition at line 349 of file [fasp.h](#).

8.13.2.2 val

`REAL* val`

actual vector entries

Definition at line 352 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.14 grid2d Struct Reference

Two dimensional grid data structure.

```
#include <fasp_grid.h>
```

Data Fields

- `REAL(* p)[2]`
- `INT(* e)[2]`
- `INT(* t)[3]`
- `INT(* s)[3]`
- `INT * pdir`
- `INT * edir`
- `INT * pfather`
- `INT * efather`
- `INT * tfather`
- `INT vertices`
- `INT edges`
- `INT triangles`

8.14.1 Detailed Description

Two dimensional grid data structure.

Note

The [grid2d](#) structure is simply a list of triangles, edges and vertices. edge i has 2 vertices $e[i]$, triangle i has 3 edges $s[i]$, 3 vertices $t[i]$ vertex i has two coordinates $p[i]$

Definition at line [24](#) of file [fasp_grid.h](#).

8.14.2 Field Documentation

8.14.2.1 e

`INT (* e) [2]`

Vertices of edges

Definition at line [27](#) of file [fasp_grid.h](#).

8.14.2.2 edges

`INT edges`

Number of edges

Definition at line [38](#) of file [fasp_grid.h](#).

8.14.2.3 ediri

`INT* ediri`

Boundary flags (0 <=> interior edge)

Definition at line [31](#) of file [fasp_grid.h](#).

8.14.2.4 efather

`INT* efather`

Father edge or triangle

Definition at line 34 of file [fasp_grid.h](#).

8.14.2.5 p

`REAL (* p) [2]`

Coordinates of vertices

Definition at line 26 of file [fasp_grid.h](#).

8.14.2.6 pdiri

`INT* pdiri`

Boundary flags (0 <=> interior point)

Definition at line 30 of file [fasp_grid.h](#).

8.14.2.7 pfather

`INT* pfather`

Father point or edge

Definition at line 33 of file [fasp_grid.h](#).

8.14.2.8 s

`INT (* s) [3]`

Edges of triangles

Definition at line 29 of file [fasp_grid.h](#).

8.14.2.9 t

```
INT (* t) [3]
```

Vertices of triangles

Definition at line 28 of file [fasp_grid.h](#).

8.14.2.10 tfather

```
INT* tfather
```

Father triangle

Definition at line 35 of file [fasp_grid.h](#).

8.14.2.11 triangles

```
INT triangles
```

Number of triangles

Definition at line 39 of file [fasp_grid.h](#).

8.14.2.12 vertices

```
INT vertices
```

Number of grid points

Definition at line 37 of file [fasp_grid.h](#).

The documentation for this struct was generated from the following file:

- [fasp_grid.h](#)

8.15 iBLCmat Struct Reference

Block INT CSR matrix format.

```
#include <fasp_block.h>
```

Data Fields

- [INT brow](#)
row number of blocks in A, m
- [INT bcol](#)
column number of blocks A, n
- [iCSRmat](#) ** [blocks](#)
blocks of [iCSRmat](#), point to blocks[brow][bcol]

8.15.1 Detailed Description

Block INT CSR matrix format.

Note

The starting index of A is 0.

Definition at line 93 of file [fasp_block.h](#).

8.15.2 Field Documentation

8.15.2.1 bcol

[INT](#) bcol

column number of blocks A, n

Definition at line 99 of file [fasp_block.h](#).

8.15.2.2 blocks

[iCSRmat](#)** blocks

blocks of [iCSRmat](#), point to blocks[brow][bcol]

Definition at line 102 of file [fasp_block.h](#).

8.15.2.3 brow

`INT brow`

row number of blocks in A, m

Definition at line 96 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.16 iCOOmat Struct Reference

Sparse matrix of INT type in COO (IJ) format.

```
#include <fasp.h>
```

Data Fields

- [INT row](#)
row number of matrix A, m
- [INT col](#)
column of matrix A, n
- [INT nnz](#)
number of nonzero entries
- [INT * I](#)
integer array of row indices, the size is nnz
- [INT * J](#)
integer array of column indices, the size is nnz
- [INT * val](#)
nonzero entries of A

8.16.1 Detailed Description

Sparse matrix of INT type in COO (IJ) format.

Coordinate Format (I,J,A)

Note

The starting index of A is 0.

Definition at line 243 of file [fasp.h](#).

8.16.2 Field Documentation

8.16.2.1 col

`INT col`

column of matrix A, n

Definition at line 249 of file [fasp.h](#).

8.16.2.2 I

`INT* I`

integer array of row indices, the size is nnz

Definition at line 255 of file [fasp.h](#).

8.16.2.3 J

`INT* J`

integer array of column indices, the size is nnz

Definition at line 258 of file [fasp.h](#).

8.16.2.4 nnz

`INT nnz`

number of nonzero entries

Definition at line 252 of file [fasp.h](#).

8.16.2.5 row

`INT row`

row number of matrix A, m

Definition at line 246 of file [fasp.h](#).

8.16.2.6 val

`INT* val`

nonzero entries of A

Definition at line 261 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.17 iCSRmat Struct Reference

Sparse matrix of INT type in CSR format.

```
#include <fasp.h>
```

Data Fields

- [INT row](#)
row number of matrix A, m
- [INT col](#)
column of matrix A, n
- [INT nnz](#)
number of nonzero entries
- [INT * IA](#)
integer array of row pointers, the size is m+1
- [INT * JA](#)
integer array of column indexes, the size is nnz
- [INT * val](#)
nonzero entries of A

8.17.1 Detailed Description

Sparse matrix of INT type in CSR format.

CSR Format (IA,JA,A) in integer

Note

The starting index of A is 0.

Definition at line 182 of file [fasp.h](#).

8.17.2 Field Documentation

8.17.2.1 col

`INT col`

column of matrix A, n

Definition at line 188 of file [fasp.h](#).

8.17.2.2 IA

`INT* IA`

integer array of row pointers, the size is m+1

Definition at line 194 of file [fasp.h](#).

8.17.2.3 JA

`INT* JA`

integer array of column indexes, the size is nnz

Definition at line 197 of file [fasp.h](#).

8.17.2.4 nnz

`INT nnz`

number of nonzero entries

Definition at line 191 of file [fasp.h](#).

8.17.2.5 row

`INT row`

row number of matrix A, m

Definition at line 185 of file [fasp.h](#).

8.17.2.6 val

`INT* val`

nonzero entries of A

Definition at line 200 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.18 idenmat Struct Reference

Dense matrix of INT type.

```
#include <fasp.h>
```

Data Fields

- [INT row](#)
number of rows
- [INT col](#)
number of columns
- [INT ** val](#)
actual matrix entries

8.18.1 Detailed Description

Dense matrix of INT type.

A dense INT matrix

Definition at line 122 of file [fasp.h](#).

8.18.2 Field Documentation

8.18.2.1 col

`INT col`

number of columns

Definition at line 128 of file [fasp.h](#).

8.18.2.2 row

`INT row`

number of rows

Definition at line 125 of file [fasp.h](#).

8.18.2.3 val

`INT** val`

actual matrix entries

Definition at line 131 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.19 ILU_data Struct Reference

Data for ILU setup.

```
#include <fasp.h>
```

Data Fields

- [dCSRmat](#) * [A](#)
pointer to the original coefficient matrix
- [INT](#) [type](#)
type of ILUdata
- [INT](#) [row](#)
row number of matrix LU, m
- [INT](#) [col](#)
column of matrix LU, n
- [INT](#) [nzlu](#)
number of nonzero entries
- [INT](#) * [ijlu](#)
integer array of row pointers and column indexes, the size is nzlu
- [REAL](#) * [luval](#)
nonzero entries of LU
- [INT](#) [nb](#)
block size for BSR type only
- [INT](#) [nwork](#)
work space size
- [REAL](#) * [work](#)
work space
- [INT](#) * [iperm](#)
permutation arrays for ILUtp
- [INT](#) [ncolors](#)
number of colors for multi-threading
- [INT](#) * [ic](#)
indices for different colors
- [INT](#) * [icmap](#)
mapping from vertex to color
- [INT](#) * [uptr](#)
temporary work space
- [INT](#) [nlevL](#)
number of colors for lower triangle
- [INT](#) [nlevU](#)
number of colors for upper triangle
- [INT](#) * [ilevL](#)
number of vertices in each color for lower triangle
- [INT](#) * [ilevU](#)
number of vertices in each color for upper triangle
- [INT](#) * [jlevL](#)
mapping from row to color for lower triangle
- [INT](#) * [jlevU](#)
mapping from row to color for upper triangle

8.19.1 Detailed Description

Data for ILU setup.

Definition at line 637 of file [fasp.h](#).

8.19.2 Field Documentation

8.19.2.1 A

[dCSRmat*](#) A

pointer to the original coefficient matrix

Definition at line 640 of file [fasp.h](#).

8.19.2.2 col

[INT](#) col

column of matrix LU, n

Definition at line 649 of file [fasp.h](#).

8.19.2.3 ic

[INT*](#) ic

indices for different colors

Definition at line 678 of file [fasp.h](#).

8.19.2.4 icmap

[INT*](#) icmap

mapping from vertex to color

Definition at line 681 of file [fasp.h](#).

8.19.2.5 `ijlu`

`INT* ijlu`

integer array of row pointers and column indexes, the size is `nzlu`

Definition at line 655 of file [fasp.h](#).

8.19.2.6 `ilevL`

`INT* ilevL`

number of vertices in each color for lower triangle

Definition at line 693 of file [fasp.h](#).

8.19.2.7 `ilevU`

`INT* ilevU`

number of vertices in each color for upper triangle

Definition at line 696 of file [fasp.h](#).

8.19.2.8 `iperm`

`INT* iperm`

permutation arrays for ILUtp

Definition at line 670 of file [fasp.h](#).

8.19.2.9 `jlevL`

`INT* jlevL`

mapping from row to color for lower triangle

Definition at line 699 of file [fasp.h](#).

8.19.2.10 jlevU

`INT* jlevU`

mapping from row to color for upper triangle

Definition at line 702 of file [fasp.h](#).

8.19.2.11 luval

`REAL* luval`

nonzero entries of LU

Definition at line 658 of file [fasp.h](#).

8.19.2.12 nb

`INT nb`

block size for BSR type only

Definition at line 661 of file [fasp.h](#).

8.19.2.13 ncolors

`INT ncolors`

number of colors for multi-threading

Definition at line 675 of file [fasp.h](#).

8.19.2.14 nlevL

`INT nlevL`

number of colors for lower triangle

Definition at line 687 of file [fasp.h](#).

8.19.2.15 nlevU

`INT nlevU`

number of colors for upper triangle

Definition at line 690 of file [fasp.h](#).

8.19.2.16 nwork

`INT nwork`

work space size

Definition at line 664 of file [fasp.h](#).

8.19.2.17 nzlu

`INT nzlu`

number of nonzero entries

Definition at line 652 of file [fasp.h](#).

8.19.2.18 row

`INT row`

row number of matrix LU, m

Definition at line 646 of file [fasp.h](#).

8.19.2.19 type

`INT type`

type of ILUdata

Definition at line 643 of file [fasp.h](#).

8.19.2.20 uptr

`INT*` uptr

temporary work space

Definition at line 684 of file [fasp.h](#).

8.19.2.21 work

`REAL*` work

work space

Definition at line 667 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.20 ILU_param Struct Reference

Parameters for ILU.

```
#include <fasp.h>
```

Data Fields

- `SHORT` print_level
print level
- `SHORT` ILU_type
ILU type for decomposition.
- `INT` ILU_lfil
level of fill-in for ILUk
- `REAL` ILU_droptol
drop tolerance for ILUt
- `REAL` ILU_relax
add the sum of dropped elements to diagonal element in proportion relax
- `REAL` ILU_permtol
*permuted if $\text{permtol} * |a(i,j)| > |a(i,i)|$*

8.20.1 Detailed Description

Parameters for ILU.

Definition at line 396 of file [fasp.h](#).

8.20.2 Field Documentation

8.20.2.1 ILU_droptol

`REAL ILU_droptol`

drop tolerance for ILU_t

Definition at line 408 of file [fasp.h](#).

8.20.2.2 ILU_lfil

`INT ILU_lfil`

level of fill-in for ILU_k

Definition at line 405 of file [fasp.h](#).

8.20.2.3 ILU_permtol

`REAL ILU_permtol`

permuted if $\text{permtol} * |a(i,j)| > |a(i,i)|$

Definition at line 414 of file [fasp.h](#).

8.20.2.4 ILU_relax

`REAL ILU_relax`

add the sum of dropped elements to diagonal element in proportion relax

Definition at line 411 of file [fasp.h](#).

8.20.2.5 ILU_type

`SHORT` ILU_type

ILU type for decomposition.

Definition at line 402 of file [fasp.h](#).

8.20.2.6 print_level

`SHORT` print_level

print level

Definition at line 399 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.21 input_param Struct Reference

Input parameters.

```
#include <fasp.h>
```

Data Fields

- `SHORT` print_level
- `SHORT` output_type
- char inifile [STRLEN]
- char workdir [STRLEN]
- `INT` problem_num
- `SHORT` solver_type
- `SHORT` decoup_type
- `SHORT` precondition_type
- `SHORT` stop_type
- `REAL` itsolver_tol
- `INT` itsolver_maxit
- `INT` restart
- `SHORT` ILU_type
- `INT` ILU_lfil
- `REAL` ILU_droptol
- `REAL` ILU_relax

- [REAL ILU_permtol](#)
- [INT SWZ_mmsize](#)
- [INT SWZ_maxlvl](#)
- [INT SWZ_type](#)
- [INT SWZ_blksolver](#)
- [SHORT AMG_type](#)
- [SHORT AMG_levels](#)
- [SHORT AMG_cycle_type](#)
- [SHORT AMG_smoother](#)
- [SHORT AMG_smooth_order](#)
- [REAL AMG_relaxation](#)
- [SHORT AMG_polynomial_degree](#)
- [SHORT AMG_presmooth_iter](#)
- [SHORT AMG_postsmooth_iter](#)
- [REAL AMG_tol](#)
- [INT AMG_coarse_dof](#)
- [INT AMG_maxit](#)
- [SHORT AMG_ILU_levels](#)
- [SHORT AMG_coarse_solver](#)
- [SHORT AMG_coarse_scaling](#)
- [SHORT AMG_amli_degree](#)
- [SHORT AMG_nl_amli_krylov_type](#)
- [INT AMG_SWZ_levels](#)
- [SHORT AMG_coarsening_type](#)
- [SHORT AMG_aggregation_type](#)
- [SHORT AMG_interpolation_type](#)
- [REAL AMG_strong_threshold](#)
- [REAL AMG_truncation_threshold](#)
- [REAL AMG_max_row_sum](#)
- [INT AMG_aggressive_level](#)
- [INT AMG_aggressive_path](#)
- [INT AMG_pair_number](#)
- [REAL AMG_quality_bound](#)
- [REAL AMG_strong_coupled](#)
- [INT AMG_max_aggregation](#)
- [REAL AMG_tentative_smooth](#)
- [SHORT AMG_smooth_filter](#)
- [SHORT AMG_smooth_restriction](#)

8.21.1 Detailed Description

Input parameters.

Input parameters, reading from disk file

Definition at line 1111 of file [fasp.h](#).

8.21.2 Field Documentation

8.21.2.1 AMG_aggregation_type

`SHORT` AMG_aggregation_type

aggregation type

Definition at line 1166 of file [fasp.h](#).

8.21.2.2 AMG_aggressive_level

`INT` AMG_aggressive_level

number of levels use aggressive coarsening

Definition at line 1171 of file [fasp.h](#).

8.21.2.3 AMG_aggressive_path

`INT` AMG_aggressive_path

number of paths to determine strongly coupled C-set

Definition at line 1172 of file [fasp.h](#).

8.21.2.4 AMG_amli_degree

`SHORT` AMG_amli_degree

degree of the polynomial used by AMLI cycle

Definition at line 1160 of file [fasp.h](#).

8.21.2.5 AMG_coarse_dof

`INT` AMG_coarse_dof

max number of coarsest level DOF

Definition at line 1155 of file [fasp.h](#).

8.21.2.6 AMG_coarse_scaling

`SHORT` AMG_coarse_scaling

switch of scaling of the coarse grid correction

Definition at line 1159 of file [fasp.h](#).

8.21.2.7 AMG_coarse_solver

`SHORT` AMG_coarse_solver

coarse solver type

Definition at line 1158 of file [fasp.h](#).

8.21.2.8 AMG_coarsening_type

`SHORT` AMG_coarsening_type

coarsening type

Definition at line 1165 of file [fasp.h](#).

8.21.2.9 AMG_cycle_type

`SHORT` AMG_cycle_type

type of cycle

Definition at line 1147 of file [fasp.h](#).

8.21.2.10 AMG_ILU_levels

`SHORT` AMG_ILU_levels

how many levels use ILU smoother

Definition at line 1157 of file [fasp.h](#).

8.21.2.11 AMG_interpolation_type

`SHORT` AMG_interpolation_type

interpolation type

Definition at line 1167 of file [fasp.h](#).

8.21.2.12 AMG_levels

`SHORT` AMG_levels

maximal number of levels

Definition at line 1146 of file [fasp.h](#).

8.21.2.13 AMG_max_aggregation

`INT` AMG_max_aggregation

max size of each aggregate

Definition at line 1178 of file [fasp.h](#).

8.21.2.14 AMG_max_row_sum

`REAL` AMG_max_row_sum

maximal row sum

Definition at line 1170 of file [fasp.h](#).

8.21.2.15 AMG_maxit

`INT AMG_maxit`

number of iterations for AMG used as preconditioner

Definition at line 1156 of file [fasp.h](#).

8.21.2.16 AMG_nl_amli_krylov_type

`SHORT AMG_nl_amli_krylov_type`

type of Krylov method used by nonlinear AMLI cycle

Definition at line 1161 of file [fasp.h](#).

8.21.2.17 AMG_pair_number

`INT AMG_pair_number`

number of pairs in matching algorithm

Definition at line 1173 of file [fasp.h](#).

8.21.2.18 AMG_polynomial_degree

`SHORT AMG_polynomial_degree`

degree of the polynomial smoother

Definition at line 1151 of file [fasp.h](#).

8.21.2.19 AMG_postsmooth_iter

`SHORT AMG_postsmooth_iter`

number of postsmoothing

Definition at line 1153 of file [fasp.h](#).

8.21.2.20 AMG_presmooth_iter

`SHORT AMG_presmooth_iter`

number of presmoothing

Definition at line 1152 of file [fasp.h](#).

8.21.2.21 AMG_quality_bound

`REAL AMG_quality_bound`

threshold for pair wise aggregation

Definition at line 1174 of file [fasp.h](#).

8.21.2.22 AMG_relaxation

`REAL AMG_relaxation`

over-relaxation parameter for SOR

Definition at line 1150 of file [fasp.h](#).

8.21.2.23 AMG_smooth_filter

`SHORT AMG_smooth_filter`

use filter for smoothing the tentative prolongation or not

Definition at line 1180 of file [fasp.h](#).

8.21.2.24 AMG_smooth_order

`SHORT AMG_smooth_order`

order for smoothers

Definition at line 1149 of file [fasp.h](#).

8.21.2.25 AMG_smooth_restriction

`SHORT` AMG_smooth_restriction

smoothing the restriction or not

Definition at line 1181 of file [fasp.h](#).

8.21.2.26 AMG_smoother

`SHORT` AMG_smoother

type of smoother

Definition at line 1148 of file [fasp.h](#).

8.21.2.27 AMG_strong_coupled

`REAL` AMG_strong_coupled

strong coupled threshold for aggregate

Definition at line 1177 of file [fasp.h](#).

8.21.2.28 AMG_strong_threshold

`REAL` AMG_strong_threshold

strong threshold for coarsening

Definition at line 1168 of file [fasp.h](#).

8.21.2.29 AMG_SWZ_levels

`INT` AMG_SWZ_levels

number of levels use Schwarz smoother

Definition at line 1162 of file [fasp.h](#).

8.21.2.30 AMG_tentative_smooth

`REAL` AMG_tentative_smooth

relaxation factor for smoothing the tentative prolongation

Definition at line 1179 of file [fasp.h](#).

8.21.2.31 AMG_tol

`REAL` AMG_tol

tolerance for AMG if used as preconditioner

Definition at line 1154 of file [fasp.h](#).

8.21.2.32 AMG_truncation_threshold

`REAL` AMG_truncation_threshold

truncation factor for interpolation

Definition at line 1169 of file [fasp.h](#).

8.21.2.33 AMG_type

`SHORT` AMG_type

Type of AMG

Definition at line 1145 of file [fasp.h](#).

8.21.2.34 decoup_type

`SHORT` decoup_type

type of decoupling method for PDE systems

Definition at line 1124 of file [fasp.h](#).

8.21.2.35 ILU_droptol

`REAL ILU_droptol`

drop tolerance

Definition at line 1134 of file [fasp.h](#).

8.21.2.36 ILU_lfil

`INT ILU_lfil`

level of fill-in

Definition at line 1133 of file [fasp.h](#).

8.21.2.37 ILU_permtol

`REAL ILU_permtol`

permutation tolerance

Definition at line 1136 of file [fasp.h](#).

8.21.2.38 ILU_relax

`REAL ILU_relax`

scaling factor: add the sum of dropped entries to diagonal

Definition at line 1135 of file [fasp.h](#).

8.21.2.39 ILU_type

`SHORT ILU_type`

ILU type for decomposition

Definition at line 1132 of file [fasp.h](#).

8.21.2.40 inifile

`char inifile[STRLEN]`

ini file name

Definition at line 1118 of file [fasp.h](#).

8.21.2.41 itsolver_maxit

`INT itsolver_maxit`

maximal number of iterations for iterative solvers

Definition at line 1128 of file [fasp.h](#).

8.21.2.42 itsolver_tol

`REAL itsolver_tol`

tolerance for iterative linear solver

Definition at line 1127 of file [fasp.h](#).

8.21.2.43 output_type

`SHORT output_type`

type of output stream

Definition at line 1115 of file [fasp.h](#).

8.21.2.44 precondition_type

`SHORT precondition_type`

type of preconditioner for iterative solvers

Definition at line 1125 of file [fasp.h](#).

8.21.2.45 print_level

`SHORT print_level`

print level

Definition at line 1114 of file [fasp.h](#).

8.21.2.46 problem_num

`INT problem_num`

problem number to solve

Definition at line 1120 of file [fasp.h](#).

8.21.2.47 restart

`INT restart`

restart number used in GMRES

Definition at line 1129 of file [fasp.h](#).

8.21.2.48 solver_type

`SHORT solver_type`

type of iterative solvers

Definition at line 1123 of file [fasp.h](#).

8.21.2.49 stop_type

`SHORT stop_type`

type of stopping criteria for iterative solvers

Definition at line 1126 of file [fasp.h](#).

8.21.2.50 SWZ_blksolver

`INT SWZ_blksolver`

type of Schwarz block solver

Definition at line 1142 of file [fasp.h](#).

8.21.2.51 SWZ_maxlvl

`INT SWZ_maxlvl`

maximal levels

Definition at line 1140 of file [fasp.h](#).

8.21.2.52 SWZ_mmsize

`INT SWZ_mmsize`

maximal block size

Definition at line 1139 of file [fasp.h](#).

8.21.2.53 SWZ_type

`INT SWZ_type`

type of Schwarz method

Definition at line 1141 of file [fasp.h](#).

8.21.2.54 workdir

`char workdir[STRLEN]`

working directory for data files

Definition at line 1119 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.22 ITS_param Struct Reference

Parameters for iterative solvers.

```
#include <fasp.h>
```

Data Fields

- [SHORT print_level](#)
- [SHORT itsolver_type](#)
- [SHORT decoup_type](#)
- [SHORT precondition_type](#)
- [SHORT stop_type](#)
- [INT restart](#)
- [INT maxit](#)
- [REAL tol](#)

8.22.1 Detailed Description

Parameters for iterative solvers.

Definition at line [379](#) of file [fasp.h](#).

8.22.2 Field Documentation

8.22.2.1 decoup_type

[SHORT](#) decoup_type

decoupling type

Definition at line [383](#) of file [fasp.h](#).

8.22.2.2 itsolver_type

[SHORT](#) itsolver_type

solver type: see [fasp_const.h](#)

Definition at line [382](#) of file [fasp.h](#).

8.22.2.3 maxit

`INT maxit`

max number of iterations

Definition at line 387 of file [fasp.h](#).

8.22.2.4 precondition_type

`SHORT precondition_type`

preconditioner type

Definition at line 384 of file [fasp.h](#).

8.22.2.5 print_level

`SHORT print_level`

print level: 0–10

Definition at line 381 of file [fasp.h](#).

8.22.2.6 restart

`INT restart`

number of steps for restarting: for GMRES etc

Definition at line 386 of file [fasp.h](#).

8.22.2.7 stop_type

`SHORT stop_type`

stopping type

Definition at line 385 of file [fasp.h](#).

8.22.2.8 tol

`REAL tol`

convergence tolerance

Definition at line 388 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.23 ivector Struct Reference

Vector with n entries of INT type.

```
#include <fasp.h>
```

Data Fields

- [INT row](#)
number of rows
- [INT * val](#)
actual vector entries

8.23.1 Detailed Description

Vector with n entries of INT type.

Definition at line 361 of file [fasp.h](#).

8.23.2 Field Documentation

8.23.2.1 row

`INT row`

number of rows

Definition at line 364 of file [fasp.h](#).

8.23.2.2 val

`INT* val`

actual vector entries

Definition at line 367 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.24 Mumps_data Struct Reference

Data for MUMPS interface.

```
#include <fasp.h>
```

Data Fields

- [INT job](#)
work for MUMPS

8.24.1 Detailed Description

Data for MUMPS interface.

Added on 10/10/2014

Definition at line 593 of file [fasp.h](#).

8.24.2 Field Documentation

8.24.2.1 job

`INT job`

work for MUMPS

Definition at line 601 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.25 mxv_matfree Struct Reference

Matrix-vector multiplication, replace the actual matrix.

```
#include <fasp.h>
```

Data Fields

- void * [data](#)
data for MxV, can be a Matrix or something else
- void(* [fct](#))(const void *, const [REAL](#) *, [REAL](#) *)
action for MxV, void function pointer

8.25.1 Detailed Description

Matrix-vector multiplication, replace the actual matrix.

Definition at line [1095](#) of file [fasp.h](#).

8.25.2 Field Documentation

8.25.2.1 data

```
void* data
```

data for MxV, can be a Matrix or something else

Definition at line [1098](#) of file [fasp.h](#).

8.25.2.2 fct

```
void(* fct) (const void *, const REAL *, REAL *)
```

action for MxV, void function pointer

Definition at line [1101](#) of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.26 Pardiso_data Struct Reference

Data for Intel MKL PARDISO interface.

```
#include <fasp.h>
```

Data Fields

- void * [pt](#) [64]
Internal solver memory pointer.

8.26.1 Detailed Description

Data for Intel MKL PARDISO interface.

Added on 11/28/2015

Definition at line [611](#) of file [fasp.h](#).

8.26.2 Field Documentation

8.26.2.1 pt

```
void* pt[64]
```

Internal solver memory pointer.

Definition at line [614](#) of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.27 precondition Struct Reference

Preconditioner data and action.

```
#include <fasp.h>
```

Data Fields

- void * [data](#)
data for preconditioner, void pointer
- void(* [fct](#))([REAL](#) *, [REAL](#) *, void *)
action for preconditioner, void function pointer

8.27.1 Detailed Description

Preconditioner data and action.

Note

This is the preconditioner structure for preconditioned iterative methods.

Definition at line [1081](#) of file [fasp.h](#).

8.27.2 Field Documentation

8.27.2.1 data

```
void* data
```

data for preconditioner, void pointer

Definition at line [1084](#) of file [fasp.h](#).

8.27.2.2 fct

```
void(* fct) (REAL *, REAL *, void *)
```

action for preconditioner, void function pointer

Definition at line [1087](#) of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.28 precondition_data Struct Reference

Data for preconditioners.

```
#include <fasp.h>
```

Data Fields

- [SHORT AMG_type](#)
type of AMG method
- [SHORT print_level](#)
print level in AMG preconditioner
- [INT maxit](#)
max number of iterations of AMG preconditioner
- [SHORT max_levels](#)
max number of AMG levels
- [REAL tol](#)
tolerance for AMG preconditioner
- [SHORT cycle_type](#)
AMG cycle type.
- [SHORT smoother](#)
AMG smoother type.
- [SHORT smooth_order](#)
AMG smoother ordering.
- [SHORT presmooth_iter](#)
number of presmoothing
- [SHORT postsmooth_iter](#)
number of postsmoothing
- [REAL relaxation](#)
relaxation parameter for SOR smoother
- [SHORT polynomial_degree](#)
degree of the polynomial smoother
- [SHORT coarsening_type](#)
switch of scaling of the coarse grid correction
- [SHORT coarse_solver](#)
coarse solver type for AMG
- [SHORT coarse_scaling](#)
switch of scaling of the coarse grid correction
- [SHORT amli_degree](#)
degree of the polynomial used by AMLI cycle
- [SHORT nl_amli_krylov_type](#)
type of Krylov method used by Nonlinear AMLI cycle
- [REAL tentative_smooth](#)
smooth factor for smoothing the tentative prolongation
- [REAL * amli_coef](#)
coefficients of the polynomial used by AMLI cycle

- [AMG_data](#) * [mgl_data](#)
AMG preconditioner data.
- [ILU_data](#) * [LU](#)
ILU preconditioner data (needed for CPR type preconditioner)
- [dCSRmat](#) * [A](#)
Matrix data.
- [dCSRmat](#) * [A_nk](#)
Matrix data for near kernel.
- [dCSRmat](#) * [P_nk](#)
Prolongation for near kernel.
- [dCSRmat](#) * [R_nk](#)
Restriction for near kernel.
- [dvector](#) [r](#)
temporary dvector used to store and restore the residual
- [REAL](#) * [w](#)
temporary work space for other usage

8.28.1 Detailed Description

Data for preconditioners.

Definition at line [880](#) of file [fasp.h](#).

8.28.2 Field Documentation

8.28.2.1 A

[dCSRmat](#)* [A](#)

Matrix data.

Definition at line [946](#) of file [fasp.h](#).

8.28.2.2 A_nk

[dCSRmat](#)* [A_nk](#)

Matrix data for near kernel.

Definition at line [951](#) of file [fasp.h](#).

8.28.2.3 AMG_type

`SHORT` AMG_type

type of AMG method

Definition at line 883 of file [fasp.h](#).

8.28.2.4 amli_coef

`REAL*` amli_coef

coefficients of the polynomial used by AMLI cycle

Definition at line 937 of file [fasp.h](#).

8.28.2.5 amli_degree

`SHORT` amli_degree

degree of the polynomial used by AMLI cycle

Definition at line 928 of file [fasp.h](#).

8.28.2.6 coarse_scaling

`SHORT` coarse_scaling

switch of scaling of the coarse grid correction

Definition at line 925 of file [fasp.h](#).

8.28.2.7 coarse_solver

`SHORT` coarse_solver

coarse solver type for AMG

Definition at line 922 of file [fasp.h](#).

8.28.2.8 coarsening_type

`SHORT coarsening_type`

switch of scaling of the coarse grid correction

Definition at line 919 of file [fasp.h](#).

8.28.2.9 cycle_type

`SHORT cycle_type`

AMG cycle type.

Definition at line 898 of file [fasp.h](#).

8.28.2.10 LU

`ILU_data* LU`

ILU preconditioner data (needed for CPR type preconditioner)

Definition at line 943 of file [fasp.h](#).

8.28.2.11 max_levels

`SHORT max_levels`

max number of AMG levels

Definition at line 892 of file [fasp.h](#).

8.28.2.12 maxit

`INT maxit`

max number of iterations of AMG preconditioner

Definition at line 889 of file [fasp.h](#).

8.28.2.13 mgl_data

`AMG_data*` mgl_data

AMG preconditioner data.

Definition at line 940 of file [fasp.h](#).

8.28.2.14 nl_amli_krylov_type

`SHORT` nl_amli_krylov_type

type of Krylov method used by Nonlinear AMLI cycle

Definition at line 931 of file [fasp.h](#).

8.28.2.15 P_nk

`dCSRmat*` P_nk

Prolongation for near kernel.

Definition at line 954 of file [fasp.h](#).

8.28.2.16 polynomial_degree

`SHORT` polynomial_degree

degree of the polynomial smoother

Definition at line 916 of file [fasp.h](#).

8.28.2.17 postsmooth_iter

`SHORT` postsmooth_iter

number of postsmoothing

Definition at line 910 of file [fasp.h](#).

8.28.2.18 presmooth_iter

`SHORT presmooth_iter`

number of presmoothing

Definition at line 907 of file [fasp.h](#).

8.28.2.19 print_level

`SHORT print_level`

print level in AMG preconditioner

Definition at line 886 of file [fasp.h](#).

8.28.2.20 r

`dvector r`

temporary dvector used to store and restore the residual

Definition at line 962 of file [fasp.h](#).

8.28.2.21 R_nk

`dCSRmat* R_nk`

Restriction for near kernel.

Definition at line 957 of file [fasp.h](#).

8.28.2.22 relaxation

`REAL relaxation`

relaxation parameter for SOR smoother

Definition at line 913 of file [fasp.h](#).

8.28.2.23 smooth_order

`SHORT` smooth_order

AMG smoother ordering.

Definition at line 904 of file [fasp.h](#).

8.28.2.24 smoother

`SHORT` smoother

AMG smoother type.

Definition at line 901 of file [fasp.h](#).

8.28.2.25 tentative_smooth

`REAL` tentative_smooth

smooth factor for smoothing the tentative prolongation

Definition at line 934 of file [fasp.h](#).

8.28.2.26 tol

`REAL` tol

tolerance for AMG preconditioner

Definition at line 895 of file [fasp.h](#).

8.28.2.27 w

`REAL*` w

temporary work space for other usage

Definition at line 965 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.29 precondition_data_blc Struct Reference

Data for block preconditioners in [dBLMat](#) format.

```
#include <fasp_block.h>
```

Data Fields

- [dBLMat](#) * [Ablc](#)
- [dCSRmat](#) * [A_diag](#)
- [dvector](#) [r](#)
- void ** [LU_diag](#)
- [AMG_data](#) ** [mgl](#)
- [AMG_param](#) * [amgparam](#)

8.29.1 Detailed Description

Data for block preconditioners in [dBLMat](#) format.

This is needed for the block preconditioner.

Definition at line [349](#) of file [fasp_block.h](#).

8.29.2 Field Documentation

8.29.2.1 [A_diag](#)

[dCSRmat](#)* [A_diag](#)

data for each diagonal block

Definition at line [356](#) of file [fasp_block.h](#).

8.29.2.2 [Ablc](#)

[dBLMat](#)* [Ablc](#)

problem data, the blocks

Definition at line [354](#) of file [fasp_block.h](#).

8.29.2.3 amgparam

`AMG_param*` amgparam

parameters for AMG

Definition at line 370 of file [fasp_block.h](#).

8.29.2.4 LU_diag

`void**` LU_diag

LU decomposition for the diagonal blocks (for UMFpack)

Definition at line 365 of file [fasp_block.h](#).

8.29.2.5 mgl

`AMG_data**` mgl

AMG data for the diagonal blocks

Definition at line 368 of file [fasp_block.h](#).

8.29.2.6 r

`dvector` r

temp work space

Definition at line 358 of file [fasp_block.h](#).

The documentation for this struct was generated from the following file:

- [fasp_block.h](#)

8.30 precondition_data_bsr Struct Reference

Data for preconditioners in [dBSRmat](#) format.

```
#include <fasp_block.h>
```

Data Fields

- [SHORT AMG_type](#)
type of AMG method
- [SHORT print_level](#)
print level in AMG preconditioner
- [INT maxit](#)
max number of iterations of AMG preconditioner
- [INT max_levels](#)
max number of AMG levels
- [REAL tol](#)
tolerance for AMG preconditioner
- [SHORT cycle_type](#)
AMG cycle type.
- [SHORT smoother](#)
AMG smoother type.
- [SHORT smooth_order](#)
AMG smoother ordering.
- [SHORT presmooth_iter](#)
number of presmoothing
- [SHORT postsmooth_iter](#)
number of postsmoothing
- [SHORT coarsening_type](#)
coarsening type
- [REAL relaxation](#)
relaxation parameter for SOR smoother
- [SHORT coarse_solver](#)
coarse solver type for AMG
- [SHORT coarse_scaling](#)
switch of scaling of the coarse grid correction
- [SHORT aml_i_degree](#)
degree of the polynomial used by AMLI cycle
- [REAL * aml_i_coef](#)
coefficients of the polynomial used by AMLI cycle
- [REAL tentative_smooth](#)
smooth factor for smoothing the tentative prolongation
- [SHORT nl_aml_i_krylov_type](#)
type of krylov method used by Nonlinear AMLI cycle
- [AMG_data_bsr * mgl_data](#)
AMG preconditioner data.
- [AMG_data * pres_mgl_data](#)
AMG preconditioner data for pressure block.
- [ILU_data * LU](#)
ILU preconditioner data (needed for CPR type preconditioner)
- [dBSRmat * A](#)
Matrix data.
- [dCSRmat * A_nk](#)

- Matrix data for near kernal.*
- `dCSRmat * P_nk`
- Prolongation for near kernal.*
- `dCSRmat * R_nk`
- Resriction for near kernal.*
- `dvector r`
- temporary dvector used to store and restore the residual*
- `REAL * w`
- temporary work space for other usage*

8.30.1 Detailed Description

Data for preconditioners in `dBSRmat` format.

Note

This structure is needed for the AMG solver/preconditioner in BSR format

Definition at line 257 of file `fasp_block.h`.

8.30.2 Field Documentation

8.30.2.1 A

`dBSRmat * A`

Matrix data.

Definition at line 323 of file `fasp_block.h`.

8.30.2.2 A_nk

`dCSRmat * A_nk`

Matrix data for near kernal.

Definition at line 328 of file `fasp_block.h`.

8.30.2.3 AMG_type

`SHORT` AMG_type

type of AMG method

Definition at line 260 of file [fasp_block.h](#).

8.30.2.4 amli_coef

`REAL*` amli_coef

coefficients of the polynomial used by AMLI cycle

Definition at line 305 of file [fasp_block.h](#).

8.30.2.5 amli_degree

`SHORT` amli_degree

degree of the polynomial used by AMLI cycle

Definition at line 302 of file [fasp_block.h](#).

8.30.2.6 coarse_scaling

`SHORT` coarse_scaling

switch of scaling of the coarse grid correction

Definition at line 299 of file [fasp_block.h](#).

8.30.2.7 coarse_solver

`SHORT` coarse_solver

coarse solver type for AMG

Definition at line 296 of file [fasp_block.h](#).

8.30.2.8 coarsening_type

`SHORT coarsening_type`

coarsening type

Definition at line 290 of file [fasp_block.h](#).

8.30.2.9 cycle_type

`SHORT cycle_type`

AMG cycle type.

Definition at line 275 of file [fasp_block.h](#).

8.30.2.10 LU

`ILU_data* LU`

ILU preconditioner data (needed for CPR type preconditioner)

Definition at line 320 of file [fasp_block.h](#).

8.30.2.11 max_levels

`INT max_levels`

max number of AMG levels

Definition at line 269 of file [fasp_block.h](#).

8.30.2.12 maxit

`INT maxit`

max number of iterations of AMG preconditioner

Definition at line 266 of file [fasp_block.h](#).

8.30.2.13 mgl_data

`AMG_data_bsr*` mgl_data

AMG preconditioner data.

Definition at line 314 of file [fasp_block.h](#).

8.30.2.14 nl_amli_krylov_type

`SHORT` nl_amli_krylov_type

type of krylov method used by Nonlinear AMLI cycle

Definition at line 311 of file [fasp_block.h](#).

8.30.2.15 P_nk

`dCSRmat*` P_nk

Prolongation for near kernal.

Definition at line 331 of file [fasp_block.h](#).

8.30.2.16 postsmooth_iter

`SHORT` postsmooth_iter

number of postsmoothing

Definition at line 287 of file [fasp_block.h](#).

8.30.2.17 pres_mgl_data

`AMG_data*` pres_mgl_data

AMG preconditioner data for pressure block.

Definition at line 317 of file [fasp_block.h](#).

8.30.2.18 presmooth_iter

`SHORT presmooth_iter`

number of presmoothing

Definition at line 284 of file [fasp_block.h](#).

8.30.2.19 print_level

`SHORT print_level`

print level in AMG preconditioner

Definition at line 263 of file [fasp_block.h](#).

8.30.2.20 r

`dvector r`

temporary dvector used to store and restore the residual

Definition at line 337 of file [fasp_block.h](#).

8.30.2.21 R_nk

`dCSRmat* R_nk`

Restriction for near kernel.

Definition at line 334 of file [fasp_block.h](#).

8.30.2.22 relaxation

`REAL relaxation`

relaxation parameter for SOR smoother

Definition at line 293 of file [fasp_block.h](#).

8.30.2.23 `smooth_order`

`SHORT` `smooth_order`

AMG smoother ordering.

Definition at line 281 of file `fasp_block.h`.

8.30.2.24 `smoother`

`SHORT` `smoother`

AMG smoother type.

Definition at line 278 of file `fasp_block.h`.

8.30.2.25 `tentative_smooth`

`REAL` `tentative_smooth`

smooth factor for smoothing the tentative prolongation

Definition at line 308 of file `fasp_block.h`.

8.30.2.26 `tol`

`REAL` `tol`

tolerance for AMG preconditioner

Definition at line 272 of file `fasp_block.h`.

8.30.2.27 `w`

`REAL*` `w`

temporary work space for other usage

Definition at line 340 of file `fasp_block.h`.

The documentation for this struct was generated from the following file:

- `fasp_block.h`

8.31 precondition_data_str Struct Reference

Data for preconditioners in dSTRmat format.

```
#include <fasp.h>
```

Data Fields

- [SHORT AMG_type](#)
type of AMG method
- [SHORT print_level](#)
print level in AMG preconditioner
- [INT maxit](#)
max number of iterations of AMG preconditioner
- [SHORT max_levels](#)
max number of AMG levels
- [REAL tol](#)
tolerance for AMG preconditioner
- [SHORT cycle_type](#)
AMG cycle type.
- [SHORT smoother](#)
AMG smoother type.
- [SHORT presmooth_iter](#)
number of presmoothing
- [SHORT postsmooth_iter](#)
number of postsmoothing
- [SHORT coarsening_type](#)
coarsening type
- [REAL relaxation](#)
relaxation parameter for SOR smoother
- [SHORT coarse_scaling](#)
switch of scaling of the coarse grid correction
- [AMG_data](#) * [mgl_data](#)
AMG preconditioner data.
- [ILU_data](#) * [LU](#)
ILU preconditioner data (needed for CPR type preconditioner)
- [SHORT scaled](#)
whether the matrix are scaled or not
- [dCSRmat](#) * [A](#)
the original CSR matrix
- [dSTRmat](#) * [A_str](#)
store the whole reservoir block in STR format
- [dSTRmat](#) * [SS_str](#)
store Saturation block in STR format
- [dvector](#) * [diaginv](#)
the inverse of the diagonals for GS/block GS smoother (whole reservoir matrix)

- `ivector * pivot`
the pivot for the GS/block GS smoother (whole reservoir matrix)
- `dvector * diaginvS`
the inverse of the diagonals for GS/block GS smoother (saturation block)
- `ivector * pivotS`
the pivot for the GS/block GS smoother (saturation block)
- `ivector * order`
order for smoothing
- `ivector * neigh`
array to store neighbor information
- `dvector r`
temporary dvector used to store and restore the residual
- `REAL * w`
temporary work space for other usage

8.31.1 Detailed Description

Data for preconditioners in `dSTRmat` format.

Definition at line 973 of file `fasp.h`.

8.31.2 Field Documentation

8.31.2.1 A

`dCSRmat* A`

the original CSR matrix

Definition at line 1021 of file `fasp.h`.

8.31.2.2 A_str

`dSTRmat* A_str`

store the whole reservoir block in STR format

Definition at line 1024 of file `fasp.h`.

8.31.2.3 AMG_type

`SHORT` AMG_type

type of AMG method

Definition at line 976 of file [fasp.h](#).

8.31.2.4 coarse_scaling

`SHORT` coarse_scaling

switch of scaling of the coarse grid correction

Definition at line 1009 of file [fasp.h](#).

8.31.2.5 coarsening_type

`SHORT` coarsening_type

coarsening type

Definition at line 1003 of file [fasp.h](#).

8.31.2.6 cycle_type

`SHORT` cycle_type

AMG cycle type.

Definition at line 991 of file [fasp.h](#).

8.31.2.7 diaginv

`dvector*` diaginv

the inverse of the diagonals for GS/block GS smoother (whole reservoir matrix)

Definition at line 1032 of file [fasp.h](#).

8.31.2.8 `diaginvS`

`dvector*` `diaginvS`

the inverse of the diagonals for GS/block GS smoother (saturation block)

Definition at line 1038 of file [fasp.h](#).

8.31.2.9 `LU`

`ILU_data*` `LU`

ILU preconditioner data (needed for CPR type preconditioner)

Definition at line 1015 of file [fasp.h](#).

8.31.2.10 `max_levels`

`SHORT` `max_levels`

max number of AMG levels

Definition at line 985 of file [fasp.h](#).

8.31.2.11 `maxit`

`INT` `maxit`

max number of iterations of AMG preconditioner

Definition at line 982 of file [fasp.h](#).

8.31.2.12 `mgl_data`

`AMG_data*` `mgl_data`

AMG preconditioner data.

Definition at line 1012 of file [fasp.h](#).

8.31.2.13 neigh

`ivector*` neigh

array to store neighbor information

Definition at line 1047 of file [fasp.h](#).

8.31.2.14 order

`ivector*` order

order for smoothing

Definition at line 1044 of file [fasp.h](#).

8.31.2.15 pivot

`ivector*` pivot

the pivot for the GS/block GS smoother (whole reservoir matrix)

Definition at line 1035 of file [fasp.h](#).

8.31.2.16 pivotS

`ivector*` pivotS

the pivot for the GS/block GS smoother (saturation block)

Definition at line 1041 of file [fasp.h](#).

8.31.2.17 postsmooth_iter

`SHORT` postsmooth_iter

number of postsmoothing

Definition at line 1000 of file [fasp.h](#).

8.31.2.18 presmooth_iter

`SHORT presmooth_iter`

number of presmoothing

Definition at line 997 of file [fasp.h](#).

8.31.2.19 print_level

`SHORT print_level`

print level in AMG preconditioner

Definition at line 979 of file [fasp.h](#).

8.31.2.20 r

`dvector r`

temporary dvector used to store and restore the residual

Definition at line 1052 of file [fasp.h](#).

8.31.2.21 relaxation

`REAL relaxation`

relaxation parameter for SOR smoother

Definition at line 1006 of file [fasp.h](#).

8.31.2.22 scaled

`SHORT scaled`

whether the matrix are scaled or not

Definition at line 1018 of file [fasp.h](#).

8.31.2.23 smoother

`SHORT smoother`

AMG smoother type.

Definition at line 994 of file [fasp.h](#).

8.31.2.24 SS_str

`dSTRmat* SS_str`

store Saturation block in STR format

Definition at line 1027 of file [fasp.h](#).

8.31.2.25 tol

`REAL tol`

tolerance for AMG preconditioner

Definition at line 988 of file [fasp.h](#).

8.31.2.26 w

`REAL* w`

temporary work space for other usage

Definition at line 1055 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.32 precondition_data_sweeping Struct Reference

Data for sweeping preconditioner.

```
#include <fasp_block.h>
```

Data Fields

- [INT NumLayers](#)
- [dBLMat * A](#)
- [dBLMat * Ai](#)
- [dCSRmat * local_A](#)
- [void ** local_LU](#)
- [ivector * local_index](#)
- [dvector r](#)
- [REAL * w](#)

8.32.1 Detailed Description

Data for sweeping preconditioner.

Author

Xiaozhe Hu

Date

05/01/2014

Note

This is needed for the sweeping preconditioner.

Definition at line [384](#) of file [fasp_block.h](#).

8.32.2 Field Documentation

8.32.2.1 A

[dBLMat](#) * A

problem data, the sparse matrix

Definition at line [388](#) of file [fasp_block.h](#).

8.32.2.2 Ai

`dBLMat*` Ai

preconditioner data, the sparse matrix

Definition at line 389 of file [fasp_block.h](#).

8.32.2.3 local_A

`dCSRmat*` local_A

local stiffness matrix for each layer

Definition at line 391 of file [fasp_block.h](#).

8.32.2.4 local_index

`ivector*` local_index

local index for each layer

Definition at line 394 of file [fasp_block.h](#).

8.32.2.5 local_LU

`void**` local_LU

local LU decomposition (for UMFPack)

Definition at line 392 of file [fasp_block.h](#).

8.32.2.6 NumLayers

`INT` NumLayers

number of layers

Definition at line 386 of file [fasp_block.h](#).

8.32.2.7 **r**`dvector` **r**

temporary dvector used to store and restore the residual

Definition at line 397 of file `fasp_block.h`.

8.32.2.8 **w**`REAL*` **w**

temporary work space for other usage

Definition at line 398 of file `fasp_block.h`.

The documentation for this struct was generated from the following file:

- `fasp_block.h`

8.33 **precond_diag_bsr Struct Reference**

Data for diagnol preconditioners in `dbSRmat` format.

```
#include <fasp_block.h>
```

Data Fields

- `INT` **nb**
dimension of each sub-block
- `dvector` **diag**
diagnol elements

8.33.1 **Detailed Description**

Data for diagnol preconditioners in `dbSRmat` format.

Note

This is needed for the diagnol preconditioner.

Definition at line 241 of file `fasp_block.h`.

8.33.2 Field Documentation

8.33.2.1 diag

`dvector` diag

diagonal elements

Definition at line 247 of file `fasp_block.h`.

8.33.2.2 nb

`INT` nb

dimension of each sub-block

Definition at line 244 of file `fasp_block.h`.

The documentation for this struct was generated from the following file:

- `fasp_block.h`

8.34 precondition_diag_str Struct Reference

Data for diagonal preconditioners in `dSTRmat` format.

```
#include <fasp.h>
```

Data Fields

- `INT` *nc*
number of components
- `dvector` *diag*
diagonal elements

8.34.1 Detailed Description

Data for diagonal preconditioners in [dSTRmat](#) format.

Note

This is needed for the diagonal preconditioner.

Definition at line [1065](#) of file [fasp.h](#).

8.34.2 Field Documentation

8.34.2.1 diag

[dvector](#) diag

diagonal elements

Definition at line [1071](#) of file [fasp.h](#).

8.34.2.2 nc

[INT](#) nc

number of components

Definition at line [1068](#) of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.35 SWZ_data Struct Reference

Data for Schwarz methods.

```
#include <fasp.h>
```

Data Fields

- [dCSRmat A](#)
pointer to the original coefficient matrix
- [INT nblk](#)
number of blocks
- [INT * iblock](#)
row index of blocks
- [INT * jblock](#)
column index of blocks
- [REAL * rhsloc](#)
temp work space ???
- [dvector rhsloc1](#)
local right hand side
- [dvector xloc1](#)
local solution
- [REAL * au](#)
LU decomposition: the U block.
- [REAL * al](#)
LU decomposition: the L block.
- [INT SWZ_type](#)
Schwarz method type.
- [INT blk_solver](#)
Schwarz block solver.
- [INT memt](#)
working space size
- [INT * mask](#)
mask
- [INT maxbs](#)
maximal block size
- [INT * maxa](#)
maxa
- [dCSRmat * blk_data](#)
matrix for each partition
- [Mumps_data * mumps](#)
param for MUMPS
- [SWZ_param * swzparam](#)
param for Schwarz

8.35.1 Detailed Description

Data for Schwarz methods.

This is needed for the Schwarz solver/preconditioner/smoothier.

Definition at line 712 of file [fasp.h](#).

8.35.2 Field Documentation

8.35.2.1 A

`dCSRmat` A

pointer to the original coefficient matrix

Definition at line 717 of file [fasp.h](#).

8.35.2.2 al

`REAL*` al

LU decomposition: the L block.

Definition at line 743 of file [fasp.h](#).

8.35.2.3 au

`REAL*` au

LU decomposition: the U block.

Definition at line 740 of file [fasp.h](#).

8.35.2.4 blk_data

`dCSRmat*` blk_data

matrix for each partition

Definition at line 764 of file [fasp.h](#).

8.35.2.5 blk_solver

`INT blk_solver`

Schwarz block solver.

Definition at line 749 of file [fasp.h](#).

8.35.2.6 iblock

`INT* iblock`

row index of blocks

Definition at line 725 of file [fasp.h](#).

8.35.2.7 jblock

`INT* jblock`

column index of blocks

Definition at line 728 of file [fasp.h](#).

8.35.2.8 mask

`INT* mask`

mask

Definition at line 755 of file [fasp.h](#).

8.35.2.9 maxa

`INT* maxa`

maxa

Definition at line 761 of file [fasp.h](#).

8.35.2.10 maxbs

`INT maxbs`

maximal block size

Definition at line 758 of file [fasp.h](#).

8.35.2.11 memt

`INT memt`

working space size

Definition at line 752 of file [fasp.h](#).

8.35.2.12 mumps

`Mumps_data*` mumps

param for MUMPS

Definition at line 777 of file [fasp.h](#).

8.35.2.13 nblk

`INT nblk`

number of blocks

Definition at line 722 of file [fasp.h](#).

8.35.2.14 rhsloc

`REAL*` rhsloc

temp work space ???

Definition at line 731 of file [fasp.h](#).

8.35.2.15 rhsloc1

`dvector` rhsloc1

local right hand side

Definition at line 734 of file [fasp.h](#).

8.35.2.16 SWZ_type

`INT` SWZ_type

Schwarz method type.

Definition at line 746 of file [fasp.h](#).

8.35.2.17 swzparam

`SWZ_param*` swzparam

param for Schwarz

Definition at line 780 of file [fasp.h](#).

8.35.2.18 xloc1

`dvector` xloc1

local solution

Definition at line 737 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

8.36 SWZ_param Struct Reference

Parameters for Schwarz method.

```
#include <fasp.h>
```

Data Fields

- [SHORT print_level](#)
print leve
- [SHORT SWZ_type](#)
type for Schwarz method
- [INT SWZ_maxlvl](#)
maximal level for constructing the blocks
- [INT SWZ_mmsize](#)
maximal size of blocks
- [INT SWZ_blksolver](#)
type of Schwarz block solver

8.36.1 Detailed Description

Parameters for Schwarz method.

Definition at line [422](#) of file [fasp.h](#).

8.36.2 Field Documentation

8.36.2.1 print_level

[SHORT](#) print_level

print leve

Definition at line [425](#) of file [fasp.h](#).

8.36.2.2 SWZ_blksolver

[INT](#) SWZ_blksolver

type of Schwarz block solver

Definition at line [437](#) of file [fasp.h](#).

8.36.2.3 SWZ_maxlvl

`INT` SWZ_maxlvl

maximal level for constructing the blocks

Definition at line 431 of file [fasp.h](#).

8.36.2.4 SWZ_mmsize

`INT` SWZ_mmsize

maximal size of blocks

Definition at line 434 of file [fasp.h](#).

8.36.2.5 SWZ_type

`SHORT` SWZ_type

type for Schwarz method

Definition at line 428 of file [fasp.h](#).

The documentation for this struct was generated from the following file:

- [fasp.h](#)

Chapter 9

File Documentation

9.1 doxygen.h File Reference

Main page for Doygen documentation.

9.1.1 Detailed Description

Main page for Doygen documentation.
Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Less Public License 3.0 or later.

Definition in file [doxygen.h](#).

9.2 doxygen.h

[Go to the documentation of this file.](#)

```
00001
00183 /*-----*/
00184 /*--      End of File      --*/
00185 /*-----*/
00186
```

9.3 XtrMumps.c File Reference

Interface to MUMPS direct solvers.

```
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Macros

- `#define ICNTL(l) icntl[(l)-1]`

Functions

- int `fasp_solver_mumps` (`dCSRmat` *ptrA, `dvector` *b, `dvector` *u, const `SHORT` prtlvl)
Solve $Ax=b$ by MUMPS directly.
- int `fasp_solver_mumps_steps` (`dCSRmat` *ptrA, `dvector` *b, `dvector` *u, `Mumps_data` *mumps)
Solve $Ax=b$ by MUMPS in three steps.

9.3.1 Detailed Description

Interface to MUMPS direct solvers.

Reference for MUMPS: <http://mumps.enseeiht.fr/>

Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [XtrMumps.c](#).

9.3.2 Macro Definition Documentation

9.3.2.1 ICNTL

```
#define ICNTL(
    I ) icntl[(I)-1]
macro s.t. indices match documentation
Definition at line 23 of file XtrMumps.c.
```

9.3.3 Function Documentation

9.3.3.1 fasp_solver_mumps()

```
int fasp_solver_mumps (
    dCSRmat * ptrA,
    dvector * b,
    dvector * u,
    const SHORT prtlvl )
```

Solve $Ax=b$ by MUMPS directly.

Parameters

<i>ptrA</i>	Pointer to a <code>dCSRmat</code> matrix
<i>b</i>	Pointer to the dvector of right-hand side term
<i>u</i>	Pointer to the dvector of solution
<i>prtlvl</i>	Output level

Author

Chunsheng Feng

Date

02/27/2013

Modified by Chensong Zhang on 02/27/2013 for new FASP function names.
Definition at line 45 of file [XtrMumps.c](#).

9.3.3.2 fasp_solver_mumps_steps()

```
int fasp_solver_mumps_steps (
    dCSRmat * ptrA,
    dvector * b,
    dvector * u,
    Mumps_data * mumps )
```

Solve $Ax=b$ by MUMPS in three steps.

Parameters

<i>ptrA</i>	Pointer to a dCSRmat matrix
<i>b</i>	Pointer to the dvector of right-hand side term
<i>u</i>	Pointer to the dvector of solution
<i>mumps</i>	Pointer to MUMPS data

Author

Chunsheng Feng

Date

02/27/2013

Modified by Chensong Zhang on 02/27/2013 for new FASP function names. Modified by Zheng Li on 10/10/2014 to adjust input parameters. Modified by Chunsheng Feng on 08/11/2017 for debug information.
Definition at line 188 of file [XtrMumps.c](#).

9.4 XtrMumps.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <time.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_functs.h"
00018
00019 #if WITH_MUMPS
00020 #include "dmumps_c.h"
00021 #endif
00022
00023 #define ICNTL(I) icntl[(I)-1]
00025 /*-----*/
00026 /*--      Public Functions      --*/
00027 /*-----*/
00028
00045 int fasp_solver_mumps(dCSRmat* ptrA, dvector* b, dvector* u, const SHORT prtlvl)
00046 {
00047
00048 #if WITH_MUMPS
00049
00050     DMUMPS_STRUC_C id;
00051
00052     const int n = ptrA->row;
```

```

00053     const int nz = ptrA->nnz;
00054     int*      IA = ptrA->IA;
00055     int*      JA = ptrA->JA;
00056     double*   AA = ptrA->val;
00057     double*   f  = b->val;
00058     double*   x  = u->val;
00059
00060     int*      irn;
00061     int*      jcn;
00062     double*   a;
00063     double*   rhs;
00064     int       i, j;
00065     int       begin_row, end_row;
00066
00067 #if DEBUG_MODE
00068     printf("### DEBUG: fasp_solver_mumps ... [Start]\n");
00069     printf("### DEBUG: nr=%d, nnz=%d\n", n, nz);
00070 #endif
00071
00072     // First check the matrix format
00073     if (IA[0] != 0 && IA[0] != 1) {
00074         printf("### ERROR: Matrix format is wrong -- IA[0] = %d\n", IA[0]);
00075         return ERROR_SOLVER_EXIT;
00076     }
00077
00078     REAL start_time, end_time;
00079     fasp_gettime(&start_time);
00080
00081     /* Define A and rhs */
00082     irn = (int*)malloc(sizeof(int) * nz);
00083     jcn = (int*)malloc(sizeof(int) * nz);
00084     a   = (double*)malloc(sizeof(double) * nz);
00085     rhs = (double*)malloc(sizeof(double) * n);
00086
00087     if (IA[0] == 0) { // C-convention
00088         for (i = 0; i < n; i++) {
00089             begin_row = IA[i];
00090             end_row   = IA[i + 1];
00091             for (j = begin_row; j < end_row; j++) {
00092                 irn[j] = i + 1;
00093                 jcn[j] = JA[j] + 1;
00094                 a[j]   = AA[j];
00095             }
00096         }
00097     } else { // F-convention
00098         for (i = 0; i < n; i++) {
00099             begin_row = IA[i] - 1;
00100             end_row   = IA[i + 1] - 1;
00101             for (j = begin_row; j < end_row; j++) {
00102                 irn[j] = i + 1;
00103                 jcn[j] = JA[j];
00104                 a[j]   = AA[j];
00105             }
00106         }
00107     }
00108
00109     /* Initialize a MUMPS instance. */
00110     id.job      = -1;
00111     id.par      = 1; // host involved in factorization/solve
00112     id.sym      = 0; // 0: general, 1: spd, 2: sym
00113     id.comm_fortran = 0;
00114     dmumps_c(&id);
00115
00116     /* Define the problem on the host */
00117     id.n        = n;
00118     id.nz       = nz;
00119     id.irn      = irn;
00120     id.jcn      = jcn;
00121     id.a        = a;
00122     id.rhs      = rhs;
00123
00124     if (prtlvl < PRINT_MOST) { // no debug
00125         id.ICNTL(1) = -1;
00126         id.ICNTL(2) = -1;
00127         id.ICNTL(3) = -1;
00128         id.ICNTL(4) = 0;
00129     } else { // debug
00130         id.ICNTL(1) = 6; // err output stream
00131         id.ICNTL(2) = 6; // warn/info output stream
00132         id.ICNTL(3) = 6; // global output stream
00133         id.ICNTL(4) = 3; // 0:none, 1: err, 2: warn/stats, 3:diagnostics, 4:parameters

```



```

00134     }
00135
00136     /* Call the MUMPS package */
00137     for (i = 0; i < n; i++) rhs[i] = f[i];
00138
00139     id.job = 6; /* Combines phase 1, 2, and 3 */
00140     dmumps_c(&id); /* Sometimes segmentation faults in phase 1 */
00141
00142     for (i = 0; i < n; i++) x[i] = id.rhs[i];
00143
00144     id.job = -2;
00145     dmumps_c(&id); /* Terminate instance */
00146
00147     free(irn);
00148     free(jcn);
00149     free(a);
00150     free(rhs);
00151
00152     if (prtlvl > PRINT_MIN) {
00153         fasp_gettime(&end_time);
00154         fasp_cputime("MUMPS solver", end_time - start_time);
00155     }
00156
00157 #if DEBUG_MODE
00158     printf("### DEBUG: fasp_solver_mumps ... [Finish]\n");
00159 #endif
00160     return FASP_SUCCESS;
00161
00162 #else
00163
00164     printf("### ERROR: MUMPS is not available!\n");
00165     return ERROR_SOLVER_EXIT;
00166
00167 #endif
00168 }
00169
00188 int fasp_solver_mumps_steps(dCSRmat* ptrA, dvector* b, dvector* u, Mumps_data* mumps)
00189 {
00190     #if WITH_MUMPS
00191
00192         DMUMPS_STRUC_C id;
00193
00194         int job = mumps->job;
00195
00196         static int job_stat = 0;
00197         int i, j;
00198
00199         int* irn;
00200         int* jcn;
00201         double* a;
00202         double* rhs;
00203
00204         switch (job) {
00205
00206             case 1:
00207             {
00208                 #if DEBUG_MODE
00209                     printf("### DEBUG: %s, step %d, job_stat = %d... [Start]\n",
00210                           __FUNCTION__, job, job_stat);
00211                 #endif
00212
00213                 int begin_row, end_row;
00214                 const int n = ptrA->row;
00215                 const int nz = ptrA->nnz;
00216                 int* IA = ptrA->IA;
00217                 int* JA = ptrA->JA;
00218                 double* AA = ptrA->val;
00219
00220                 irn = id.irn = (int*)malloc(sizeof(int) * nz);
00221                 jcn = id.jcn = (int*)malloc(sizeof(int) * nz);
00222                 a = id.a = (double*)malloc(sizeof(double) * nz);
00223                 rhs = id.rhs = (double*)malloc(sizeof(double) * n);
00224
00225                 // First check the matrix format
00226                 if (IA[0] != 0 && IA[0] != 1) {
00227                     printf("### ERROR: Matrix format is wrong, IA[0] = %d!\n", IA[0]);
00228                     return ERROR_SOLVER_EXIT;
00229                 }
00230
00231                 // Define A and rhs
00232                 if (IA[0] == 0) { // C-convention
00233                     for (i = 0; i < n; i++) {

```

```

00233         begin_row = IA[i];
00234         end_row   = IA[i + 1];
00235         for (j = begin_row; j < end_row; j++) {
00236             irn[j] = i + 1;
00237             jcn[j] = JA[j] + 1;
00238             a[j]   = AA[j];
00239         }
00240     }
00241 } else { // F-convention
00242     for (i = 0; i < n; i++) {
00243         begin_row = IA[i] - 1;
00244         end_row   = IA[i + 1] - 1;
00245         for (j = begin_row; j < end_row; j++) {
00246             irn[j] = i + 1;
00247             jcn[j] = JA[j];
00248             a[j]   = AA[j];
00249         }
00250     }
00251 }
00252
00253 /* Initialize a MUMPS instance. */
00254 id.job      = -1;
00255 id.par      = 1;
00256 id.sym      = 0;
00257 id.comm_fortran = 0;
00258 dmumps_c(&id);
00259
00260 /* Define the problem on the host */
00261 id.n        = n;
00262 id.nz       = nz;
00263 id.irn      = irn;
00264 id.jcn      = jcn;
00265 id.a        = a;
00266 id.rhs      = rhs;
00267
00268 /* No outputs */
00269 id.ICNTL(1) = -1;
00270 id.ICNTL(2) = -1;
00271 id.ICNTL(3) = -1;
00272 id.ICNTL(4) = 0;
00273
00274 id.job = 4;
00275 dmumps_c(&id);
00276 job_stat = 1;
00277
00278 mumps->id = id;
00279
00280 #if DEBUG_MODE
00281 printf("### DEBUG: %s, step %d, job_stat = %d... [Finish]\n",
00282        __FUNCTION__, job, job_stat);
00283 #endif
00284 break;
00285     }
00286
00287     case 2:
00288     {
00289 #if DEBUG_MODE
00290 printf("### DEBUG: %s, step %d, job_stat = %d... [Start]\n",
00291        __FUNCTION__, job, job_stat);
00292 #endif
00293 id = mumps->id;
00294
00295 if (job_stat != 1)
00296     printf("### ERROR: %s setup failed!\n", __FUNCTION__);
00297
00298 /* Call the MUMPS package. */
00299 for (i = 0; i < id.n; i++) id.rhs[i] = b->val[i];
00300
00301 id.job = 3;
00302 dmumps_c(&id);
00303
00304 for (i = 0; i < id.n; i++) u->val[i] = id.rhs[i];
00305
00306 #if DEBUG_MODE
00307 printf("### DEBUG: %s, step %d, job_stat = %d... [Finish]\n",
00308        __FUNCTION__, job, job_stat);
00309 #endif
00310 break;
00311     }
00312
00313     case 3:

```

```

00314     {
00315     #if DEBUG_MODE
00316         printf("### DEBUG: %s, step %d, job_stat = %d... [Start]\n",
00317             __FUNCTION__, job, job_stat);
00318     #endif
00319         id = mumps->id;
00320
00321         if (job_stat != 1)
00322             printf("### ERROR: %s setup failed!\n", __FUNCTION__);
00323
00324         free(id.irn);
00325         free(id.jcn);
00326         free(id.a);
00327         free(id.rhs);
00328         id.job = -2;
00329         dmumps_c(&id); /* Terminate instance */
00330
00331     #if DEBUG_MODE
00332         printf("### DEBUG: %s, step %d, job_stat = %d... [Finish]\n",
00333             __FUNCTION__, job, job_stat);
00334     #endif
00335
00336         break;
00337     }
00338
00339     default:
00340         printf("### ERROR: job = %d. Should be 1, 2, or 3!\n", job);
00341         return ERROR_SOLVER_EXIT;
00342     }
00343
00344     return FASP_SUCCESS;
00345
00346 #else
00347     printf("### ERROR: MUMPS is not available!\n");
00348     return ERROR_SOLVER_EXIT;
00349 #endif
00350 }
00351
00352 #if WITH_MUMPS
00353 Mumps_data fasp_mumps_factorize(dCSRmat* ptrA, dvector* b, dvector* u,
00354     const SHORT prtlvl)
00355 {
00356     Mumps_data mumps;
00357     DMUMPS_STRUC_C id;
00358
00359     int i, j;
00360     const int m = ptrA->row;
00361     const int n = ptrA->col;
00362     const int nz = ptrA->nnz;
00363     int* IA = ptrA->IA;
00364     int* JA = ptrA->JA;
00365     double* AA = ptrA->val;
00366
00367     int* irn = id.irn = (int*)malloc(sizeof(int) * nz);
00368     int* jcn = id.jcn = (int*)malloc(sizeof(int) * nz);
00369     double* a = id.a = (double*)malloc(sizeof(double) * nz);
00370     double* rhs = id.rhs = (double*)malloc(sizeof(double) * n);
00371
00372     int begin_row, end_row;
00373
00374     #if DEBUG_MODE
00375         printf("### DEBUG: %s ... [Start]\n", __FUNCTION__);
00376         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nz);
00377     #endif
00378
00379     clock_t start_time = clock();
00380
00381     if (IA[0] == 0) { // C-convention
00382         for (i = 0; i < n; i++) {
00383             begin_row = IA[i];
00384             end_row = IA[i + 1];
00385             for (j = begin_row; j < end_row; j++) {
00386                 irn[j] = i + 1;
00387                 jcn[j] = JA[j] + 1;
00388                 a[j] = AA[j];
00389             }
00390         }
00391     } else { // F-convention
00392         for (i = 0; i < n; i++) {

```

```

00408         begin_row = IA[i] - 1;
00409         end_row   = IA[i + 1] - 1;
00410         for (j = begin_row; j < end_row; j++) {
00411             irn[j] = i + 1;
00412             jcn[j] = JA[j];
00413             a[j]   = AA[j];
00414         }
00415     }
00416 }
00417
00418 /* Initialize a MUMPS instance. */
00419 id.job      = -1;
00420 id.par      = 1;
00421 id.sym      = 0;
00422 id.comm_fortran = 0;
00423 dmumps_c(&id);
00424
00425 /* Define the problem on the host */
00426 id.n        = n;
00427 id.nz       = nz;
00428 id.irn      = irn;
00429 id.jcn      = jcn;
00430 id.a        = a;
00431 id.rhs      = rhs;
00432
00433 if (prtlvl < PRINT_MOST) { // no debug
00434     id.ICNTL(1) = -1;
00435     id.ICNTL(2) = -1;
00436     id.ICNTL(3) = -1;
00437     id.ICNTL(4) = 0;
00438 } else { // debug
00439     id.ICNTL(1) = 6; // err output stream
00440     id.ICNTL(2) = 6; // warn/info output stream
00441     id.ICNTL(3) = 6; // global output stream
00442     id.ICNTL(4) = 3; // 0:none, 1: err, 2: warn/stats, 3:diagnostics, 4:parameters
00443 }
00444
00445 id.job = 4;
00446 dmumps_c(&id);
00447
00448 if (prtlvl > PRINT_MIN) {
00449     clock_t end_time = clock();
00450     double fac_time = (double)(end_time - start_time) / (double)(CLOCKS_PER_SEC);
00451     printf("MUMPS factorize costs %f seconds.\n", fac_time);
00452 }
00453
00454 #if DEBUG_MODE
00455     printf("### DEBUG: %s ... [Finish]\n", __FUNCTION__);
00456 #endif
00457
00458 mumps.id = id;
00459
00460 return mumps;
00461 }
00462 #endif
00463
00464 #if WITH_MUMPS
00479 void fasp_mumps_solve(dCSRmat* ptrA, dvector* b, dvector* u, Mumps_data mumps,
00480                     const SHORT prtlvl)
00481 {
00482     int i, j;
00483
00484     DMUMPS_STRUC_C id = mumps.id;
00485
00486     const int m = ptrA->row;
00487     const int n = ptrA->row;
00488     const int nz = ptrA->nnz;
00489     int*      IA = ptrA->IA;
00490     int*      JA = ptrA->JA;
00491     double*   AA = ptrA->val;
00492
00493     int*      irn = id.irn;
00494     int*      jcn = id.jcn;
00495     double*   a   = id.a;
00496     double*   rhs = id.rhs;
00497
00498 #if DEBUG_MODE
00499     printf("### DEBUG: %s ... [Start]\n", __FUNCTION__);
00500     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nz);
00501 #endif
00502

```

```

00503     clock_t start_time = clock();
00504
00505     double* f = b->val;
00506     double* x = u->val;
00507
00508     /* Call the MUMPS package. */
00509     for (i = 0; i < id.n; i++) rhs[i] = f[i];
00510
00511     if (prtlvl < PRINT_MOST) { // no debug
00512         id.ICNTL(1) = -1;
00513         id.ICNTL(2) = -1;
00514         id.ICNTL(3) = -1;
00515         id.ICNTL(4) = 0;
00516     } else { // debug
00517         id.ICNTL(1) = 6; // err output stream
00518         id.ICNTL(2) = 6; // warn/info output stream
00519         id.ICNTL(3) = 6; // global output stream
00520         id.ICNTL(4) = 3; // 0:none, 1: err, 2: warn/stats, 3:diagnostics, 4:parameters
00521     }
00522
00523     id.job = 3;
00524     dmumps_c(&id);
00525
00526     for (i = 0; i < id.n; i++) x[i] = id.rhs[i];
00527
00528     if (prtlvl > PRINT_NONE) {
00529         clock_t end_time = clock();
00530         double solve_time = (double)(end_time - start_time) / (double)(CLOCKS_PER_SEC);
00531         printf("MUMPS costs %f seconds.\n", solve_time);
00532     }
00533
00534     #if DEBUG_MODE
00535     printf("### DEBUG: %s ... [Finish]\n", __FUNCTION__);
00536     #endif
00537 }
00538 #endif
00539
00540 #if WITH_MUMPS
00541 void fasp_mumps_free(Mumps_data* mumps)
00542 {
00543     DMUMPS_STRUC_C id = mumps->id;
00544
00545     free(id.irn);
00546     free(id.jcn);
00547     free(id.a);
00548     free(id.rhs);
00549 }
00550 #endif
00551
00552 /*-----*/
00553 /*--          End of File          --*/
00554 /*-----*/

```

9.5 XtrPardiso.c File Reference

Interface to Intel MKL PARDISO direct solvers.

```

#include <time.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- `INT fasp_solver_pardiso (dCSRmat *ptrA, dvector *b, dvector *u, const SHORT prtlvl)`
Solve $Ax=b$ by PARDISO directly.

9.5.1 Detailed Description

Interface to Intel MKL PARDISO direct solvers.

Reference for Intel MKL PARDISO: <https://software.intel.com/en-us/node/470282>
Copyright (C) 2015–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [XtrPardiso.c](#).

9.5.2 Function Documentation

9.5.2.1 fasp_solver_pardiso()

```
INT fasp_solver_pardiso (
    dCSRmat * ptrA,
    dvector * b,
    dvector * u,
    const SHORT prtlvl )
```

Solve $Ax=b$ by PARDISO directly.

Parameters

<i>ptrA</i>	Pointer to a dCSRmat matrix
<i>b</i>	Pointer to the dvector of right-hand side term
<i>u</i>	Pointer to the dvector of solution
<i>prtlvl</i>	Output level

Author

Hongxuan Zhang

Date

11/28/2015

Note

Each row of A should be in ascending order w.r.t. column indices.

Definition at line 45 of file [XtrPardiso.c](#).

9.6 XtrPardiso.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <time.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_funcs.h"
00018
00019 #if WITH_PARDISO
00020 #include "mkl_pardiso.h"
00021 #include "mkl_types.h"
00022 #include "mkl_spblas.h"
00023 #endif
00024
00025 /*-----*/
00026 /*--      Public Functions      --*/
00027 /*-----*/
00028
00045 INT fasp_solver_pardiso (dCSRmat * ptrA,
00046                          dvector *b,
00047                          dvector *u,
```

```

00048                                     const SHORT prtlvl)
00049 {
00050     #if WITH_PARDISO
00051
00052         INT status = FASP_SUCCESS;
00053
00054         MKL_INT n = ptrA->col;
00055         MKL_INT *ia = ptrA->IA;
00056         MKL_INT *ja = ptrA->JA;
00057         REAL *a = ptrA->val;
00058
00059         MKL_INT mtype = 11; /* Real unsymmetric matrix */
00060         MKL_INT nrhs = 1; /* Number of right hand sides */
00061         MKL_INT idum; /* Integer dummy */
00062         MKL_INT iparm[64]; /* Pardiso control parameters */
00063         MKL_INT maxfct, mnum, phase, error, msglvl; /* Auxiliary variables */
00064
00065         REAL *f = b->val; /* RHS vector */
00066         REAL *x = u->val; /* Solution vector */
00067         void *pt[64]; /* Internal solver memory pointer pt */
00068         double ddum; /* Double dummy */
00069
00070     #if DEBUG_MODE
00071         printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00072         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00073     #endif
00074
00075     REAL start_time, end_time;
00076     fasp_gettime(&start_time);
00077
00078     PARDISOINIT(pt, &mtype, iparm); /* Initialize */
00079     iparm[34] = 1; /* Use 0-based indexing */
00080     maxfct = 1; /* Maximum number of numerical factorizations */
00081     mnum = 1; /* Which factorization to use */
00082     msglvl = 0; /* Do not print statistical information in file */
00083     error = 0; /* Initialize error flag */
00084
00085     phase = 11; /* Reordering and symbolic factorization */
00086     PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
00087             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
00088     if ( error != 0 ) {
00089         printf ("### ERROR: Symbolic factorization failed %d!\n", error);
00090         exit (1);
00091     }
00092
00093     phase = 22; /* Numerical factorization */
00094     PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
00095             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
00096     if ( error != 0 ) {
00097         printf ("### ERROR: Numerical factorization failed %d!\n", error);
00098         exit (2);
00099     }
00100
00101     phase = 33; /* Back substitution and iterative refinement */
00102     PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
00103             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, f, x, &error);
00104
00105     if ( error != 0 ) {
00106         printf ("### ERROR: Solution failed %d!\n", error);
00107         exit (3);
00108     }
00109
00110     if ( prtlvl > PRINT_MIN ) {
00111         fasp_gettime(&end_time);
00112         fasp_cputime("PARDISO solver", end_time - start_time);
00113     }
00114
00115     phase = -1; /* Release internal memory */
00116     PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
00117             &n, &ddum, ia, ja, &idum, &nrhs,
00118             iparm, &msglvl, &ddum, &ddum, &error);
00119
00120     #if DEBUG_MODE
00121         printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00122     #endif
00123
00124     return status;
00125
00126 #else
00127     printf("### ERROR: PARDISO is not available!\n");

```

```

00129     return ERROR_SOLVER_EXIT;
00130
00131 #endif
00132 }
00133 }
00134
00135 #if WITH_PARDISO
00149 INT fasp_pardiso_factorize (dCSRmat *ptrA,
00150                             Pardiso_data *pdata,
00151                             const SHORT prtlvl)
00152 {
00153     INT status = FASP_SUCCESS;
00154
00155     MKL_INT n = ptrA->col;
00156     MKL_INT *ia = ptrA->IA;
00157     MKL_INT *ja = ptrA->JA;
00158     REAL *a = ptrA->val;
00159
00160     double ddum;          /* Double dummy */
00161     MKL_INT nrhs = 1;     /* Number of right hand sides */
00162     MKL_INT idum;         /* Integer dummy */
00163     MKL_INT phase, error, msglvl; /* Auxiliary variables */
00164
00165 #if DEBUG_MODE
00166     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00167     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00168 #endif
00169
00170     pdata->mtype = 11;     /* Real unsymmetric matrix */
00171
00172     PARDISOINIT(pdata->pt, &(pdata->mtype), pdata->iparm); /* Initialize */
00173     pdata->iparm[34] = 1; /* Use 0-based indexing */
00174
00175     /* Numbers of processors, value of OMP_NUM_THREADS */
00176 #ifdef _OPENMP
00177     pdata->iparm[2] = fasp_get_num_threads();
00178 #endif
00179
00180     REAL start_time, end_time;
00181     fasp_gettime(&start_time);
00182
00183     pdata->maxfct = 1;     /* Maximum number of numerical factorizations */
00184     pdata->mnum = 1;       /* Which factorization to use */
00185     msglvl = 0;           /* Do not print statistical information in file */
00186     error = 0;            /* Initialize error flag */
00187
00188     phase = 11; /* Reordering and symbolic factorization */
00189     PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase, &n,
00190             a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum, &ddum, &error);
00191     if ( error != 0 ) {
00192         printf ("### ERROR: Symbolic factorization failed %d!\n", error);
00193         exit (1);
00194     }
00195
00196     phase = 22; /* Numerical factorization */
00197     PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase, &n,
00198             a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum, &ddum, &error);
00199
00200     if ( error != 0 ) {
00201         printf ("### ERROR: Numerical factorization failed %d!\n", error);
00202         exit (2);
00203     }
00204
00205     if ( prtlvl > PRINT_MIN ) {
00206         fasp_gettime(&end_time);
00207         fasp_cputime("PARDISO setup", end_time - start_time);
00208     }
00209
00210 #if DEBUG_MODE
00211     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00212 #endif
00213
00214     return status;
00215 }
00216
00232 INT fasp_pardiso_solve (dCSRmat *ptrA,
00233                        dvector *b,
00234                        dvector *u,
00235                        Pardiso_data *pdata,
00236                        const SHORT prtlvl)
00237 {

```



```

00238     INT status = FASP_SUCCESS;
00239
00240     MKL_INT n = ptrA->col;
00241     MKL_INT *ia = ptrA->IA;
00242     MKL_INT *ja = ptrA->JA;
00243
00244     REAL *a = ptrA->val;
00245     REAL *f = b->val;      /* RHS vector */
00246     REAL *x = u->val;      /* Solution vector */
00247     MKL_INT nrhs = 1;      /* Number of right hand sides */
00248     MKL_INT idum;          /* Integer dummy */
00249     MKL_INT phase, error, msglvl; /* Auxiliary variables */
00250
00251     REAL start_time, end_time;
00252     fasp_gettime(&start_time);
00253
00254     msglvl = 0; /* Do not print statistical information in file */
00255
00256     phase = 33; /* Back substitution and iterative refinement */
00257     PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase,
00258             &n, a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, f, x, &error);
00259
00260     if ( error != 0 ) {
00261         printf ("### ERROR: Solution failed %d!\n", error);
00262         exit (3);
00263     }
00264
00265     if ( prtlvl > PRINT_MIN ) {
00266         fasp_gettime(&end_time);
00267         fasp_cputime("PARDISO solve", end_time - start_time);
00268     }
00269
00270 #if DEBUG_MODE
00271     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00272 #endif
00273
00274     return status;
00275 }
00276
00286 INT fasp_pardiso_free_internal_mem (Pardiso_data *pdata)
00287 {
00288     INT status = FASP_SUCCESS;
00289
00290     MKL_INT *ia = NULL;
00291     MKL_INT *ja = NULL;
00292
00293     double ddum;      /* Double dummy */
00294     MKL_INT idum;      /* Integer dummy */
00295     MKL_INT nrhs = 1;  /* Number of right hand sides */
00296     MKL_INT phase, error, msglvl; /* Auxiliary variables */
00297
00298     msglvl = 0; /* Do not print statistical information in file */
00299
00300 #if DEBUG_MODE
00301     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00302 #endif
00303
00304     phase = -1; /* Release internal memory */
00305     PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase,
00306             &idum, &ddum, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum,
00307             &ddum, &error);
00308
00309 #if DEBUG_MODE
00310     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00311 #endif
00312
00313     return status;
00314 }
00315
00316 #endif
00317
00318 /*-----*/
00319 /*--      End of File      --*/
00320 /*-----*/

```

9.7 XtrSamg.c File Reference

Interface to SAMG solvers.

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- void [dvector2SAMGInput](#) ([dvector](#) *vec, char *filename)
Write a dvector to disk file in SAMG format (coordinate format)
- [INT dCSRmat2SAMGInput](#) ([dCSRmat](#) *A, char *filefrm, char *fileamg)
Write SAMG Input data from a sparse matrix of CSR format.

9.7.1 Detailed Description

Interface to SAMG solvers.

Reference for SAMG: <http://www.scai.fraunhofer.de/geschaeftsfelder/nuso/produkte/samg.html> ↩

Warning

This interface has *only* been tested for SAMG24a1 (2010 version)!

Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [XtrSamg.c](#).

9.7.2 Function Documentation

9.7.2.1 dCSRmat2SAMGInput()

```
INT dCSRmat2SAMGInput (
    dCSRmat * A,
    char * filefrm,
    char * fileamg )
```

Write SAMG Input data from a sparse matrix of CSR format.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>filefrm</i>	Name of the .frm file
<i>fileamg</i>	Name of the .amg file

Author

Zhiyang Zhou

Date

2010/08/25

Definition at line 65 of file [XtrSamg.c](#).

9.7.2.2 dvector2SAMGInput()

```
void dvector2SAMGInput (
    dvector * vec,
    char * filename )
```

Write a dvector to disk file in SAMG format (coordinate format)

Parameters

<i>vec</i>	Pointer to the dvector
<i>filename</i>	File name for input

Author

Zhiyang Zhou

Date

08/25/2010

Definition at line 36 of file [XtrSamg.c](#).

9.8 XtrSamg.c

[Go to the documentation of this file.](#)

```
00001
00016 #include <math.h>
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--      Public Functions      --*/
00024 /*-----*/
00025
00036 void dvector2SAMGInput (dvector *vec,
00037                        char *filename)
00038 {
00039     INT m = vec->row, i;
00040
00041     FILE *fp=fopen(filename,"w");
00042     if ( fp == NULL ) {
00043         printf("### ERROR: Opening file %s failed!\n",filename);
00044         exit(ERROR_OPEN_FILE);
00045     }
00046
00047     printf("%s:  writing vector to '%s'...\n", __FUNCTION__, filename);
00048
00049     for (i=0;i<m;++i) fprintf(fp,"%0.15le\n",vec->val[i]);
00050
00051     fclose(fp);
00052 }
00053
00065 INT dCSRmat2SAMGInput (dCSRmat *A,
00066                      char *filefrm,
00067                      char *fileamg)
00068 {
00069     FILE      *fp          = NULL;
00070     INT       file_base   = 1;
00071
00072     REAL      *A_data      = A -> val;
00073     INT       *A_i         = A -> IA;
00074     INT       *A_j         = A -> JA;
00075     INT       num_rowsA    = A -> row;
00076     INT       num_nonzeros = A_i[num_rowsA] - A_i[0];
00077
00078     INT       matrix_type  = 0;
```

```

00079     INT      rowsum_type   = 0;
00080     INT      symmetry_type = 0;
00081
00082     INT      i,j;
00083     REAL     rowsum;
00084
00085     fasp_dcsr_diagpref(A);
00086
00087     /* check symmetry type of the matrix */
00088     symmetry_type = fasp_check_symm(A);
00089
00090     /* check rowsum type of the matrix */
00091     for (i = 0; i < num_rowsA; ++i) {
00092         rowsum = 0.0;
00093         for (j = A_i[i]; j < A_i[i+1]; ++j) {
00094             rowsum += A_data[j];
00095         }
00096         if (rowsum*rowsum > 0.0) {
00097             rowsum_type = 1;
00098             break;
00099         }
00100     }
00101
00102     /* Get the matrix type of A */
00103     if (symmetry_type == 0) {
00104         if (rowsum_type == 0)
00105             matrix_type = 11;
00106         else
00107             matrix_type = 12;
00108     }
00109     else {
00110         if (rowsum_type == 0)
00111             matrix_type = 21;
00112         else
00113             matrix_type = 22;
00114     }
00115
00116     /* write the *.frm file */
00117     fp = fopen(filefrm, "w");
00118     fprintf(fp, "%s %d\n", "f", 4);
00119     fprintf(fp, "%d %d %d %d %d\n", num_nonzeros, num_rowsA, matrix_type, 1, 0);
00120     fclose(fp);
00121
00122     /* write the *.amg file */
00123     fp = fopen(fileamg, "w");
00124     for (j = 0; j <= num_rowsA; ++j) {
00125         fprintf(fp, "%d\n", A_i[j] + file_base);
00126     }
00127     for (j = 0; j < num_nonzeros; ++j) {
00128         fprintf(fp, "%d\n", A_j[j] + file_base);
00129     }
00130     if (A_data) {
00131         for (j = 0; j < num_nonzeros; ++j) {
00132             fprintf(fp, "%.15le\n", A_data[j]); // we always use "%.15le\n"
00133         }
00134     }
00135     else {
00136         fprintf(fp, "### WARNING: No matrix data!\n");
00137     }
00138     fclose(fp);
00139
00140     return FASP_SUCCESS;
00141 }
00142
00143 /*-----*/
00144 /*--          End of File          --*/
00145 /*-----*/

```

9.9 XtrSuperlu.c File Reference

Interface to SuperLU direct solvers.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "fasp.h"

```

```
#include "fasp_funcs.h"
```

Functions

- int `fasp_solver_superlu` (`dCSRmat` *ptrA, `dvector` *b, `dvector` *u, const `SHORT` prtlvl)
Solve $Au=b$ by SuperLU.

9.9.1 Detailed Description

Interface to SuperLU direct solvers.

Reference for SuperLU: <http://crd-legacy.lbl.gov/~xiaoye/SuperLU/>

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [XtrSuperlu.c](#).

9.9.2 Function Documentation

9.9.2.1 `fasp_solver_superlu()`

```
int fasp_solver_superlu (
    dCSRmat * ptrA,
    dvector * b,
    dvector * u,
    const SHORT prtlvl )
```

Solve $Au=b$ by SuperLU.

Parameters

<i>ptrA</i>	Pointer to a <code>dCSRmat</code> matrix
<i>b</i>	Pointer to the <code>dvector</code> of right-hand side term
<i>u</i>	Pointer to the <code>dvector</code> of solution
<i>prtlvl</i>	Output level

Author

Xiaozhe Hu

Date

11/05/2009

Modified by Chensong Zhang on 02/27/2013 for new FASP function names.

Note

Factorization and solution are combined together!!! Not efficient!!!

Definition at line 47 of file [XtrSuperlu.c](#).

9.10 XtrSuperlu.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <stdio.h>
00015 #include <stdlib.h>
00016 #include <time.h>
00017
00018 #include "fasp.h"
00019 #include "fasp_functs.h"
00020
00021 #if WITH_SuperLU
00022 #include "slu_ddefs.h"
00023 #endif
00024
00025 /*-----*/
00026 /*--      Public Functions      --*/
00027 /*-----*/
00028
00047 int fasp_solver_superlu(dCSRmat* ptrA, dvector* b, dvector* u, const SHORT prtlvl)
00048 {
00049
00050 #if WITH_SuperLU
00051
00052     SuperMatrix A, L, U, B;
00053
00054     int* perm_r; /* row permutations from partial pivoting */
00055     int* perm_c; /* column permutation vector */
00056     int nrhs = 1, info, m = ptrA->row, n = ptrA->col, nnz = ptrA->nnz;
00057
00058     if (prtlvl > PRINT_NONE) printf("superlu: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00059
00060     REAL start_time, end_time;
00061     fasp_gettime(&start_time);
00062
00063     dCSRmat tempA = fasp_dcsr_create(m, n, nnz);
00064     fasp_dcsr_cp(ptrA, &tempA);
00065
00066     dvector tempb = fasp_dvec_create(n);
00067     fasp_dvec_cp(b, &tempb);
00068
00069     /* Create matrix A in the format expected by SuperLU. */
00070     dCreate_CompCol_Matrix(&A, m, n, nnz, tempA.val, tempA.JA, tempA.IA, SLU_NR, SLU_D,
00071                           SLU_GE);
00072
00073     /* Create right-hand side B. */
00074     dCreate_Dense_Matrix(&B, m, nrhs, tempb.val, m, SLU_DN, SLU_D, SLU_GE);
00075
00076     if (!(perm_r = intMalloc(m))) ABORT("Malloc fails for perm_r[.].");
00077     if (!(perm_c = intMalloc(n))) ABORT("Malloc fails for perm_c[.].");
00078
00079     /* Set the default input options. */
00080     superlu_options_t options;
00081     set_default_options(&options);
00082     options.ColPerm = COLAMD; // MMD_AT_PLUS_A; MMD_ATA; NATURAL;
00083
00084     /* Initialize the statistics variables. */
00085     SuperLUStat_t stat;
00086     StatInit(&stat);
00087
00088     /* SuperLU */
00089     dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);
00090
00091     DNformat* BB = (DNformat*)B.Store;
00092     u->val = (double*)BB->nzval;
00093     u->row = n;
00094
00095     if (prtlvl > PRINT_MIN) {
00096         fasp_gettime(&end_time);
00097         fasp_cputime("SUPERLU solver", end_time - start_time);
00098     }
00099
00100     /* De-allocate storage */
00101     SUPERLU_FREE(perm_r);
00102     SUPERLU_FREE(perm_c);
00103     Destroy_CompCol_Matrix(&A);
00104     Destroy_SuperMatrix_Store(&B);
00105     Destroy_SuperNode_Matrix(&L);
00106     Destroy_CompCol_Matrix(&U);
00107     StatFree(&stat);

```

```

00108
00109     return FASP_SUCCESS;
00110
00111 #else
00112
00113     printf("### ERROR: SuperLU is not available!\n");
00114     return ERROR_SOLVER_EXIT;
00115
00116 #endif
00117 }
00118
00119 /*-----*/
00120 /*--      End of File      --*/
00121 /*-----*/

```

9.11 XtrUmfpack.c File Reference

Interface to UMFPACK direct solvers.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- `INT fasp_solver_umfpack (dCSRmat *ptrA, dvector *b, dvector *u, const SHORT prtlvl)`
Solve $Au=b$ by UMFPack.

9.11.1 Detailed Description

Interface to UMFPACK direct solvers.

Reference for SuiteSparse: <http://faculty.cse.tamu.edu/davis/suitesparse.html>
Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [XtrUmfpack.c](#).

9.11.2 Function Documentation

9.11.2.1 fasp_solver_umfpack()

```

INT fasp_solver_umfpack (
    dCSRmat * ptrA,
    dvector * b,
    dvector * u,
    const SHORT prtlvl )

```

Solve $Au=b$ by UMFPack.

Parameters

<i>ptrA</i>	Pointer to a dCSRmat matrix
<i>b</i>	Pointer to the dvector of right-hand side term
<i>u</i>	Pointer to the dvector of solution
<i>prtlvl</i>	Output level

Author

Chensong Zhang

Date

05/20/2010

Modified by Chensong Zhang on 02/27/2013 for new FASP function names. Modified by Chensong Zhang on 08/14/2022 for checking return status.

Definition at line 44 of file [XtrUmfpack.c](#).

9.12 XtrUmfpack.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <time.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_funcs.h"
00018
00019 #if WITH_UMFPACK
00020 #include "umfpack.h"
00021 #endif
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00044 INT fasp_solver_umfpack (dCSRmat *ptrA,
00045                         dvector *b,
00046                         dvector *u,
00047                         const SHORT prtlvl)
00048 {
00049
00050 #if WITH_UMFPACK
00051
00052     const INT n = ptrA->col;
00053
00054     INT *Ap = ptrA->IA;
00055     INT *Ai = ptrA->JA;
00056     double *Ax = ptrA->val;
00057     void *Symbolic, *Numeric;
00058     INT status = FASP_SUCCESS;
00059
00060 #if DEBUG_MODE
00061     const INT m = ptrA->row;
00062     const INT nnz = ptrA->nnz;
00063     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00064     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00065 #endif
00066
00067     REAL start_time, end_time;
00068     fasp_gettime(&start_time);
00069
00070     status = umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
00071     if (status < 0) {
00072         printf("### ERROR: %d, %s %d\n", status, __FUNCTION__, __LINE__);
00073         printf("### ERROR: Symbolic factorization failed!\n");
00074         exit(ERROR_SOLVER_MISC);
00075     }
00076
00077     status = umfpack_di_numeric(Ap, Ai, Ax, Symbolic, &Numeric, NULL, NULL);
00078     if (status < 0) {
00079         printf("### ERROR: %d, %s %d\n", status, __FUNCTION__, __LINE__);
00080         printf("### ERROR: Numerica factorization failed!\n");
00081         exit(ERROR_SOLVER_MISC);
00082     }
00083     umfpack_di_free_symbolic(&Symbolic);
00084
00085     status = umfpack_di_solve(UMFPACK_A, Ap, Ai, Ax, u->val, b->val, Numeric, NULL, NULL);
00086     if (status < 0) {
00087         printf("### ERROR: %d, %s %d\n", status, __FUNCTION__, __LINE__);
00088         printf("### ERROR: UMFPACK solver failed!\n");
00089         exit(ERROR_SOLVER_MISC);

```



```

00090     }
00091     umfpack_di_free_numeric(&Numeric);
00092
00093     if ( prtlvl > PRINT_MIN ) {
00094         fasp_gettime(&end_time);
00095         fasp_cputime("UMFPACK costs", end_time - start_time);
00096     }
00097
00098 #if DEBUG_MODE
00099     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00100 #endif
00101
00102     return status;
00103
00104 #else
00105
00106     printf("### ERROR: UMFPACK is not available!\n");
00107     return ERROR_SOLVER_EXIT;
00108 #endif
00109
00110 }
00111
00112 #if WITH_UMFPACK
00113 void* fasp_umfpack_factorize (dCSRmat *ptrA,
00114                             const SHORT prtlvl)
00115 {
00116     const INT n = ptrA->col;
00117
00118     INT *Ap = ptrA->IA;
00119     INT *Ai = ptrA->JA;
00120     double *Ax = ptrA->val;
00121     void *Symbolic;
00122     void *Numeric;
00123
00124 #if DEBUG_MODE
00125     const INT m = ptrA->row;
00126     const INT nnz = ptrA->nnz;
00127     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00128     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00129 #endif
00130
00131     REAL start_time, end_time;
00132     fasp_gettime(&start_time);
00133
00134     umfpack_di_symbolic (n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
00135     umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, NULL, NULL);
00136     umfpack_di_free_symbolic (&Symbolic);
00137
00138     if ( prtlvl > PRINT_MIN ) {
00139         fasp_gettime(&end_time);
00140         fasp_cputime("UMFPACK setup", end_time - start_time);
00141     }
00142
00143 #if DEBUG_MODE
00144     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00145 #endif
00146
00147     return Numeric;
00148 }
00149
00150 INT fasp_umfpack_solve (dCSRmat *ptrA,
00151                       dvector *b,
00152                       dvector *u,
00153                       void *Numeric,
00154                       const SHORT prtlvl)
00155 {
00156     INT *Ap = ptrA->IA;
00157     INT *Ai = ptrA->JA;
00158     double *Ax = ptrA->val;
00159     INT status = FASP_SUCCESS;
00160
00161 #if DEBUG_MODE
00162     const INT m = ptrA->row;
00163     const INT n = ptrA->col;
00164     const INT nnz = ptrA->nnz;
00165     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00166     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00167 #endif
00168
00169     REAL start_time, end_time;

```

```

00195     fasp_gettime(&start_time);
00196
00197     status = umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, u->val, b->val, Numeric, NULL, NULL);
00198
00199     if ( prtlvl > PRINT_NONE ) {
00200         fasp_gettime(&end_time);
00201         fasp_cputime("UMFPACK solve", end_time - start_time);
00202     }
00203
00204 #if DEBUG_MODE
00205     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00206 #endif
00207
00208     return status;
00209 }
00210
00220 INT fasp_umfpack_free_numeric (void *Numeric)
00221 {
00222     INT status = FASP_SUCCESS;
00223
00224 #if DEBUG_MODE
00225     printf("### DEBUG: %s ..... [Start]\n", __FUNCTION__);
00226 #endif
00227
00228     umfpack_di_free_numeric (&Numeric);
00229
00230 #if DEBUG_MODE
00231     printf("### DEBUG: %s ..... [Finish]\n", __FUNCTION__);
00232 #endif
00233
00234     return status;
00235 }
00236
00237 #endif
00238
00239 /*-----*/
00240 /*--      End of File      --*/
00241 /*-----*/

```

9.13 fasp.h File Reference

Main header file for the FASP project.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fasp_const.h"

```

Data Structures

- struct [ddenmat](#)
Dense matrix of REAL type.
- struct [idenmat](#)
Dense matrix of INT type.
- struct [dCSRmat](#)
Sparse matrix of REAL type in CSR format.
- struct [iCSRmat](#)
Sparse matrix of INT type in CSR format.
- struct [dCOOmat](#)
Sparse matrix of REAL type in COO (IJ) format.
- struct [iCOOmat](#)
Sparse matrix of INT type in COO (IJ) format.
- struct [dCSRLmat](#)
Sparse matrix of REAL type in CSRL format.

- struct [dSTRmat](#)
Structure matrix of REAL type.
- struct [dvector](#)
Vector with n entries of REAL type.
- struct [ivector](#)
Vector with n entries of INT type.
- struct [ITS_param](#)
Parameters for iterative solvers.
- struct [ILU_param](#)
Parameters for ILU.
- struct [SWZ_param](#)
Parameters for Schwarz method.
- struct [AMG_param](#)
Parameters for AMG methods.
- struct [Mumps_data](#)
Data for MUMPS interface.
- struct [Pardiso_data](#)
Data for Intel MKL PARDISO interface.
- struct [ILU_data](#)
Data for ILU setup.
- struct [SWZ_data](#)
Data for Schwarz methods.
- struct [AMG_data](#)
Data for AMG methods.
- struct [precond_data](#)
Data for preconditioners.
- struct [precond_data_str](#)
Data for preconditioners in [dSTRmat](#) format.
- struct [precond_diag_str](#)
Data for diagonal preconditioners in [dSTRmat](#) format.
- struct [precond](#)
Preconditioner data and action.
- struct [mxv_matfree](#)
Matrix-vector multiplication, replace the actual matrix.
- struct [input_param](#)
Input parameters.

Macros

- #define [__FASP_HEADER__](#)
- #define [FASP_VERSION](#) 2.0
FASP base version information.
- #define [MULTI_COLOR_ORDER](#) OFF
- #define [DLMALLOC](#) OFF
For external software package support.
- #define [NEDMALLOC](#) OFF
- #define [RS_C1](#) ON
Flags for internal uses.

- `#define` [DIAGONAL_PREF](#) `OFF`
- `#define` [SHORT](#) `short`
FASP integer and floating point numbers.
- `#define` [INT](#) `int`
- `#define` [LONG](#) `long`
- `#define` [LONGLONG](#) `long long`
- `#define` [REAL](#) `double`
- `#define` [LONGREAL](#) `long double`
- `#define` [STRLEN](#) `256`
- `#define` [MAX](#)(a, b) `((a)>(b))?(a):(b)`
Definition of max, min, abs.
- `#define` [MIN](#)(a, b) `((a)<(b))?(a):(b)`
- `#define` [ABS](#)(a) `((a)>=0.0)?(a):- (a)`
- `#define` [GT](#)(a, b) `((a)>(b))?(TRUE):(FALSE)`
Definition of >, >=, <, <=, and isnan.
- `#define` [GE](#)(a, b) `((a)>=(b))?(TRUE):(FALSE)`
- `#define` [LS](#)(a, b) `((a)<(b))?(TRUE):(FALSE)`
- `#define` [LE](#)(a, b) `((a)<=(b))?(TRUE):(FALSE)`
- `#define` [ISNAN](#)(a) `((a)!= (a))?(TRUE):(FALSE)`
- `#define` [PUT_INT](#)(A) `printf("### DEBUG: %s = %d\n", #A, (A))`
Definition of print command in DEBUG mode.
- `#define` [PUT_REAL](#)(A) `printf("### DEBUG: %s = %e\n", #A, (A))`

Typedefs

- `typedef struct` [ddenmat](#) [ddenmat](#)
- `typedef struct` [idenmat](#) [idenmat](#)
- `typedef struct` [dCSRmat](#) [dCSRmat](#)
- `typedef struct` [iCSRmat](#) [iCSRmat](#)
- `typedef struct` [dCOOmat](#) [dCOOmat](#)
- `typedef struct` [iCOOmat](#) [iCOOmat](#)
- `typedef struct` [dCSRLmat](#) [dCSRLmat](#)
- `typedef struct` [dSTRmat](#) [dSTRmat](#)
- `typedef struct` [dvector](#) [dvector](#)
- `typedef struct` [ivector](#) [ivector](#)

9.13.1 Detailed Description

Main header file for the FASP project.

Note

This header file contains general constants and data structures of FASP. It contains macros and data structure definitions; should not include function declarations here.

Copyright (C) 2008–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [fasp.h](#).

9.13.2 Macro Definition Documentation

9.13.2.1 `__FASP_HEADER__`

```
#define __FASP_HEADER__
```

indicate [fasp.h](#) has been included before
Definition at line 31 of file [fasp.h](#).

9.13.2.2 `ABS`

```
#define ABS(  
    a ) ((a)>=0.0)?(a):- (a))
```

absolute value of a
Definition at line 76 of file [fasp.h](#).

9.13.2.3 `DIAGONAL_PREF`

```
#define DIAGONAL_PREF OFF
```

order each row such that diagonal appears first
Definition at line 58 of file [fasp.h](#).

9.13.2.4 `DLMALLOC`

```
#define DLMALLOC OFF
```

For external software package support.
use dlmalloc instead of standard malloc
Definition at line 47 of file [fasp.h](#).

9.13.2.5 `FASP_VERSION`

```
#define FASP_VERSION 2.0
```

FASP base version information.
faspsolver version
Definition at line 40 of file [fasp.h](#).

9.13.2.6 `GE`

```
#define GE(  
    a,  
    b ) ((a)>=(b))?(TRUE):(FALSE))
```

is a >= b?
Definition at line 82 of file [fasp.h](#).

9.13.2.7 `GT`

```
#define GT(  
    a,  
    b ) ((a)>(b))?(TRUE):(FALSE))
```

Definition of >, >=, <, <=, and isnan.
is a > b?
Definition at line 81 of file [fasp.h](#).

9.13.2.8 INT

```
#define INT int
```

signed integer types: signed, long enough
Definition at line 64 of file [fasp.h](#).

9.13.2.9 ISNAN

```
#define ISNAN(  
    a ) ((a) != (a)) ? (TRUE) : (FALSE)
```

is a == NAN?
Definition at line 85 of file [fasp.h](#).

9.13.2.10 LE

```
#define LE(  
    a,  
    b ) ((a) <= (b)) ? (TRUE) : (FALSE)
```

is a <= b?
Definition at line 84 of file [fasp.h](#).

9.13.2.11 LONG

```
#define LONG long
```

long integer type
Definition at line 65 of file [fasp.h](#).

9.13.2.12 LONGLONG

```
#define LONGLONG long long
```

long long integer type
Definition at line 66 of file [fasp.h](#).

9.13.2.13 LONGREAL

```
#define LONGREAL long double
```

long double type
Definition at line 68 of file [fasp.h](#).

9.13.2.14 LS

```
#define LS(  
    a,  
    b ) ((a) < (b)) ? (TRUE) : (FALSE)
```

is a < b?
Definition at line 83 of file [fasp.h](#).

9.13.2.15 MAX

```
#define MAX(  
    a,  
    b ) ((a)>(b))?(a):(b)
```

Definition of max, min, abs.

bigger one in a and b

Definition at line 74 of file [fasp.h](#).

9.13.2.16 MIN

```
#define MIN(  
    a,  
    b ) ((a)<(b))?(a):(b)
```

smaller one in a and b

Definition at line 75 of file [fasp.h](#).

9.13.2.17 MULTI_COLOR_ORDER

```
#define MULTI_COLOR_ORDER OFF
```

Multicolor parallel GS smoothing method based on strongly connected matrix

Definition at line 42 of file [fasp.h](#).

9.13.2.18 NEDMALLOC

```
#define NEDMALLOC OFF
```

use nedmalloc instead of standard malloc

Definition at line 48 of file [fasp.h](#).

9.13.2.19 PUT_INT

```
#define PUT_INT(  
    A ) printf("### DEBUG: %s = %d\n", #A, (A))
```

Definition of print command in DEBUG mode.

print integer

Definition at line 90 of file [fasp.h](#).

9.13.2.20 PUT_REAL

```
#define PUT_REAL(  
    A ) printf("### DEBUG: %s = %e\n", #A, (A))
```

print real num

Definition at line 91 of file [fasp.h](#).

9.13.2.21 REAL

```
#define REAL double  
float type
```

Definition at line 67 of file [fasp.h](#).

9.13.2.22 RS_C1

```
#define RS_C1 ON
```

Flags for internal uses.

Warning

Change the following marcos with caution! CF splitting of RS: check C1 Criterion

Definition at line 56 of file [fasp.h](#).

9.13.2.23 SHORT

```
#define SHORT short
```

FASP integer and floating point numbers.

short integer type

Definition at line 63 of file [fasp.h](#).

9.13.2.24 STRLEN

```
#define STRLEN 256
```

length of strings

Definition at line 69 of file [fasp.h](#).

9.13.3 Typedef Documentation

9.13.3.1 dCOOmat

```
typedef struct dCOOmat dCOOmat
```

Sparse matrix of REAL type in COO format

9.13.3.2 dCSRLmat

```
typedef struct dCSRLmat dCSRLmat
```

Sparse matrix of REAL type in CSRL format

9.13.3.3 dCSRmat

```
typedef struct dCSRmat dCSRmat
```

Sparse matrix of REAL type in CSR format

9.13.3.4 ddenmat

```
typedef struct ddenmat ddenmat
```

Dense matrix of REAL type

9.13.3.5 dSTRmat

```
typedef struct dSTRmat dSTRmat
```

Structured matrix of REAL type

9.13.3.6 dvector

```
typedef struct dvector dvector
Vector of REAL type
```

9.13.3.7 iCOOmat

```
typedef struct iCOOmat iCOOmat
Sparse matrix of INT type in COO format
```

9.13.3.8 iCSRmat

```
typedef struct iCSRmat iCSRmat
Sparse matrix of INT type in CSR format
```

9.13.3.9 idenmat

```
typedef struct idenmat idenmat
Dense matrix of INT type
```

9.13.3.10 ivector

```
typedef struct ivector ivector
Vector of INT type
```

9.14 fasp.h

[Go to the documentation of this file.](#)

```
00001
00015 #include <stdio.h>
00016 #include <stdlib.h>
00017 #include <string.h>
00018
00019 #include "fasp_const.h"
00020
00021 #if WITH_MUMPS
00022 #include "dmumps_c.h"
00023 #endif
00024
00025 #if WITH_PARDISO
00026 #include "mkl_pardiso.h"
00027 #include "mkl_types.h"
00028 #endif
00029
00030 #ifndef __FASP_HEADER__          /*-- allow multiple inclusions --*/
00031 #define __FASP_HEADER__
00032 /*-----*/
00033 /*--- Macros definition ---*/
00034 /*-----*/
00035
00036
00040 #define FASP_VERSION      2.0
00042 #define MULTI_COLOR_ORDER OFF
00047 #define DLMALLOC          OFF
00048 #define NEDMALLOC         OFF
00055 // When this flag is OFF, do not force C1 criterion for the classical AMG method
00056 #define RS_C1              ON
00057 // When this flag is ON, the matrix rows will be reordered as diagonal entries first
00058 #define DIAGONAL_PREF     OFF
00063 #define SHORT              short
00064 #define INT                int
00065 #define LONG               long
00066 #define LONGLONG           long long
00067 #define REAL               double
00068 #define LONGREAL           long double
00069 #define STRLEN             256
00074 #define MAX(a,b)           ((a)>(b))?(a):(b)
00075 #define MIN(a,b)           ((a)<(b))?(a):(b)
00076 #define ABS(a)             ((a)>=0.0)?(a):- (a)
```

```

00081 #define GT(a,b)  (((a)>(b))?(TRUE):(FALSE))
00082 #define GE(a,b)  (((a)>=(b))?(TRUE):(FALSE))
00083 #define LS(a,b)  (((a)<(b))?(TRUE):(FALSE))
00084 #define LE(a,b)  (((a)<=(b))?(TRUE):(FALSE))
00085 #define ISNAN(a)  (((a)!=a))?(TRUE):(FALSE))
00090 #define PUT_INT(A)  printf("### DEBUG: %s = %d\n", #A, (A))
00091 #define PUT_REAL(A) printf("### DEBUG: %s = %e\n", #A, (A))
00093 /*-----*/
00094 /*--- Matrix and vector ---*/
00095 /*-----*/
00096
00103 typedef struct ddenmat{
00104
00106     INT row;
00107
00109     INT col;
00110
00112     REAL **val;
00113 } ddenmat;
00122 typedef struct idenmat{
00123
00125     INT row;
00126
00128     INT col;
00129
00131     INT **val;
00132 } idenmat;
00143 typedef struct dCSRmat{
00144
00146     INT row;
00147
00149     INT col;
00150
00152     INT nnz;
00153
00155     INT *IA;
00156
00158     INT *JA;
00159
00161     REAL *val;
00162
00163 #if MULTI_COLOR_ORDER
00165     INT color;
00167     INT *IC;
00169     INT *ICMAP;
00170 #endif
00171 } dCSRmat;
00182 typedef struct iCSRmat{
00183
00185     INT row;
00186
00188     INT col;
00189
00191     INT nnz;
00192
00194     INT *IA;
00195
00197     INT *JA;
00198
00200     INT *val;
00201 } iCSRmat;
00213 typedef struct dCOOmat{
00214
00216     INT row;
00217
00219     INT col;
00220
00222     INT nnz;
00223
00225     INT *rowind;
00226
00228     INT *colind;
00229
00231     REAL *val;
00232 } dCOOmat;
00243 typedef struct iCOOmat{

```

```

00244
00246     INT row;
00247
00249     INT col;
00250
00252     INT nnz;
00253
00255     INT *I;
00256
00258     INT *J;
00259
00261     INT *val;
00262
00263 } iCOOmat;
00269 typedef struct dCSRLmat{
00270
00272     INT row;
00273
00275     INT col;
00276
00278     INT nnz;
00279
00281     INT dif;
00282
00284     INT *nz_diff;
00285
00287     INT *index;
00288
00290     INT *start;
00291
00293     INT *ja;
00294
00296     REAL *val;
00297
00298 } dCSRLmat;
00308 typedef struct dSTRmat{
00309
00311     INT nx;
00312
00314     INT ny;
00315
00317     INT nz;
00318
00320     INT nxy;
00321
00323     INT nc;
00324
00326     INT ngrid;
00327
00329     REAL *diag;
00330
00332     INT nband;
00333
00335     INT *offsets;
00336
00338     REAL **offdiag;
00339
00340 } dSTRmat;
00346 typedef struct dvector{
00347
00349     INT row;
00350
00352     REAL *val;
00353
00354 } dvector;
00361 typedef struct ivector{
00362
00364     INT row;
00365
00367     INT *val;
00368
00369 } ivector;
00371 /*-----*/
00372 /*--- Parameter structures ---*/
00373 /*-----*/
00374
00379 typedef struct {
00380
00381     SHORT print_level;
00382     SHORT itsolver_type;
00383     SHORT decoup_type;

```

```
00384     SHORT precondition_type;
00385     SHORT stop_type;
00386     INT restart;
00387     INT maxit;
00388     REAL tol;
00390 } ITS_param;
00396 typedef struct {
00397
00399     SHORT print_level;
00400
00402     SHORT ILU_type;
00403
00405     INT ILU_lfil;
00406
00408     REAL ILU_droptol;
00409
00411     REAL ILU_relax;
00412
00414     REAL ILU_permtol;
00415
00416 } ILU_param;
00422 typedef struct {
00423
00425     SHORT print_level;
00426
00428     SHORT SWZ_type;
00429
00431     INT SWZ_maxlvl;
00432
00434     INT SWZ_mmsize;
00435
00437     INT SWZ_blksolver;
00438
00439 } SWZ_param;
00447 typedef struct {
00448
00450     SHORT AMG_type;
00451
00453     SHORT print_level;
00454
00456     INT maxit;
00457
00459     REAL tol;
00460
00462     SHORT max_levels;
00463
00465     INT coarse_dof;
00466
00468     SHORT cycle_type;
00469
00471     REAL quality_bound;
00472
00474     SHORT smoother;
00475
00477     SHORT smooth_order; // 1: nature order 2: C/F order (both are symmetric)
00478
00480     SHORT presmooth_iter;
00481
00483     SHORT postsmooth_iter;
00484
00486     REAL relaxation;
00487
00489     SHORT polynomial_degree;
00490
00492     SHORT coarse_solver;
00493
00495     SHORT coarse_scaling;
00496
00498     SHORT aml_i_degree;
00499
00501     REAL *aml_i_coef;
00502
00504     SHORT nl_aml_i_krylov_type;
00505
00507     SHORT coarsening_type;
00508
00510     SHORT aggregation_type;
00511
00513     SHORT interpolation_type;
00514
00516     REAL strong_threshold;
```

```

00517
00519     REAL max_row_sum;
00520
00522     REAL truncation_threshold;
00523
00525     INT aggressive_level;
00526
00528     INT aggressive_path;
00529
00531     INT pair_number;
00532
00534     REAL strong_coupled;
00535
00537     INT max_aggregation;
00538
00540     REAL tentative_smooth;
00541
00543     SHORT smooth_filter;
00544
00546     SHORT smooth_restriction;
00547
00549     SHORT ILU_levels;
00550
00552     SHORT ILU_type;
00553
00555     INT ILU_lfil;
00556
00558     REAL ILU_droptol;
00559
00561     REAL ILU_relax;
00562
00564     REAL ILU_permtol;
00565
00567     INT SWZ_levels;
00568
00570     INT SWZ_mmsize;
00571
00573     INT SWZ_maxlvl;
00574
00576     INT SWZ_type;
00577
00579     INT SWZ_blksolver;
00580
00581 } AMG_param;
00582 /*-----*/
00583 /*--- Work data structures ---*/
00584 /*-----*/
00585
00586
00593 typedef struct {
00594
00595     #if WITH_MUMPS
00597         DMUMPS_STRUC_C id;
00598     #endif
00599
00601     INT job;
00602
00603 } Mumps_data;
00611 typedef struct {
00612
00614     void *pt[64];
00615
00616     #if WITH_PARDISO
00618         MKL_INT iparm[64];
00619
00621         MKL_INT mtype;
00622
00624         MKL_INT maxfct;
00625
00627         MKL_INT mnum;
00628     #endif
00629
00630
00631 } Pardiso_data;
00637 typedef struct {
00638
00640     dCSRmat *A;
00641
00643     INT type;
00644
00646     INT row;
00647

```

```
00649     INT col;
00650
00652     INT nzlu;
00653
00655     INT *ijlu;
00656
00658     REAL *luval;
00659
00661     INT nb;
00662
00664     INT nwork;
00665
00667     REAL *work;
00668
00670     INT *iperm;
00671     // iperm[0:n-1] = old indices of unknowns
00672     // iperm[n:2*n-1] = reverse permutation = new indices.
00673
00675     INT ncolors;
00676
00678     INT *ic;
00679
00681     INT *icmap;
00682
00684     INT *uptr;
00685
00687     INT nlevL;
00688
00690     INT nlevU;
00691
00693     INT *ilevL;
00694
00696     INT *ilevU;
00697
00699     INT *jlevL;
00700
00702     INT *jlevU;
00703
00704 } ILU_data;
00712 typedef struct {
00713     /* matrix information */
00714     dCSRmat A; // note: must start from 1!! Change later
00715
00717     /* blocks information */
00718
00719     INT nblk;
00720
00722     INT *iblock;
00723
00725     INT *jblock;
00726
00728     REAL *rhsloc;
00731
00732     dvector rhsloc1;
00733
00734     dvector xloc1;
00735
00737     REAL *au;
00738
00740     REAL *al;
00741
00743     INT SWZ_type;
00744
00746     INT blk_solver;
00747
00749     INT memt;
00750
00752     INT *mask;
00753
00755     INT maxbs;
00756
00758     INT *maxa;
00759
00761     dCSRmat *blk_data;
00762
00764     #if WITH_UMFPACK
00765     void **numeric;
00766     #endif
00767
00769
00770
```

```
00771 #if WITH_MUMPS
00773     DMUMPS_STRUC_C *id;
00774 #endif
00775
00777     Mumps_data *mumps;
00778
00780     SWZ_param *swzparam;
00781
00782 } SWZ_data;
00790 typedef struct {
00791     /* Level information */
00793     SHORT max_levels;
00796     SHORT num_levels;
00799
00800     /* Problem information */
00801
00803     dCSRmat A;
00804
00806     dCSRmat R;
00807
00809     dCSRmat P;
00810
00812     dvector b;
00813
00815     dvector x;
00816
00817     /* Extra information */
00818
00820     void *Numeric;
00821
00823     Pardiso_data pdata;
00824
00826     ivector cfmark;
00827
00829     INT ILU_levels;
00830
00832     ILU_data LU;
00833
00835     INT near_kernel_dim;
00836
00838     REAL **near_kernel_basis;
00839
00840     // Smoother order information
00841
00843     INT SWZ_levels;
00844
00846     SWZ_data Schwarz;
00847
00849     dvector w;
00850
00852     Mumps_data mumps;
00853
00855     INT cycle_type;
00856
00858     INT *ic;
00859
00861     INT *icmap;
00862
00864     INT colors;
00865
00867     REAL weight;
00868
00869 #if MULTI_COLOR_ORDER
00871     REAL GS_Theta;
00872 #endif
00873
00874 } AMG_data;
00880 typedef struct {
00881
00883     SHORT AMG_type;
00884
00886     SHORT print_level;
00887
00889     INT maxit;
00890
00892     SHORT max_levels;
00893
00895     REAL tol;
```

```
00896
00898     SHORT cycle_type;
00899
00901     SHORT smoother;
00902
00904     SHORT smooth_order;
00905
00907     SHORT presmooth_iter;
00908
00910     SHORT postsmooth_iter;
00911
00913     REAL relaxation;
00914
00916     SHORT polynomial_degree;
00917
00919     SHORT coarsening_type;
00920
00922     SHORT coarse_solver;
00923
00925     SHORT coarse_scaling;
00926
00928     SHORT aml_i_degree;
00929
00931     SHORT nl_aml_i_krylov_type;
00932
00934     REAL tentative_smooth;
00935
00937     REAL *aml_i_coef;
00938
00940     AMG_data *mgl_data;
00941
00943     ILU_data *LU;
00944
00946     dCSRmat *A;
00947
00948     // extra near kernel space
00949
00951     dCSRmat *A_nk;
00952
00954     dCSRmat *P_nk;
00955
00957     dCSRmat *R_nk;
00958
00959     // temporary work space
00960
00962     dvector r;
00963
00965     REAL *w;
00966
00967 } precond_data;
00973 typedef struct {
00974
00976     SHORT AMG_type;
00977
00979     SHORT print_level;
00980
00982     INT maxit;
00983
00985     SHORT max_levels;
00986
00988     REAL tol;
00989
00991     SHORT cycle_type;
00992
00994     SHORT smoother;
00995
00997     SHORT presmooth_iter;
00998
01000     SHORT postsmooth_iter;
01001
01003     SHORT coarsening_type;
01004
01006     REAL relaxation;
01007
01009     SHORT coarse_scaling;
01010
01012     AMG_data *mgl_data;
01013
01015     ILU_data *LU;
01016
01018     SHORT scaled;
```



```

01019
01021     dCSRmat *A;
01022
01024     dSTRmat *A_str;
01025
01027     dSTRmat *SS_str;
01028
01029     // data for GS/block GS smoothers (STR format)
01030
01032     dvector *diainv;
01033
01035     ivector *pivot;
01036
01038     dvector *diainvS;
01039
01041     ivector *pivotS;
01042
01044     ivector *order;
01045
01047     ivector *neigh;
01048
01049     // temporary work space
01050
01052     dvector r;
01053
01055     REAL *w;
01056
01057 } precondition_data_str;
01065 typedef struct {
01066
01068     INT nc;
01069
01071     dvector diag;
01072
01073 } precondition_diag_str;
01081 typedef struct {
01082
01084     void *data;
01085
01087     void (*fct)(REAL *, REAL *, void *);
01088
01089 } precondition;
01095 typedef struct {
01096
01098     void *data;
01099
01101     void (*fct)(const void *, const REAL *, REAL *);
01102
01103 } mxv_matfree;
01111 typedef struct {
01112
01113     // output flags
01114     SHORT print_level;
01115     SHORT output_type;
01117     // problem parameters
01118     char inifile[STRLEN];
01119     char workdir[STRLEN];
01120     INT problem_num;
01122     // parameters for iterative solvers
01123     SHORT solver_type;
01124     SHORT decoup_type;
01125     SHORT precondition_type;
01126     SHORT stop_type;
01127     REAL itsolver_tol;
01128     INT itsolver_maxit;
01129     INT restart;
01131     // parameters for ILU
01132     SHORT ILU_type;
01133     INT ILU_lfil;
01134     REAL ILU_droptol;
01135     REAL ILU_relax;
01136     REAL ILU_permtol;
01138     // parameter for Schwarz
01139     INT SWZ_mmsize;
01140     INT SWZ_maxlvl;
01141     INT SWZ_type;
01142     INT SWZ_blksolver;
01144     // parameters for AMG
01145     SHORT AMG_type;
01146     SHORT AMG_levels;
01147     SHORT AMG_cycle_type;

```

```

01148     SHORT AMG_smoother;
01149     SHORT AMG_smooth_order;
01150     REAL AMG_relaxation;
01151     SHORT AMG_polynomial_degree;
01152     SHORT AMG_presmooth_iter;
01153     SHORT AMG_postsmooth_iter;
01154     REAL AMG_tol;
01155     INT AMG_coarse_dof;
01156     INT AMG_maxit;
01157     SHORT AMG_ILU_levels;
01158     SHORT AMG_coarse_solver;
01159     SHORT AMG_coarse_scaling;
01160     SHORT AMG_amli_degree;
01161     SHORT AMG_nl_amli_krylov_type;
01162     INT AMG_SWZ_levels;
01164     // parameters for classical AMG
01165     SHORT AMG_coarsening_type;
01166     SHORT AMG_aggregation_type;
01167     SHORT AMG_interpolation_type;
01168     REAL AMG_strong_threshold;
01169     REAL AMG_truncation_threshold;
01170     REAL AMG_max_row_sum;
01171     INT AMG_aggressive_level;
01172     INT AMG_aggressive_path;
01173     INT AMG_pair_number;
01174     REAL AMG_quality_bound;
01176     // parameters for smoothed aggregation AMG
01177     REAL AMG_strong_coupled;
01178     INT AMG_max_aggregation;
01179     REAL AMG_tentative_smooth;
01180     SHORT AMG_smooth_filter;
01181     SHORT AMG_smooth_restriction;
01183 } input_param;
01185 /*
01186  * OpenMP definitions and declarations
01187  */
01188 #ifdef _OPENMP
01189
01190 #include "omp.h"
01191
01192 #define ILU_MC_OMP      OFF
01194 //extern INT omp_count;    /**< Counter for multiple calls:  Remove later!!!  --Chensong */
01195
01196 extern INT THDs_AMG_GS;
01197 extern INT THDs_CPR_lGS;
01198 extern INT THDs_CPR_gGS;
01199 #ifdef DETAILTIME
01200 extern REAL total_linear_time;
01201 extern REAL total_start_time;
01202 extern REAL total_setup_time;
01203 extern INT total_iter;
01204 extern INT fasp_called_times;
01205 #endif
01206
01207 #endif /* end if for _OPENMP */
01208
01209 #endif /* end if for __FASP_HEADER__ */
01210
01211 /*-----*/
01212 /*--      End of File      --*/
01213 /*-----*/

```

9.15 fasp_block.h File Reference

Header file for FASP block matrices.

```
#include "fasp.h"
```

Data Structures

- struct [dBSRmat](#)
Block sparse row storage matrix of REAL type.
- struct [dBLCmat](#)

- *Block REAL CSR matrix format.*
struct [iBLCmat](#)
- *Block INT CSR matrix format.*
struct [block_dvector](#)
- *Block REAL vector structure.*
struct [block_ivector](#)
- *Block INT vector structure.*
struct [AMG_data_bsr](#)
- *Data for multigrid levels in dBSRmat format.*
struct [precond_diag_bsr](#)
- *Data for diagonal preconditioners in dBSRmat format.*
struct [precond_data_bsr](#)
- *Data for preconditioners in dBSRmat format.*
struct [precond_data_blc](#)
- *Data for block preconditioners in dBLCmat format.*
struct [precond_data_sweeping](#)
- *Data for sweeping preconditioner.*

Macros

- #define [__FASPBLOCK_HEADER__](#)

Typedefs

- typedef struct [dBSRmat](#) dBSRmat
- typedef struct [dBLCmat](#) dBLCmat
- typedef struct [iBLCmat](#) iBLCmat
- typedef struct [block_dvector](#) block_dvector
- typedef struct [block_ivector](#) block_ivector

9.15.1 Detailed Description

Header file for FASP block matrices.

Note

This header file contains definitions of block matrices, including grid-major type and variable-major type. In this header, we only define macros and data structures, not function declarations.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [fasp_block.h](#).

9.15.2 Macro Definition Documentation

9.15.2.1 __FASPBLOCK_HEADER__

#define [__FASPBLOCK_HEADER__](#)

indicate [fasp_block.h](#) has been included before

Definition at line 18 of file [fasp_block.h](#).

9.15.3 Typedef Documentation

9.15.3.1 block_dvector

typedef struct [block_dvector](#) [block_dvector](#)
Vector of REAL type in Block format

9.15.3.2 block_ivector

typedef struct [block_ivector](#) [block_ivector](#)
Vector of INT type in Block format

9.15.3.3 dBLCmat

typedef struct [dBLCmat](#) [dBLCmat](#)
Matrix of REAL type in Block CSR format

9.15.3.4 dBSRmat

typedef struct [dBSRmat](#) [dBSRmat](#)
Matrix of REAL type in BSR format

9.15.3.5 iBLCmat

typedef struct [iBLCmat](#) [iBLCmat](#)
Matrix of INT type in Block CSR format

9.16 fasp_block.h

[Go to the documentation of this file.](#)

```

00001
00015 #include "fasp.h"
00016
00017 #ifndef __FASPBLOCK_HEADER__          /*-- allow multiple inclusions --*/
00018 #define __FASPBLOCK_HEADER__
00020 /*-----*/
00021 /*--- Data structures ---*/
00022 /*-----*/
00023
00034 typedef struct dBSRmat {
00035
00037     INT ROW;
00038
00040     INT COL;
00041
00043     INT NNZ;
00044
00046     INT nb; // NOTE: for the moment, allow nb*nb full block
00047
00049     INT storage_manner; // 0: row-major order, 1: column-major order
00050
00057     REAL *val;
00058
00060     INT *IA;
00061
00064     INT *JA;
00065
00066 } dBSRmat;
00074 typedef struct dBLCmat {
00075
00077     INT brow;
00078
00080     INT bcol;
00081

```

```

00083     dCSRmat **blocks;
00084
00085 } dBLCMat;
00093 typedef struct iBLCMat {
00094
00096     INT brow;
00097
00099     INT bcol;
00100
00102     iCSRmat **blocks;
00103
00104 } iBLCMat;
00110 typedef struct block_dvector {
00111
00113     INT brow;
00114
00116     dvector **blocks;
00117
00118 } block_dvector;
00126 typedef struct block_ivector {
00127
00129     INT brow;
00130
00132     ivector **blocks;
00133
00134 } block_ivector;
00136 /*-----*/
00137 /*--- Parameter structures ---*/
00138 /*-----*/
00139
00146 typedef struct {
00147
00149     INT max_levels;
00150
00152     INT num_levels;
00153
00155     dBSRmat A;
00156
00158     dBSRmat R;
00159
00161     dBSRmat P;
00162
00164     dvector b;
00165
00167     dvector x;
00168
00170     dvector diaginv;
00171
00173     dCSRmat Ac;
00174
00176     void *Numeric;
00177
00179     Pardiso_data pdata;
00180
00182     dCSRmat PP;
00183
00185     REAL *pw;
00186
00188     dBSRmat SS;
00189
00191     REAL *sw;
00192
00194     dvector diaginv_SS;
00195
00197     ILU_data PP_LU;
00198
00200     ivector cfmark;
00201
00203     INT ILU_levels;
00204
00206     ILU_data LU;
00207
00209     INT near_kernel_dim;
00210
00212     REAL **near_kernel_basis;
00213
00214     //-----
00215     // extra near kernal space for extra solve
00216
00218     dCSRmat *A_nk;
00219

```

```

00221     dCSRmat *P_nk;
00222
00224     dCSRmat *R_nk;
00225     //-----
00226
00228     dvector w;
00229
00231     Mumps_data mumps;
00232
00233 } AMG_data_bsr;
00241 typedef struct {
00242
00244     INT nb;
00245
00247     dvector diag;
00248
00249 } precond_diag_bsr;
00257 typedef struct {
00258
00260     SHORT AMG_type;
00261
00263     SHORT print_level;
00264
00266     INT maxit;
00267
00269     INT max_levels;
00270
00272     REAL tol;
00273
00275     SHORT cycle_type;
00276
00278     SHORT smoother;
00279
00281     SHORT smooth_order;
00282
00284     SHORT presmooth_iter;
00285
00287     SHORT postsmooth_iter;
00288
00290     SHORT coarsening_type;
00291
00293     REAL relaxation;
00294
00296     SHORT coarse_solver;
00297
00299     SHORT coarse_scaling;
00300
00302     SHORT aml_i_degree;
00303
00305     REAL *aml_i_coef;
00306
00308     REAL tentative_smooth;
00309
00311     SHORT nl_aml_i_krylov_type;
00312
00314     AMG_data_bsr *mgl_data;
00315
00317     AMG_data *pres_mgl_data;
00318
00320     ILU_data *LU;
00321
00323     dBSRmat *A;
00324
00325     // extra near kernal space
00326
00328     dCSRmat *A_nk;
00329
00331     dCSRmat *P_nk;
00332
00334     dCSRmat *R_nk;
00335
00337     dvector r;
00338
00340     REAL *w;
00341
00342 } precond_data_bsr;
00349 typedef struct {
00350
00351     /*-----*/
00352     /* Basic data for block preconditioner */
00353     /*-----*/

```

```

00354     dBLCmat *Ablc;
00356     dCSRmat *A_diag;
00358     dvector r;
00360     /*-----*/
00361     /* Data for the diagonal blocks */
00362     /*-----*/
00363
00364     /*--- solve by direct solver ---*/
00365     void **LU_diag;
00367     /*--- solve by AMG ---*/
00368     AMG_data **mgl;
00370     AMG_param *amgparam;
00372 } precondition_data_blc;
00384 typedef struct {
00385
00386     INT NumLayers;
00388     dBLCmat *A;
00389     dBLCmat *Ai;
00391     dCSRmat *local_A;
00392     void **local_LU;
00394     ivector *local_index;
00396     // temporary work spaces
00397     dvector r;
00398     REAL *w;
00400 } precondition_data_sweeping;
00402 #endif /* end if for __FASPBLOCK_HEADER__ */
00403
00404 /*-----*/
00405 /*--          End of File          --*/
00406 /*-----*/

```

9.17 fasp_const.h File Reference

Definition of FASP constants, including messages, solver types, etc.

Macros

- #define `FASP_SUCCESS` 0
Definition of return status and error messages.
- #define `ERROR_READ_FILE` -1
- #define `ERROR_OPEN_FILE` -10
- #define `ERROR_WRONG_FILE` -11
- #define `ERROR_INPUT_PAR` -13
- #define `ERROR_REGRESS` -14
- #define `ERROR_MAT_SIZE` -15
- #define `ERROR_NUM_BLOCKS` -18
- #define `ERROR_MISC` -19
- #define `ERROR_ALLOC_MEM` -20
- #define `ERROR_DATA_STRUCTURE` -21
- #define `ERROR_DATA_ZERODIAG` -22
- #define `ERROR_DUMMY_VAR` -23
- #define `ERROR_AMG_INTERP_TYPE` -30
- #define `ERROR_AMG_SMOOTH_TYPE` -31
- #define `ERROR_AMG_COARSE_TYPE` -32
- #define `ERROR_AMG_COARSEING` -33
- #define `ERROR_AMG_SETUP` -39
- #define `ERROR_SOLVER_TYPE` -40
- #define `ERROR_SOLVER_PRECTYPE` -41
- #define `ERROR_SOLVER_STAG` -42
- #define `ERROR_SOLVER_SOLSTAG` -43
- #define `ERROR_SOLVER_TOLSMALL` -44

- #define `ERROR_SOLVER_ILUSETUP` -45
- #define `ERROR_SOLVER_MISC` -46
- #define `ERROR_SOLVER_MAXIT` -48
- #define `ERROR_SOLVER_EXIT` -49
- #define `ERROR_QUAD_TYPE` -60
- #define `ERROR_QUAD_DIM` -61
- #define `ERROR_LIC_TYPE` -80
- #define `ERROR_UNKNOWN` -99
- #define `TRUE` 1
- *Definition of logic type.*
- #define `FALSE` 0
- #define `ON` 1
- *Definition of switch.*
- #define `OFF` 0
- #define `PRINT_NONE` 0
- *Print level for all subroutines – not including `DEBUG` output.*
- #define `PRINT_MIN` 1
- #define `PRINT_SOME` 2
- #define `PRINT_MORE` 4
- #define `PRINT_MOST` 8
- #define `PRINT_ALL` 10
- #define `MAT_FREE` 0
- *Definition of matrix format.*
- #define `MAT_CSR` 1
- #define `MAT_BSR` 2
- #define `MAT_STR` 3
- #define `MAT_CSRL` 6
- #define `MAT_SymCSR` 7
- #define `MAT_BLC` 8
- #define `MAT_bCSR` 11
- #define `MAT_bBSR` 12
- #define `MAT_bSTR` 13
- #define `SOLVER_DEFAULT` 0
- *Definition of solver types for iterative methods.*
- #define `SOLVER_CG` 1
- #define `SOLVER_BiCGstab` 2
- #define `SOLVER_MinRes` 3
- #define `SOLVER_GMRES` 4
- #define `SOLVER_VGMRES` 5
- #define `SOLVER_VFGMRES` 6
- #define `SOLVER_GCG` 7
- #define `SOLVER_GCR` 8
- #define `SOLVER_SCG` 11
- #define `SOLVER_SBiCGstab` 12
- #define `SOLVER_SMinRes` 13
- #define `SOLVER_SGMRES` 14
- #define `SOLVER_SVGMRES` 15
- #define `SOLVER_SVFGMRES` 16
- #define `SOLVER_SGCG` 17
- #define `SOLVER_AMG` 21

- #define SOLVER_FMG 22
- #define SOLVER_SUPERLU 31
- #define SOLVER_UMFPACK 32
- #define SOLVER_MUMPS 33
- #define SOLVER_PARDISO 34
- #define STOP_REL_RES 1

Definition of iterative solver stopping criteria types.

- #define STOP_REL_PRECRES 2
- #define STOP_MOD_REL_RES 3
- #define PREC_NULL 0

Definition of preconditioner type for iterative methods.

- #define PREC_DIAG 1
- #define PREC_AMG 2
- #define PREC_FMG 3
- #define PREC_ILU 4
- #define PREC_SCHWARZ 5
- #define ILUk 1

Type of ILU methods.

- #define ILU_t 2
- #define ILU_{tp} 3
- #define SCHWARZ_FORWARD 1

Type of Schwarz smoother.

- #define SCHWARZ_BACKWARD 2
- #define SCHWARZ_SYMMETRIC 3
- #define CLASSIC_AMG 1

Definition of AMG types.

- #define SA_AMG 2
- #define UA_AMG 3
- #define PAIRWISE 1

Definition of aggregation types.

- #define VMB 2
- #define NPAIR 3
- #define SPAIR 4
- #define V_CYCLE 1

Definition of cycle types.

- #define W_CYCLE 2
- #define AMLI_CYCLE 3
- #define NL_AMLI_CYCLE 4
- #define VW_CYCLE 12
- #define WV_CYCLE 21
- #define SMOOTHER_JACOBI 1

Definition of standard smoother types.

- #define SMOOTHER_GS 2
- #define SMOOTHER_SGS 3
- #define SMOOTHER_CG 4
- #define SMOOTHER_SOR 5
- #define SMOOTHER_SSOR 6
- #define SMOOTHER_GSOR 7
- #define SMOOTHER_SGSOR 8

- #define SMOOTHER_POLY 9
- #define SMOOTHER_L1DIAG 10
- #define SMOOTHER_BLKOIL 11
- Definition of specialized smoother types.*
- #define SMOOTHER_SPETEN 19
- #define COARSE_RS 1
- Definition of coarsening types.*
- #define COARSE_RSP 2
- #define COARSE_CR 3
- #define COARSE_AC 4
- #define COARSE_MIS 5
- #define INTERP_DIR 1
- Definition of interpolation types.*
- #define INTERP_STD 2
- #define INTERP_ENG 3
- #define INTERP_EXT 6
- #define G0PT -5
- Type of vertices (DOFs) for coarsening.*
- #define UNPT -1
- #define FGPT 0
- #define CGPT 1
- #define ISPT 2
- #define NO_ORDER 0
- Definition of smoothing order.*
- #define CF_ORDER 1
- #define USERDEFINED 0
- Type of ordering for smoothers.*
- #define CPFIRST 1
- #define FPFIRST -1
- #define ASCEND 12
- #define DESCEND 21
- #define BIGREAL 1e+20
- Some global constants.*
- #define SMALLREAL 1e-20
- #define SMALLREAL2 1e-40
- #define MAX_REFINE_LVL 20
- #define MAX_AMG_LVL 20
- #define MIN_CDOF 20
- #define MIN_CRATE 0.9
- #define MAX_CRATE 20.0
- #define MAX_RESTART 20
- #define MAX_STAG 20
- #define STAG_RATIO 1e-4
- #define FPNA_RATIO 1e-8
- #define OPENMP_HOLDS 2000

9.17.1 Detailed Description

Definition of FASP constants, including messages, solver types, etc.
 Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Warning

This is for internal use only. Do NOT change!

Definition in file [fasp_const.h](#).

9.17.2 Macro Definition Documentation

9.17.2.1 AMLI_CYCLE

```
#define AMLI_CYCLE 3
```

AMLI-cycle

Definition at line 179 of file [fasp_const.h](#).

9.17.2.2 ASCEND

```
#define ASCEND 12
```

Ascending order

Definition at line 242 of file [fasp_const.h](#).

9.17.2.3 BIGREAL

```
#define BIGREAL 1e+20
```

Some global constants.

A large real number

Definition at line 248 of file [fasp_const.h](#).

9.17.2.4 CF_ORDER

```
#define CF_ORDER 1
```

C/F order smoothing

Definition at line 234 of file [fasp_const.h](#).

9.17.2.5 CGPT

```
#define CGPT 1
```

Coarse grid points

Definition at line 227 of file [fasp_const.h](#).

9.17.2.6 CLASSIC_AMG

```
#define CLASSIC_AMG 1
```

Definition of AMG types.

classic AMG

Definition at line 162 of file [fasp_const.h](#).

9.17.2.7 COARSE_AC

`#define COARSE_AC 4`
Aggressive coarsening
Definition at line 210 of file [fasp_const.h](#).

9.17.2.8 COARSE_CR

`#define COARSE_CR 3`
Compatible relaxation
Definition at line 209 of file [fasp_const.h](#).

9.17.2.9 COARSE_MIS

`#define COARSE_MIS 5`
Aggressive coarsening based on MIS
Definition at line 211 of file [fasp_const.h](#).

9.17.2.10 COARSE_RS

`#define COARSE_RS 1`
Definition of coarsening types.
Classical
Definition at line 207 of file [fasp_const.h](#).

9.17.2.11 COARSE_RSP

`#define COARSE_RSP 2`
Classical, with positive offdiags
Definition at line 208 of file [fasp_const.h](#).

9.17.2.12 CPFIRST

`#define CPFIRST 1`
C-points first order
Definition at line 240 of file [fasp_const.h](#).

9.17.2.13 DESCEND

`#define DESCEND 21`
Descending order
Definition at line 243 of file [fasp_const.h](#).

9.17.2.14 ERROR_ALLOC_MEM

`#define ERROR_ALLOC_MEM -20`
fail to allocate memory
Definition at line 30 of file [fasp_const.h](#).

9.17.2.15 ERROR_AMG_COARSE_TYPE

```
#define ERROR_AMG_COARSE_TYPE -32
```

unknown coarsening type
Definition at line 37 of file [fasp_const.h](#).

9.17.2.16 ERROR_AMG_COARSEING

```
#define ERROR_AMG_COARSEING -33
```

coarsening step failed to complete
Definition at line 38 of file [fasp_const.h](#).

9.17.2.17 ERROR_AMG_INTERP_TYPE

```
#define ERROR_AMG_INTERP_TYPE -30
```

unknown interpolation type
Definition at line 35 of file [fasp_const.h](#).

9.17.2.18 ERROR_AMG_SETUP

```
#define ERROR_AMG_SETUP -39
```

AMG setup failed to complete
Definition at line 39 of file [fasp_const.h](#).

9.17.2.19 ERROR_AMG_SMOOTH_TYPE

```
#define ERROR_AMG_SMOOTH_TYPE -31
```

unknown smoother type
Definition at line 36 of file [fasp_const.h](#).

9.17.2.20 ERROR_DATA_STRUCTURE

```
#define ERROR_DATA_STRUCTURE -21
```

problem with data structures
Definition at line 31 of file [fasp_const.h](#).

9.17.2.21 ERROR_DATA_ZERODIAG

```
#define ERROR_DATA_ZERODIAG -22
```

matrix has zero diagonal entries
Definition at line 32 of file [fasp_const.h](#).

9.17.2.22 ERROR_DUMMY_VAR

```
#define ERROR_DUMMY_VAR -23
```

unexpected input data
Definition at line 33 of file [fasp_const.h](#).

9.17.2.23 ERROR_INPUT_PAR

```
#define ERROR_INPUT_PAR -13
```

wrong input argument
Definition at line 24 of file [fasp_const.h](#).

9.17.2.24 ERROR_LIC_TYPE

```
#define ERROR_LIC_TYPE -80
```

wrong license type
Definition at line 54 of file [fasp_const.h](#).

9.17.2.25 ERROR_MAT_SIZE

```
#define ERROR_MAT_SIZE -15
```

wrong problem size
Definition at line 26 of file [fasp_const.h](#).

9.17.2.26 ERROR_MISC

```
#define ERROR_MISC -19
```

other error
Definition at line 28 of file [fasp_const.h](#).

9.17.2.27 ERROR_NUM_BLOCKS

```
#define ERROR_NUM_BLOCKS -18
```

wrong number of blocks
Definition at line 27 of file [fasp_const.h](#).

9.17.2.28 ERROR_OPEN_FILE

```
#define ERROR_OPEN_FILE -10
```

fail to open a file
Definition at line 22 of file [fasp_const.h](#).

9.17.2.29 ERROR_QUAD_DIM

```
#define ERROR_QUAD_DIM -61
```

unsupported quadrature dim
Definition at line 52 of file [fasp_const.h](#).

9.17.2.30 ERROR_QUAD_TYPE

```
#define ERROR_QUAD_TYPE -60
```

unknown quadrature type
Definition at line 51 of file [fasp_const.h](#).

9.17.2.31 ERROR_READ_FILE

```
#define ERROR_READ_FILE -1
```

fail to read a file
Definition at line 21 of file [fasp_const.h](#).

9.17.2.32 ERROR_REGRESS

```
#define ERROR_REGRESS -14
```

regression test fail
Definition at line 25 of file [fasp_const.h](#).

9.17.2.33 ERROR_SOLVER_EXIT

```
#define ERROR_SOLVER_EXIT -49
```

solver does not quit successfully
Definition at line 49 of file [fasp_const.h](#).

9.17.2.34 ERROR_SOLVER_ILUSETUP

```
#define ERROR_SOLVER_ILUSETUP -45
```

ILU setup error
Definition at line 46 of file [fasp_const.h](#).

9.17.2.35 ERROR_SOLVER_MAXIT

```
#define ERROR_SOLVER_MAXIT -48
```

maximal iteration number exceeded
Definition at line 48 of file [fasp_const.h](#).

9.17.2.36 ERROR_SOLVER_MISC

```
#define ERROR_SOLVER_MISC -46
```

misc solver error during run time
Definition at line 47 of file [fasp_const.h](#).

9.17.2.37 ERROR_SOLVER_PRECTYPE

```
#define ERROR_SOLVER_PRECTYPE -41
```

unknown precondition type
Definition at line 42 of file [fasp_const.h](#).

9.17.2.38 ERROR_SOLVER_SOLSTAG

```
#define ERROR_SOLVER_SOLSTAG -43
```

solver's solution is too small
Definition at line 44 of file [fasp_const.h](#).

9.17.2.39 ERROR_SOLVER_STAG

```
#define ERROR_SOLVER_STAG -42
```

solver stagnates
Definition at line 43 of file [fasp_const.h](#).

9.17.2.40 ERROR_SOLVER_TOLSMALL

```
#define ERROR_SOLVER_TOLSMALL -44
```

solver's tolerance is too small
Definition at line 45 of file [fasp_const.h](#).

9.17.2.41 ERROR_SOLVER_TYPE

```
#define ERROR_SOLVER_TYPE -40
```

unknown solver type
Definition at line 41 of file [fasp_const.h](#).

9.17.2.42 ERROR_UNKNOWN

```
#define ERROR_UNKNOWN -99
```

an unknown error type
Definition at line 56 of file [fasp_const.h](#).

9.17.2.43 ERROR_WRONG_FILE

```
#define ERROR_WRONG_FILE -11
```

input contains wrong format
Definition at line 23 of file [fasp_const.h](#).

9.17.2.44 FALSE

```
#define FALSE 0
```

logic FALSE
Definition at line 62 of file [fasp_const.h](#).

9.17.2.45 FASP_SUCCESS

```
#define FASP_SUCCESS 0
```

Definition of return status and error messages.
return from function successfully
Definition at line 19 of file [fasp_const.h](#).

9.17.2.46 FGPT

```
#define FGPT 0
```

Fine grid points

Definition at line 226 of file [fasp_const.h](#).

9.17.2.47 FPFIRST

```
#define FPFIRST -1
```

F-points first order

Definition at line 241 of file [fasp_const.h](#).

9.17.2.48 FPNA_RATIO

```
#define FPNA_RATIO 1e-8
```

Float-point number arithmetic threshold = tol*FPNA_RATIO

Definition at line 259 of file [fasp_const.h](#).

9.17.2.49 G0PT

```
#define G0PT -5
```

Type of vertices (DOFs) for coarsening.

Cannot fit in aggregates

Definition at line 224 of file [fasp_const.h](#).

9.17.2.50 ILUk

```
#define ILUk 1
```

Type of ILU methods.

ILUk

Definition at line 148 of file [fasp_const.h](#).

9.17.2.51 ILUt

```
#define ILUt 2
```

ILUt

Definition at line 149 of file [fasp_const.h](#).

9.17.2.52 ILUtp

```
#define ILUtp 3
```

ILUtp

Definition at line 150 of file [fasp_const.h](#).

9.17.2.53 INTERP_DIR

```
#define INTERP_DIR 1
```

Definition of interpolation types.

Direct interpolation

Definition at line 216 of file [fasp_const.h](#).

9.17.2.54 INTERP_ENG

`#define INTERP_ENG 3`
Energy minimization interpolation
Definition at line 218 of file [fasp_const.h](#).

9.17.2.55 INTERP_EXT

`#define INTERP_EXT 6`
Extended interpolation
Definition at line 219 of file [fasp_const.h](#).

9.17.2.56 INTERP_STD

`#define INTERP_STD 2`
Standard interpolation
Definition at line 217 of file [fasp_const.h](#).

9.17.2.57 ISPT

`#define ISPT 2`
Isolated points
Definition at line 228 of file [fasp_const.h](#).

9.17.2.58 MAT_bBSR

`#define MAT_bBSR 12`
block BSR/CSR matrix
Definition at line 95 of file [fasp_const.h](#).

9.17.2.59 MAT_bCSR

`#define MAT_bCSR 11`
block CSR/CSR matrix == 2*2 BLC matrix
Definition at line 94 of file [fasp_const.h](#).

9.17.2.60 MAT_BLC

`#define MAT_BLC 8`
block CSR matrix
Definition at line 90 of file [fasp_const.h](#).

9.17.2.61 MAT_BSR

`#define MAT_BSR 2`
block-wise compressed sparse row
Definition at line 86 of file [fasp_const.h](#).

9.17.2.62 MAT_bSTR

`#define MAT_bSTR 13`
block STR/CSR matrix
Definition at line 96 of file [fasp_const.h](#).

9.17.2.63 MAT_CSR

`#define MAT_CSR 1`
compressed sparse row
Definition at line 85 of file [fasp_const.h](#).

9.17.2.64 MAT_CSRL

`#define MAT_CSRL 6`
modified CSR to reduce cache missing
Definition at line 88 of file [fasp_const.h](#).

9.17.2.65 MAT_FREE

`#define MAT_FREE 0`
Definition of matrix format.
matrix-free format: only mxv action
Definition at line 83 of file [fasp_const.h](#).

9.17.2.66 MAT_STR

`#define MAT_STR 3`
structured sparse matrix
Definition at line 87 of file [fasp_const.h](#).

9.17.2.67 MAT_SymCSR

`#define MAT_SymCSR 7`
symmetric CSR format
Definition at line 89 of file [fasp_const.h](#).

9.17.2.68 MAX_AMG_LVL

`#define MAX_AMG_LVL 20`
Maximal AMG coarsening level
Definition at line 252 of file [fasp_const.h](#).

9.17.2.69 MAX_CRATE

`#define MAX_CRATE 20.0`
Maximal coarsening ratio
Definition at line 255 of file [fasp_const.h](#).

9.17.2.70 MAX_REFINE_LVL

```
#define MAX_REFINE_LVL 20
```

Maximal refinement level

Definition at line 251 of file [fasp_const.h](#).

9.17.2.71 MAX_RESTART

```
#define MAX_RESTART 20
```

Maximal restarting number

Definition at line 256 of file [fasp_const.h](#).

9.17.2.72 MAX_STAG

```
#define MAX_STAG 20
```

Maximal number of stagnation times

Definition at line 257 of file [fasp_const.h](#).

9.17.2.73 MIN_CDOF

```
#define MIN_CDOF 20
```

Minimal number of coarsest variables

Definition at line 253 of file [fasp_const.h](#).

9.17.2.74 MIN_CRATE

```
#define MIN_CRATE 0.9
```

Minimal coarsening ratio

Definition at line 254 of file [fasp_const.h](#).

9.17.2.75 NL_AMLI_CYCLE

```
#define NL_AMLI_CYCLE 4
```

Nonlinear AMLI-cycle

Definition at line 180 of file [fasp_const.h](#).

9.17.2.76 NO_ORDER

```
#define NO_ORDER 0
```

Definition of smoothing order.

Natural order smoothing

Definition at line 233 of file [fasp_const.h](#).

9.17.2.77 NPAIR

```
#define NPAIR 3
```

non-symmetric pairwise aggregation

Definition at line 171 of file [fasp_const.h](#).

9.17.2.78 OFF

```
#define OFF 0
```

turn off certain parameter
Definition at line 68 of file [fasp_const.h](#).

9.17.2.79 ON

```
#define ON 1
```

Definition of switch.
turn on certain parameter
Definition at line 67 of file [fasp_const.h](#).

9.17.2.80 OPENMP_HOLDS

```
#define OPENMP_HOLDS 2000
```

Smallest size for OpenMP version
Definition at line 260 of file [fasp_const.h](#).

9.17.2.81 PAIRWISE

```
#define PAIRWISE 1
```

Definition of aggregation types.
pairwise aggregation, default is SPAIR
Definition at line 169 of file [fasp_const.h](#).

9.17.2.82 PREC_AMG

```
#define PREC_AMG 2
```

with AMG precondition
Definition at line 140 of file [fasp_const.h](#).

9.17.2.83 PREC_DIAG

```
#define PREC_DIAG 1
```

with diagonal precondition
Definition at line 139 of file [fasp_const.h](#).

9.17.2.84 PREC_FMG

```
#define PREC_FMG 3
```

with full AMG precondition
Definition at line 141 of file [fasp_const.h](#).

9.17.2.85 PREC_ILU

```
#define PREC_ILU 4
```

with ILU precondition
Definition at line 142 of file [fasp_const.h](#).

9.17.2.86 PREC_NULL

```
#define PREC_NULL 0
```

Definition of preconditioner type for iterative methods.
with no precondition

Definition at line 138 of file [fasp_const.h](#).

9.17.2.87 PREC_SCHWARZ

```
#define PREC_SCHWARZ 5
```

with Schwarz preconditioner

Definition at line 143 of file [fasp_const.h](#).

9.17.2.88 PRINT_ALL

```
#define PRINT_ALL 10
```

all: all printouts, including files

Definition at line 78 of file [fasp_const.h](#).

9.17.2.89 PRINT_MIN

```
#define PRINT_MIN 1
```

quiet: print error, important warnings

Definition at line 74 of file [fasp_const.h](#).

9.17.2.90 PRINT_MORE

```
#define PRINT_MORE 4
```

more: print some useful debug info

Definition at line 76 of file [fasp_const.h](#).

9.17.2.91 PRINT_MOST

```
#define PRINT_MOST 8
```

most: maximal printouts, no files

Definition at line 77 of file [fasp_const.h](#).

9.17.2.92 PRINT_NONE

```
#define PRINT_NONE 0
```

Print level for all subroutines – not including DEBUG output.

silent: no printout at all

Definition at line 73 of file [fasp_const.h](#).

9.17.2.93 PRINT_SOME

```
#define PRINT_SOME 2
```

some: print less important warnings
Definition at line 75 of file [fasp_const.h](#).

9.17.2.94 SA_AMG

```
#define SA_AMG 2
```

smoothed aggregation AMG
Definition at line 163 of file [fasp_const.h](#).

9.17.2.95 SCHWARZ_BACKWARD

```
#define SCHWARZ_BACKWARD 2
```

Backward ordering
Definition at line 156 of file [fasp_const.h](#).

9.17.2.96 SCHWARZ_FORWARD

```
#define SCHWARZ_FORWARD 1
```

Type of Schwarz smoother.
Forward ordering
Definition at line 155 of file [fasp_const.h](#).

9.17.2.97 SCHWARZ_SYMMETRIC

```
#define SCHWARZ_SYMMETRIC 3
```

Symmetric smoother
Definition at line 157 of file [fasp_const.h](#).

9.17.2.98 SMALLREAL

```
#define SMALLREAL 1e-20
```

A small real number
Definition at line 249 of file [fasp_const.h](#).

9.17.2.99 SMALLREAL2

```
#define SMALLREAL2 1e-40
```

An extremely small real number
Definition at line 250 of file [fasp_const.h](#).

9.17.2.100 SMOOTHER_BLKOIL

```
#define SMOOTHER_BLKOIL 11
```

Definition of specialized smoother types.
Used in monolithic AMG for black-oil
Definition at line 201 of file [fasp_const.h](#).

9.17.2.101 SMOOTHER_CG

`#define SMOOTHER_CG 4`
CG as a smoother
Definition at line 190 of file [fasp_const.h](#).

9.17.2.102 SMOOTHER_GS

`#define SMOOTHER_GS 2`
Gauss-Seidel smoother
Definition at line 188 of file [fasp_const.h](#).

9.17.2.103 SMOOTHER_GSOR

`#define SMOOTHER_GSOR 7`
GS + SOR smoother
Definition at line 193 of file [fasp_const.h](#).

9.17.2.104 SMOOTHER_JACOBI

`#define SMOOTHER_JACOBI 1`
Definition of standard smoother types.
Jacobi smoother
Definition at line 187 of file [fasp_const.h](#).

9.17.2.105 SMOOTHER_L1DIAG

`#define SMOOTHER_L1DIAG 10`
L1 norm diagonal scaling smoother
Definition at line 196 of file [fasp_const.h](#).

9.17.2.106 SMOOTHER_POLY

`#define SMOOTHER_POLY 9`
Polynomial smoother
Definition at line 195 of file [fasp_const.h](#).

9.17.2.107 SMOOTHER_SGS

`#define SMOOTHER_SGS 3`
Symmetric Gauss-Seidel smoother
Definition at line 189 of file [fasp_const.h](#).

9.17.2.108 SMOOTHER_SGSOR

```
#define SMOOTHER_SGSOR 8
```

SGS + SSOR smoother
Definition at line 194 of file [fasp_const.h](#).

9.17.2.109 SMOOTHER_SOR

```
#define SMOOTHER_SOR 5
```

SOR smoother
Definition at line 191 of file [fasp_const.h](#).

9.17.2.110 SMOOTHER_SPETEN

```
#define SMOOTHER_SPETEN 19
```

Used in monolithic AMG for black-oil
Definition at line 202 of file [fasp_const.h](#).

9.17.2.111 SMOOTHER_SSOR

```
#define SMOOTHER_SSOR 6
```

SSOR smoother
Definition at line 192 of file [fasp_const.h](#).

9.17.2.112 SOLVER_AMG

```
#define SOLVER_AMG 21
```

AMG as an iterative solver
Definition at line 120 of file [fasp_const.h](#).

9.17.2.113 SOLVER_BiCGstab

```
#define SOLVER_BiCGstab 2
```

Bi-Conjugate Gradient Stabilized
Definition at line 104 of file [fasp_const.h](#).

9.17.2.114 SOLVER_CG

```
#define SOLVER_CG 1
```

Conjugate Gradient
Definition at line 103 of file [fasp_const.h](#).

9.17.2.115 SOLVER_DEFAULT

```
#define SOLVER_DEFAULT 0
```

Definition of solver types for iterative methods.
Use default solver in FASP
Definition at line 101 of file [fasp_const.h](#).

9.17.2.116 SOLVER_FMG

#define SOLVER_FMG 22
Full AMG as an solver
Definition at line 121 of file [fasp_const.h](#).

9.17.2.117 SOLVER_GCG

#define SOLVER_GCG 7
Generalized Conjugate Gradient
Definition at line 109 of file [fasp_const.h](#).

9.17.2.118 SOLVER_GCR

#define SOLVER_GCR 8
Generalized Conjugate Residual
Definition at line 110 of file [fasp_const.h](#).

9.17.2.119 SOLVER_GMRES

#define SOLVER_GMRES 4
Generalized Minimal Residual
Definition at line 106 of file [fasp_const.h](#).

9.17.2.120 SOLVER_MinRes

#define SOLVER_MinRes 3
Minimal Residual
Definition at line 105 of file [fasp_const.h](#).

9.17.2.121 SOLVER_MUMPS

#define SOLVER_MUMPS 33
Direct Solver: MUMPS
Definition at line 125 of file [fasp_const.h](#).

9.17.2.122 SOLVER_PARDISO

#define SOLVER_PARDISO 34
Direct Solver: PARDISO
Definition at line 126 of file [fasp_const.h](#).

9.17.2.123 SOLVER_SBiCGstab

#define SOLVER_SBiCGstab 12
BiCGstab with safety net
Definition at line 113 of file [fasp_const.h](#).

9.17.2.124 SOLVER_SCG

```
#define SOLVER_SCG 11
```

Conjugate Gradient with safety net
Definition at line 112 of file [fasp_const.h](#).

9.17.2.125 SOLVER_SGCG

```
#define SOLVER_SGCG 17
```

GCG with safety net
Definition at line 118 of file [fasp_const.h](#).

9.17.2.126 SOLVER_SGMRES

```
#define SOLVER_SGMRES 14
```

GMRes with safety net
Definition at line 115 of file [fasp_const.h](#).

9.17.2.127 SOLVER_SMinRes

```
#define SOLVER_SMinRes 13
```

MinRes with safety net
Definition at line 114 of file [fasp_const.h](#).

9.17.2.128 SOLVER_SUPERLU

```
#define SOLVER_SUPERLU 31
```

Direct Solver: SuperLU
Definition at line 123 of file [fasp_const.h](#).

9.17.2.129 SOLVER_SVFGMRES

```
#define SOLVER_SVFGMRES 16
```

Variable-restart FGMRES with safety net
Definition at line 117 of file [fasp_const.h](#).

9.17.2.130 SOLVER_SVGMRES

```
#define SOLVER_SVGMRES 15
```

Variable-restart GMRES with safety net
Definition at line 116 of file [fasp_const.h](#).

9.17.2.131 SOLVER_UMFPACK

```
#define SOLVER_UMFPACK 32
```

Direct Solver: UMFPack
Definition at line 124 of file [fasp_const.h](#).

9.17.2.132 SOLVER_VFGMRES

#define SOLVER_VFGMRES 6
Variable Restarting Flexible GMRES
Definition at line 108 of file [fasp_const.h](#).

9.17.2.133 SOLVER_VGMRES

#define SOLVER_VGMRES 5
Variable Restarting GMRES
Definition at line 107 of file [fasp_const.h](#).

9.17.2.134 SPAIR

#define SPAIR 4
symmetric pairwise aggregation
Definition at line 172 of file [fasp_const.h](#).

9.17.2.135 STAG_RATIO

#define STAG_RATIO 1e-4
Stagnation tolerance = tol*STAGRATIO
Definition at line 258 of file [fasp_const.h](#).

9.17.2.136 STOP_MOD_REL_RES

#define STOP_MOD_REL_RES 3
modified relative residual $\|r\|/\|x\|$
Definition at line 133 of file [fasp_const.h](#).

9.17.2.137 STOP_REL_PRECRES

#define STOP_REL_PRECRES 2
relative B-residual $\|r\|_B/\|b\|_B$
Definition at line 132 of file [fasp_const.h](#).

9.17.2.138 STOP_REL_RES

#define STOP_REL_RES 1
Definition of iterative solver stopping criteria types.
relative residual $\|r\|/\|b\|$
Definition at line 131 of file [fasp_const.h](#).

9.17.2.139 TRUE

#define TRUE 1
Definition of logic type.
logic TRUE
Definition at line 61 of file [fasp_const.h](#).

9.17.2.140 UA_AMG

`#define UA_AMG 3`
unsmoothed aggregation AMG
Definition at line 164 of file [fasp_const.h](#).

9.17.2.141 UNPT

`#define UNPT -1`
Undetermined points
Definition at line 225 of file [fasp_const.h](#).

9.17.2.142 USERDEFINED

`#define USERDEFINED 0`
Type of ordering for smoothers.
User defined order
Definition at line 239 of file [fasp_const.h](#).

9.17.2.143 V_CYCLE

`#define V_CYCLE 1`
Definition of cycle types.
V-cycle
Definition at line 177 of file [fasp_const.h](#).

9.17.2.144 VMB

`#define VMB 2`
VMB aggregation
Definition at line 170 of file [fasp_const.h](#).

9.17.2.145 VW_CYCLE

`#define VW_CYCLE 12`
VW-cycle
Definition at line 181 of file [fasp_const.h](#).

9.17.2.146 W_CYCLE

`#define W_CYCLE 2`
W-cycle
Definition at line 178 of file [fasp_const.h](#).

9.17.2.147 WV_CYCLE

```
#define WV_CYCLE 21
```

WV-cycle

Definition at line 182 of file [fasp_const.h](#).

9.18 fasp_const.h

[Go to the documentation of this file.](#)

```
00001
00013 #ifndef __FASP_CONST__          /*-- allow multiple inclusions --*/
00014 #define __FASP_CONST__
00015
00019 #define FASP_SUCCESS              0
00020 //-----
00021 #define ERROR_READ_FILE          -1
00022 #define ERROR_OPEN_FILE          -10
00023 #define ERROR_WRONG_FILE         -11
00024 #define ERROR_INPUT_PAR          -13
00025 #define ERROR_REGRESS            -14
00026 #define ERROR_MAT_SIZE           -15
00027 #define ERROR_NUM_BLOCKS         -18
00028 #define ERROR_MISC               -19
00029 //-----
00030 #define ERROR_ALLOC_MEM          -20
00031 #define ERROR_DATA_STRUCTURE     -21
00032 #define ERROR_DATA_ZERODIAG     -22
00033 #define ERROR_DUMMY_VAR          -23
00034 //-----
00035 #define ERROR_AMG_INTERP_TYPE    -30
00036 #define ERROR_AMG_SMOOTH_TYPE    -31
00037 #define ERROR_AMG_COARSE_TYPE    -32
00038 #define ERROR_AMG_COARSEING      -33
00039 #define ERROR_AMG_SETUP          -39
00040 //-----
00041 #define ERROR_SOLVER_TYPE        -40
00042 #define ERROR_SOLVER_PRECTYPE    -41
00043 #define ERROR_SOLVER_STAG        -42
00044 #define ERROR_SOLVER_SOLSTAG     -43
00045 #define ERROR_SOLVER_TOLSMALL    -44
00046 #define ERROR_SOLVER_ILUSETUP    -45
00047 #define ERROR_SOLVER_MISC        -46
00048 #define ERROR_SOLVER_MAXIT       -48
00049 #define ERROR_SOLVER_EXIT        -49
00050 //-----
00051 #define ERROR_QUAD_TYPE          -60
00052 #define ERROR_QUAD_DIM           -61
00053 //-----
00054 #define ERROR_LIC_TYPE           -80
00055 //-----
00056 #define ERROR_UNKNOWN            -99
00061 #define TRUE                     1
00062 #define FALSE                    0
00067 #define ON                       1
00068 #define OFF                      0
00073 #define PRINT_NONE               0
00074 #define PRINT_MIN                1
00075 #define PRINT_SOME               2
00076 #define PRINT_MORE               4
00077 #define PRINT_MOST               8
00078 #define PRINT_ALL                10
00083 #define MAT_FREE                 0
00084 //-----
00085 #define MAT_CSR                  1
00086 #define MAT_BSR                  2
00087 #define MAT_STR                  3
00088 #define MAT_CSRL                 6
00089 #define MAT_SymCSR               7
00090 #define MAT_BLC                  8
00091 //-----
00092 //      For bordered systems in reservoir simulation
00093 //-----
00094 #define MAT_bCSR                  11
00095 #define MAT_bBSR                  12
00096 #define MAT_bSTR                  13
00101 #define SOLVER_DEFAULT           0
```

```

00102 //-----
00103 #define SOLVER_CG 1
00104 #define SOLVER_BiCGstab 2
00105 #define SOLVER_MinRes 3
00106 #define SOLVER_GMRES 4
00107 #define SOLVER_VGMRES 5
00108 #define SOLVER_VFGMRES 6
00109 #define SOLVER_GCG 7
00110 #define SOLVER_GCR 8
00111 //-----
00112 #define SOLVER_SCG 11
00113 #define SOLVER_SBiCGstab 12
00114 #define SOLVER_SMinRes 13
00115 #define SOLVER_SGMRES 14
00116 #define SOLVER_SVGMRES 15
00117 #define SOLVER_SVFGMRES 16
00118 #define SOLVER_SGCG 17
00119 //-----
00120 #define SOLVER_AMG 21
00121 #define SOLVER_FMG 22
00122 //-----
00123 #define SOLVER_SUPERLU 31
00124 #define SOLVER_UMFPACK 32
00125 #define SOLVER_MUMPS 33
00126 #define SOLVER_PARDISO 34
00131 #define STOP_REL_RES 1
00132 #define STOP_REL_PRECRES 2
00133 #define STOP_MOD_REL_RES 3
00138 #define PREC_NULL 0
00139 #define PREC_DIAG 1
00140 #define PREC_AMG 2
00141 #define PREC_FMG 3
00142 #define PREC_ILU 4
00143 #define PREC_SCHWARZ 5
00148 #define ILUk 1
00149 #define ILUt 2
00150 #define ILUtp 3
00155 #define SCHWARZ_FORWARD 1
00156 #define SCHWARZ_BACKWARD 2
00157 #define SCHWARZ_SYMMETRIC 3
00162 #define CLASSIC_AMG 1
00163 #define SA_AMG 2
00164 #define UA_AMG 3
00169 #define PAIRWISE 1
00170 #define VMB 2
00171 #define NPAIR 3
00172 #define SPAIR 4
00177 #define V_CYCLE 1
00178 #define W_CYCLE 2
00179 #define AMLI_CYCLE 3
00180 #define NL_AMLI_CYCLE 4
00181 #define VW_CYCLE 12
00182 #define WV_CYCLE 21
00187 #define SMOOTHER_JACOBI 1
00188 #define SMOOTHER_GS 2
00189 #define SMOOTHER_SGS 3
00190 #define SMOOTHER_CG 4
00191 #define SMOOTHER_SOR 5
00192 #define SMOOTHER_SSOR 6
00193 #define SMOOTHER_GSOR 7
00194 #define SMOOTHER_SGSOR 8
00195 #define SMOOTHER_POLY 9
00196 #define SMOOTHER_L1DIAG 10
00201 #define SMOOTHER_BLKOIL 11
00202 #define SMOOTHER_SPETEN 19
00207 #define COARSE_RS 1
00208 #define COARSE_RSP 2
00209 #define COARSE_CR 3
00210 #define COARSE_AC 4
00211 #define COARSE_MIS 5
00216 #define INTERP_DIR 1
00217 #define INTERP_STD 2
00218 #define INTERP_ENG 3
00219 #define INTERP_EXT 6
00224 #define GOPT -5
00225 #define UNPT -1
00226 #define FGPT 0
00227 #define CGPT 1
00228 #define ISPT 2
00233 #define NO_ORDER 0
00234 #define CF_ORDER 1

```

```

00239 #define USERDEFINED          0
00240 #define CPFIRST                1
00241 #define FPFIRST               -1
00242 #define ASCEND                 12
00243 #define DESCEND                21
00248 #define BIGREAL              1e+20
00249 #define SMALLREAL              1e-20
00250 #define SMALLREAL2             1e-40
00251 #define MAX_REFINE_LVL        20
00252 #define MAX_AMG_LVL            20
00253 #define MIN_CDOF               20
00254 #define MIN_CRATE              0.9
00255 #define MAX_CRATE              20.0
00256 #define MAX_RESTART            20
00257 #define MAX_STAG                20
00258 #define STAG_RATIO              1e-4
00259 #define FPNA_RATIO              1e-8
00260 #define OPENMP_HOLDS           2000
00262 #endif                        /* end if for __FASP_CONST__ */
00263
00264 /*-----*/
00265 /*--          End of File          --*/
00266 /*-----*/

```

9.19 fasp_grid.h File Reference

Header file for FASP grid.

Data Structures

- struct [grid2d](#)
Two dimensional grid data structure.

Macros

- #define [__FASPGRID_HEADER__](#)

Typedefs

- typedef struct [grid2d](#) [grid2d](#)
- typedef [grid2d](#) * [pgrid2d](#)
- typedef const [grid2d](#) * [pcgrid2d](#)

9.19.1 Detailed Description

Header file for FASP grid.
Copyright (C) 2015–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [fasp_grid.h](#).

9.19.2 Macro Definition Documentation

9.19.2.1 [__FASPGRID_HEADER__](#)

#define [__FASPGRID_HEADER__](#)
indicate [fasp_grid.h](#) has been included before
Definition at line 12 of file [fasp_grid.h](#).

9.19.3 Typedef Documentation

9.19.3.1 grid2d

typedef struct `grid2d` `grid2d`
2D grid type for plotting

9.19.3.2 pcgrid2d

typedef const `grid2d*` `pcgrid2d`
Grid in 2d
Definition at line 45 of file `fasp_grid.h`.

9.19.3.3 pgrid2d

typedef `grid2d*` `pgrid2d`
Grid in 2d
Definition at line 43 of file `fasp_grid.h`.

9.20 fasp_grid.h

[Go to the documentation of this file.](#)

```
00001
00011 #ifndef __FASPGRID_HEADER__ /*-- allow multiple inclusions --*/
00012 #define __FASPGRID_HEADER__
00024 typedef struct grid2d {
00025
00026     REAL (*p)[2];
00027     INT (*e)[2];
00028     INT (*t)[3];
00029     INT (*s)[3];
00030     INT *pdir;
00031     INT *edir;
00033     INT *pfather;
00034     INT *efather;
00035     INT *tfather;
00037     INT vertices;
00038     INT edges;
00039     INT triangles;
00041 } grid2d;
00043 typedef grid2d *pgrid2d;
00045 typedef const grid2d *pcgrid2d;
00047 #endif /* end if for __FASPGRID_HEADER__ */
00048
00049 /*-----*/
00050 /*--      End of File      --*/
00051 /*-----*/
```

9.21 AuxArray.c File Reference

Simple array operations – init, set, copy, etc.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- void `fasp_darray_set` (const `INT` n, `REAL` *x, const `REAL` val)

Set initial value for an array to be $x=val$.

- void `fasp_iarray_set` (const `INT` n , `INT` $*x$, const `INT` val)

Set initial value for an array to be $x=val$.

- void `fasp_darray_cp` (const `INT` n , const `REAL` $*x$, `REAL` $*y$)

Copy an array to the other $y=x$.

- void `fasp_iarray_cp` (const `INT` n , const `INT` $*x$, `INT` $*y$)

Copy an array to the other $y=x$.

9.21.1 Detailed Description

Simple array operations – init, set, copy, etc.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxThreads.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxArray.c](#).

9.21.2 Function Documentation

9.21.2.1 `fasp_darray_cp()`

```
void fasp_darray_cp (
    const INT  $n$ ,
    const REAL  $*x$ ,
    REAL  $*y$  )
```

Copy an array to the other $y=x$.

Parameters

n	Number of variables
x	Pointer to the original vector
y	Pointer to the destination vector

Author

Chensong Zhang

Date

2010/04/03

Definition at line 164 of file [AuxArray.c](#).

9.21.2.2 `fasp_darray_set()`

```
void fasp_darray_set (
    const INT  $n$ ,
```

```
    REAL * x,  
    const REAL val )
```

Set initial value for an array to be x=val.

Parameters

<i>n</i>	Number of variables
<i>x</i>	Pointer to the vector
<i>val</i>	Initial value for the REAL array

Author

Chensong Zhang

Date

04/03/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 41 of file [AuxArray.c](#).

9.21.2.3 fasp_iarray_cp()

```
void fasp_iarray_cp (  
    const INT n,  
    const INT * x,  
    INT * y )
```

Copy an array to the other y=x.

Parameters

<i>n</i>	Number of variables
<i>x</i>	Pointer to the original vector
<i>y</i>	Pointer to the destination vector

Author

Chunsheng Feng, Xiaoqiang Yue

Date

05/23/2012

Definition at line 184 of file [AuxArray.c](#).

9.21.2.4 fasp_iarray_set()

```
void fasp_iarray_set (  
    const INT n,  
    INT * x,  
    const INT val )
```

Set initial value for an array to be `x=val`.

Parameters

<i>n</i>	Number of variables
<i>x</i>	Pointer to the vector
<i>val</i>	Initial value for the REAL array

Author

Chensong Zhang

Date

04/03/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/25/2012

Definition at line 103 of file [AuxArray.c](#).

9.22 AuxArray.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <math.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00041 void fasp_darray_set (const INT    n,
00042                     REAL          **x,
00043                     const REAL    val)
00044 {
00045     SHORT use_omp = FALSE;
00046
00047 #ifdef _OPENMP
00048     INT nthreads = 1;
00049
00050     if ( n > OPENMP_HOLDS ) {
00051         use_omp = TRUE;
00052         nthreads = fasp_get_num_threads();
00053     }
00054 #endif
00055
00056     if (val == 0.0) {
00057         if (use_omp) {
00058             #ifdef _OPENMP
00059                 INT mybegin, myend, myid;
00060                 #pragma omp parallel for private(myid, mybegin, myend)
00061                 for (myid = 0; myid < nthreads; myid++) {
00062                     fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00063                     memset(&x[mybegin], 0x0, sizeof(REAL)*(myend-mybegin));
00064                 }
00065             #endif
00066         }
00067         else
00068             memset(x, 0x0, sizeof(REAL)*n);
00069     }
00070     else {
00071         INT i;
00072
00073         if (use_omp) {
00074             #ifdef _OPENMP
00075                 INT mybegin, myend, myid;

```

```

00076 #pragma omp parallel for private(myid,mybegin,myend,i)
00077     for (myid = 0; myid < nthreads; myid++) {
00078         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00079         for (i=mybegin; i<myend; ++i) x[i]=val;
00080     }
00081 #endif
00082     }
00083     else {
00084         for (i=0; i<n; ++i) x[i] = val;
00085     }
00086 }
00087 }
00088
00103 void fasp_iarray_set (const INT    n,
00104                      INT          *x,
00105                      const INT    val)
00106 {
00107     SHORT use_omp = FALSE;
00108
00109 #ifdef _OPENMP
00110     INT nthreads = 1;
00111
00112     if ( n > OPENMP_HOLDS ) {
00113         use_omp = TRUE;
00114         nthreads = fasp_get_num_threads();
00115     }
00116 #endif
00117
00118     if (val == 0) {
00119         if (use_omp) {
00120 #ifdef _OPENMP
00121             INT mybegin,myend,myid;
00122 #pragma omp parallel for private(myid, mybegin, myend)
00123             for (myid = 0; myid < nthreads; myid++) {
00124                 fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00125                 memset(&x[mybegin], 0, sizeof(INT)*(myend-mybegin));
00126             }
00127 #endif
00128         }
00129         else {
00130             memset(x, 0, sizeof(INT)*n);
00131         }
00132     }
00133     else {
00134         INT i;
00135
00136         if (use_omp) {
00137 #ifdef _OPENMP
00138             INT mybegin,myend,myid;
00139 #pragma omp parallel for private(myid, mybegin, myend,i)
00140             for (myid = 0; myid < nthreads; myid++) {
00141                 fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00142                 for (i=mybegin; i<myend; ++i) x[i]=val;
00143             }
00144 #endif
00145         }
00146         else {
00147             for (i=0; i<n; ++i) x[i]=val;
00148         }
00149     }
00150 }
00151
00164 void fasp_darray_cp (const INT    n,
00165                     const REAL  *x,
00166                     REAL        *y)
00167 {
00168     memcpy(y, x, n*sizeof(REAL));
00169 }
00170
00171
00184 void fasp_iarray_cp (const INT    n,
00185                     const INT    *x,
00186                     INT          *y)
00187 {
00188     memcpy(y, x, n*sizeof(INT));
00189 }
00190
00191 /*-----*/
00192 /*--      End of File      --*/
00193 /*-----*/

```

9.23 AuxConvert.c File Reference

Utilities for encoding format conversion.

```
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- unsigned long [fasp_aux_change_endian4](#) (const unsigned long x)
Swap order for different endian systems.
- double [fasp_aux_change_endian8](#) (const double x)
Swap order for different endian systems.
- double [fasp_aux_bbyteToldouble](#) (const unsigned char bytes[])
Swap order of double-precision float for different endian systems.

9.23.1 Detailed Description

Utilities for encoding format conversion.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxConvert.c](#).

9.23.2 Function Documentation

9.23.2.1 [fasp_aux_bbyteToldouble\(\)](#)

```
double fasp_aux_bbyteToldouble (
    const unsigned char bytes[] )
```

Swap order of double-precision float for different endian systems.

Parameters

<i>bytes</i>	A unsigned char
--------------	-----------------

Returns

Unsigend long ineger after swapping

Author

Chensong Zhang

Date

11/16/2009

Definition at line 81 of file [AuxConvert.c](#).**9.23.2.2 fasp_aux_change_endian4()**

```
unsigned long fasp_aux_change_endian4 (  
    const unsigned long x )
```

Swap order for different endian systems.

Parameters

x	An unsigned long integer
---	--------------------------

Returns

Unsigned long integer after swapping

Author

Chensong Zhang

Date

11/16/2009

Definition at line 32 of file [AuxConvert.c](#).**9.23.2.3 fasp_aux_change_endian8()**

```
double fasp_aux_change_endian8 (  
    const double x )
```

Swap order for different endian systems.

Parameters

x	A unsigned long integer
---	-------------------------

Returns

Unsigned long integer after swapping

Author

Chensong Zhang

Date

11/16/2009

Definition at line 50 of file [AuxConvert.c](#).

9.24 AuxConvert.c

[Go to the documentation of this file.](#)

```

00001
00013 #include "fasp.h"
00014 #include "fasp_funcs.h"
00015
00016 /*-----*/
00017 /*--      Public Functions      --*/
00018 /*-----*/
00019
00032 unsigned long fasp_aux_change_endian4 (const unsigned long x)
00033 {
00034     unsigned char *ptr = (unsigned char *)&x;
00035     return (ptr[0] < 24) | (ptr[1] < 16) | (ptr[2] < 8) | ptr[3];
00036 }
00037
00050 double fasp_aux_change_endian8 (const double x)
00051 {
00052     double dbl;
00053     unsigned char *bytes, *buffer;
00054
00055     buffer=(unsigned char *)&dbl;
00056     bytes=(unsigned char *)&x;
00057
00058     buffer[0]=bytes[7];
00059     buffer[1]=bytes[6];
00060     buffer[2]=bytes[5];
00061     buffer[3]=bytes[4];
00062     buffer[4]=bytes[3];
00063     buffer[5]=bytes[2];
00064     buffer[6]=bytes[1];
00065     buffer[7]=bytes[0];
00066     return dbl;
00067 }
00068
00081 double fasp_aux_bbyteToldouble (const unsigned char bytes[])
00082 {
00083     double dbl;
00084     unsigned char *buffer;
00085     buffer=(unsigned char *)&dbl;
00086     buffer[0]=bytes[7];
00087     buffer[1]=bytes[6];
00088     buffer[2]=bytes[5];
00089     buffer[3]=bytes[4];
00090     buffer[4]=bytes[3];
00091     buffer[5]=bytes[2];
00092     buffer[6]=bytes[1];
00093     buffer[7]=bytes[0];
00094     return dbl;
00095 }
00096
00097 /*-----*/
00098 /*--      End of File      --*/
00099 /*-----*/

```

9.25 AuxGivens.c File Reference

Givens transformation.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_aux_givens` (const `REAL` beta, const `dCSRmat` *H, `dvector` *y, `REAL` *work)
*Perform Givens rotations to compute $y \mid \beta e_1 - H*y$.*

9.25.1 Detailed Description

Givens transformation.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2008–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxGivens.c](#).

9.25.2 Function Documentation

9.25.2.1 fasp_aux_givens()

```
void fasp_aux_givens (
    const REAL beta,
    const dCSRmat * H,
    dvector * y,
    REAL * work )
```

Perform Givens rotations to compute $y \mid \beta e_1 - H y$.

Parameters

<i>beta</i>	Norm of residual r_0
<i>H</i>	Upper Hessenberg dCSRmat matrix: $(m+1) \times m$
<i>y</i>	Minimizer of $\mid \beta e_1 - H y \mid$
<i>work</i>	Temporary work array

Author

Xuehai Huang

Date

10/19/2008

Definition at line 36 of file [AuxGivens.c](#).

9.26 AuxGivens.c

[Go to the documentation of this file.](#)

```
00001
00013 #include <math.h>
00014
00015 #include "fasp.h"
00016 #include "fasp_funcs.h"
00017
00018 /*-----*/
00019 /*--      Public Functions      --*/
00020 /*-----*/
00021
00036 void fasp_aux_givens (const REAL      beta,
00037                     const dCSRmat *H,
```

```

00038             dvector      *y,
00039             REAL          *work)
00040 {
00041     const INT  Hsize = H->row;
00042     INT        i, j, istart, idiag, iplstart;
00043     REAL       h0, h1, r, c, s, tempi, temp1, sum;
00044
00045     memset(&work, 0x0, sizeof(REAL)*Hsize);
00046     work[0] = beta;
00047
00048     for ( i=0; i<Hsize-1; ++i ) {
00049         istart = H->IA[i];
00050         iplstart = H->IA[i+1];
00051         if (i==0) idiag = istart;
00052         else idiag = istart+1;
00053
00054         h0 = H->val[idiag]; // h0=H[i][i]
00055         h1 = H->val[H->IA[i+1]]; // h1=H[i+1][i]
00056         r = sqrt(h0*h0+h1*h1);
00057         c = h0/r; s = h1/r;
00058
00059         for ( j=idiag; j<iplstart; ++j ) {
00060             tempi = H->val[j];
00061             temp1 = H->val[iplstart+(j-idiag)];
00062             H->val[j] = c*tempi+s*temp1;
00063             H->val[iplstart+(j-idiag)] = c*temp1-s*tempi;
00064         }
00065
00066         tempi = c*work[i]+s*work[i+1];
00067         temp1 = c*work[i+1]-s*work[i];
00068
00069         work[i] = tempi; work[i+1]=temp1;
00070     }
00071
00072     for ( i = Hsize-2; i >= 0; --i ) {
00073         sum = work[i];
00074         istart = H->IA[i];
00075         if (i==0) idiag = istart;
00076         else idiag = istart+1;
00077
00078         for ( j=Hsize-2; j>i; --j ) sum-=H->val[idiag+j-i]*y->val[j];
00079
00080         y->val[i] = sum/H->val[idiag];
00081     }
00082 }
00083 }
00084
00085 /*-----*/
00086 /*--      End of File      --*/
00087 /*-----*/

```

9.27 AuxGraphics.c File Reference

Graphical output for CSR matrix.

```

#include <math.h>
#include "fasp.h"
#include "fasp_grid.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_dcsr_subplot` (const `dCSRmat` *A, const char *filename, int size)
Write sparse matrix pattern in BMP file format.
- void `fasp_dcsr_plot` (const `dCSRmat` *A, const char *fname)
Write dCSR sparse matrix pattern in BMP file format.
- void `fasp_dbsr_subplot` (const `dBSRmat` *A, const char *filename, int size)
Write sparse matrix pattern in BMP file format.
- void `fasp_dbsr_plot` (const `dBSRmat` *A, const char *fname)

Write dBSR sparse matrix pattern in BMP file format.

- void [fasp_grid2d_plot](#) ([pgrid2d](#) pg, int level)

Output grid to a EPS file.

9.27.1 Detailed Description

Graphical output for CSR matrix.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxMemory.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxGraphics.c](#).

9.27.2 Function Documentation

9.27.2.1 [fasp_dbsr_plot\(\)](#)

```
void fasp_dbsr_plot (
    const dBSRmat * A,
    const char * fname )
```

Write dBSR sparse matrix pattern in BMP file format.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>fname</i>	File name

Author

Chunsheng Feng

Date

11/16/2013

Note

The routine [fasp_dbsr_plot](#) writes pattern of the specified [dBSRmat](#) matrix in uncompressed BMP file format (Windows bitmap) to a binary file whose name is specified by the character string filename.

Each pixel corresponds to one matrix element. The pixel colors have the following meaning:

White structurally zero element Black zero element Blue positive element Red negative element Brown nearly zero element

Definition at line [339](#) of file [AuxGraphics.c](#).

9.27.2.2 [fasp_dbsr_subplot\(\)](#)

```
void fasp_dbsr_subplot (
    const dBSRmat * A,
```

```
const char * filename,  
int size )
```

Write sparse matrix pattern in BMP file format.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>filename</i>	File name
<i>size</i>	size*size is the picture size for the picture

Author

Chunsheng Feng

Date

11/16/2013

Note

The routine `fasp_dbsr_subplot` writes pattern of the specified [dBSRmat](#) matrix in uncompressed BMP file format (Windows bitmap) to a binary file whose name is specified by the character string `filename`.

Each pixel corresponds to one matrix element. The pixel colors have the following meaning:

White structurally zero element Black zero element Blue positive element Red negative element Brown nearly zero element

Definition at line [259](#) of file [AuxGraphics.c](#).

9.27.2.3 fasp_dcsr_plot()

```
void fasp_dcsr_plot (  
    const dCSRmat * A,  
    const char * fname )
```

Write dCSR sparse matrix pattern in BMP file format.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>fname</i>	File name to plot to

Author

Chunsheng Feng

Date

11/16/2013

Note

The routine `fasp_dcsr_plot` writes pattern of the specified [dCSRmat](#) matrix in uncompressed BMP file format (Windows bitmap) to a binary file whose name is specified by the character string `filename`.

Each pixel corresponds to one matrix element. The pixel colors have the following meaning:

White structurally zero element Black zero element Blue positive element Red negative element Brown nearly zero element

Definition at line 117 of file [AuxGraphics.c](#).

9.27.2.4 fasp_dcsr_subplot()

```
void fasp_dcsr_subplot (
    const dCSRmat * A,
    const char * filename,
    int size )
```

Write sparse matrix pattern in BMP file format.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>filename</i>	File name
<i>size</i>	size*size is the picture size for the picture

Author

Chensong Zhang

Date

03/29/2009

Note

The routine `fasp_dcsr_subplot` writes pattern of the specified [dCSRmat](#) matrix in uncompressed BMP file format (Windows bitmap) to a binary file whose name is specified by the character string `filename`.

Each pixel corresponds to one matrix element. The pixel colors have the following meaning:

White structurally zero element Blue positive element Red negative element Brown nearly zero element

Definition at line 57 of file [AuxGraphics.c](#).

9.27.2.5 fasp_grid2d_plot()

```
void fasp_grid2d_plot (
    pgrid2d pg,
    int level )
```

Output grid to a EPS file.

Parameters

<i>pg</i>	Pointer to grid in 2d
<i>level</i>	Number of levels

Author

Chensong Zhang

Date

03/29/2009

Definition at line 478 of file AuxGraphics.c.

9.28 AuxGraphics.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <math.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_grid.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/
00021 /*--  Declare Private Functions  --*/
00022 /*-----*/
00023
00024 static void put_byte (FILE *fp, const int c);
00025 static void put_word (FILE *fp, const int w);
00026 static void put_dword (FILE *fp, const int d);
00027 static int write_bmp16 (const char *fname, int m, int n, const char map[]);
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00057 void fasp_dcsr_subplot (const dCSRmat *A,
00058                       const char *filename,
00059                       int size)
00060 {
00061     INT m = A->row, n = A->col, minmn = MIN(m,n);
00062     int i, j, k;
00063     char *map;
00064
00065     if ( size>minmn ) size = minmn;
00066     map = (char *)fasp_mem_malloc(size * size, sizeof(char));
00067
00068     printf("Writing matrix pattern to '%s'...\n",filename);
00069
00070     memset((void *)map, 0x0F, size * size);
00071
00072     for (i = 0; i < size; ++i) {
00073         for (j = A->IA[i]; j < A->IA[i+1]; ++j) {
00074             if (A->JA[j]<size) {
00075                 k = size*i + A->JA[j];
00076                 if (map[k] != 0x0F)
00077                     map[k] = 0x0F;
00078                 else if (A->val[j] > 1e-20)
00079                     map[k] = 0x09; /* bright blue */
00080                 else if (A->val[j] < -1e-20)
00081                     map[k] = 0x0C; /* bright red */
00082                 else
00083                     map[k] = 0x06; /* brown */
00084             } // end if
00085         } // end for j
00086     } // end for i
00087
00088     write_bmp16(filename, size, size, map);
00089
00090     fasp_mem_free(map); map = NULL;
00091 }
00092
00117 void fasp_dcsr_plot (const dCSRmat *A,
00118                    const char *fname)
00119 {
00120     FILE *fp;
00121     INT offset, bmsize, i, j, b;
00122     INT n = A->col, m = A->row;
00123     INT size;

```

```

00124
00125     INT col;
00126     REAL val;
00127     char *map;
00128
00129     size = ( (n+7)/8 ) * 8;
00130
00131     map = (char *)fasp_mem_malloc(size, sizeof(char));
00132
00133     memset(map, 0x0F, size);
00134
00135     if (!(1 <= m && m <= 32767))
00136         printf("### ERROR: Invalid num of rows %d! [%s]\n", m, __FUNCTION__);
00137
00138     if (!(1 <= n && n <= 32767))
00139         printf("### ERROR: Invalid num of cols %d! [%s]\n", n, __FUNCTION__);
00140
00141     fp = fopen(fname, "wb");
00142     if (fp == NULL) {
00143         printf("### ERROR: Unable to create '%s'! [%s]\n", fname, __FUNCTION__);
00144         goto FINISH;
00145     }
00146
00147     offset = 14 + 40 + 16 * 4;
00148     bmsize = (4 * n + 31) / 32;
00149     /* struct BMPFILEHEADER (14 bytes) */
00150     /* UINT bfType */ put_byte(fp, 'B'); put_byte(fp, 'M');
00151     /* DWORD bfSize */ put_dword(fp, offset + bmsize * 4);
00152     /* UINT bfReserved1 */ put_word(fp, 0);
00153     /* UNIT bfReserved2 */ put_word(fp, 0);
00154     /* DWORD bfOffBits */ put_dword(fp, offset);
00155     /* struct BMPINFOHEADER (40 bytes) */
00156     /* DWORD biSize */ put_dword(fp, 40);
00157     /* LONG biWidth */ put_dword(fp, n);
00158     /* LONG biHeight */ put_dword(fp, m);
00159     /* WORD biPlanes */ put_word(fp, 1);
00160     /* WORD biBitCount */ put_word(fp, 4);
00161     /* DWORD biCompression */ put_dword(fp, 0 /* BI_RGB */);
00162     /* DWORD biSizeImage */ put_dword(fp, 0);
00163     /* LONG biXPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
00164     /* LONG biYPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
00165     /* DWORD biClrUsed */ put_dword(fp, 0);
00166     /* DWORD biClrImportant */ put_dword(fp, 0);
00167     /* struct RGBQUAD (16 * 4 = 64 bytes) */
00168     /* CGA-compatible colors: */
00169     /* 0x00 = black */ put_dword(fp, 0x000000);
00170     /* 0x01 = blue */ put_dword(fp, 0x000080);
00171     /* 0x02 = green */ put_dword(fp, 0x008000);
00172     /* 0x03 = cyan */ put_dword(fp, 0x008080);
00173     /* 0x04 = red */ put_dword(fp, 0x800000);
00174     /* 0x05 = magenta */ put_dword(fp, 0x800080);
00175     /* 0x06 = brown */ put_dword(fp, 0x808000);
00176     /* 0x07 = light gray */ put_dword(fp, 0xC0C0C0);
00177     /* 0x08 = dark gray */ put_dword(fp, 0x808080);
00178     /* 0x09 = bright blue */ put_dword(fp, 0x0000FF);
00179     /* 0x0A = bright green */ put_dword(fp, 0x00FF00);
00180     /* 0x0B = bright cyan */ put_dword(fp, 0x00FFFF);
00181     /* 0x0C = bright red */ put_dword(fp, 0xFF0000);
00182     /* 0x0D = bright magenta */ put_dword(fp, 0xFF00FF);
00183     /* 0x0E = yellow */ put_dword(fp, 0xFFFF00);
00184     /* 0x0F = white */ put_dword(fp, 0FFFFFFF);
00185     /* pixel data bits */
00186     b = 0;
00187
00188     // for(i=((m+7)/8)*8 - 1; i>=m; i--){
00189     //     memset(map, 0x0F, size);
00190     //     for (j = 0; j < size; ++j) {
00191     //         b <<= 4;
00192     //         b |= (j < n ? map[j] & 15 : 0);
00193     //         if (j & 1) put_byte(fp, b);
00194     //     }
00195     // }
00196
00197     for (i = A->row-1; i >=0; i-- ) {
00198         memset(map, 0x0F, size);
00199
00200         for (j = A->IA[i]; j < A->IA[i+1]; j++ ) {
00201             col = A->JA[j];
00202             val = A->val[j];
00203             if (map[col] != 0x0F)
00204                 map[col] = 0x0F;
00205         }
00206     }

```



```

00207         else if ( val > 1e-20)
00208             map[col] = 0x09; /* bright blue */
00209         else if ( val < -1e-20)
00210             map[col] = 0x0C; /* bright red */
00211         else if (val == 0)
00212             map[col] = 0x00; /* bright red */
00213         else
00214             map[col] = 0x06; /* brown */
00215     } // for j
00216
00217     for (j = 0; j < size; ++j) {
00218         b <= 4;
00219         b |= (j < n ? map[j] & 15 : 0);
00220         if (j & 1) put_byte(fp, b);
00221     }
00222 }
00223
00224 fflush(fp);
00225 if (ferror(fp)) {
00226     printf("### ERROR: Write error on '%s'! [%s]\n", fname, __FUNCTION__);
00227 }
00228
00229 FINISH: if (fp != NULL) fclose(fp);
00230
00231 fasp_mem_free(map); map = NULL;
00232 }
00233
00259 void fasp_dbsr_subplot (const dBSRmat *A,
00260                        const char *filename,
00261                        int size)
00262 {
00263     INT m = A->ROW;
00264     INT n = A->COL;
00265     INT nb = A->nb;
00266     INT nb2 = nb*nb;
00267     INT offset;
00268     INT row, col, i, j, k, l, minmn=nb*MIN(m,n);
00269     REAL val;
00270     char *map;
00271
00272     if (size>minmn) size=minmn;
00273
00274     printf("Writing matrix pattern to '%s'...\n", filename);
00275
00276     map = (char *)fasp_mem_calloc(size * size, sizeof(char));
00277
00278     memset((void *)map, 0x0F, size * size);
00279
00280     for ( i = 0; i < size/nb; i++ ) {
00281
00282         for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
00283             for ( k = 0; k < A->nb; k++ ) {
00284                 for ( l = 0; l < A->nb; l++ ) {
00285
00286                     row = i*nb + k;
00287                     col = A->JA[j]*nb + l;
00288                     val = A->val[ A->JA[j]*nb2 + k*nb + l];
00289
00290                     if (col<size) {
00291
00292                         offset = size*row + col;
00293
00294                         if (map[offset] != 0x0F)
00295                             map[offset] = 0x0F;
00296                         else if ( val > 1e-20)
00297                             map[offset] = 0x09; /* bright blue */
00298                         else if ( val < -1e-20)
00299                             map[offset] = 0x0C; /* bright red */
00300                         else if (val == 0)
00301                             map[offset] = 0x00; /* bright red */
00302                         else
00303                             map[offset] = 0x06; /* brown */
00304                     } // end if
00305                 }
00306             }
00307         }
00308     }
00309
00310     write_bmp16(filename, size, size, map);
00311
00312     fasp_mem_free(map); map = NULL;

```

```

00313 }
00314
00339 void fasp_dbsr_plot (const dBSRmat *A,
00340                     const char *fname)
00341 {
00342     FILE *fp;
00343     INT offset, bmsize, i, j, b;
00344     INT size;
00345     INT nb = A->nb;
00346     INT nb2 = nb*nb;
00347     INT n = A->COL*A->nb, m = A->ROW*A->nb;
00348     INT col,k,l;
00349     REAL val;
00350     char *map;
00351
00352     size = ( (n+7)/8 ) * 8;
00353
00354     map = (char *)fasp_mem_calloc(size, sizeof(char));
00355
00356     memset((void *)map, 0x0F, size);
00357
00358     if (!(1 <= m && m <= 32767))
00359         printf("### ERROR: Invalid num of rows %d! [%s]\n", m, __FUNCTION__);
00360
00361     if (!(1 <= n && n <= 32767))
00362         printf("### ERROR: Invalid num of cols %d! [%s]\n", n, __FUNCTION__);
00363
00364     fp = fopen(fname, "wb");
00365     if (fp == NULL) {
00366         printf("### ERROR: Unable to create '%s'! [%s]\n", fname, __FUNCTION__);
00367         goto FINISH;
00368     }
00369
00370     offset = 14 + 40 + 16 * 4;
00371     bmsize = (4 * n + 31) / 32;
00372     /* struct BMPFILEHEADER (14 bytes) */
00373     /* UINT bfType */ put_byte(fp, 'B'); put_byte(fp, 'M');
00374     /* DWORD bfSize */ put_dword(fp, offset + bmsize * 4);
00375     /* UINT bfReserved1 */ put_word(fp, 0);
00376     /* UNIT bfReserved2 */ put_word(fp, 0);
00377     /* DWORD bOffBits */ put_dword(fp, offset);
00378     /* struct BMPINFOHEADER (40 bytes) */
00379     /* DWORD biSize */ put_dword(fp, 40);
00380     /* LONG biWidth */ put_dword(fp, n);
00381     /* LONG biHeight */ put_dword(fp, m);
00382     /* WORD biPlanes */ put_word(fp, 1);
00383     /* WORD biBitCount */ put_word(fp, 4);
00384     /* DWORD biCompression */ put_dword(fp, 0 /* BI_RGB */);
00385     /* DWORD biSizeImage */ put_dword(fp, 0);
00386     /* LONG biXPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
00387     /* LONG biYPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
00388     /* DWORD biClrUsed */ put_dword(fp, 0);
00389     /* DWORD biClrImportant */ put_dword(fp, 0);
00390     /* struct RGBQUAD (16 * 4 = 64 bytes) */
00391     /* CGA-compatible colors: */
00392     /* 0x00 = black */ put_dword(fp, 0x000000);
00393     /* 0x01 = blue */ put_dword(fp, 0x000080);
00394     /* 0x02 = green */ put_dword(fp, 0x008000);
00395     /* 0x03 = cyan */ put_dword(fp, 0x008080);
00396     /* 0x04 = red */ put_dword(fp, 0x800000);
00397     /* 0x05 = magenta */ put_dword(fp, 0x800080);
00398     /* 0x06 = brown */ put_dword(fp, 0x808000);
00399     /* 0x07 = light gray */ put_dword(fp, 0xC0C0C0);
00400     /* 0x08 = dark gray */ put_dword(fp, 0x808080);
00401     /* 0x09 = bright blue */ put_dword(fp, 0x0000FF);
00402     /* 0x0A = bright green */ put_dword(fp, 0x00FF00);
00403     /* 0x0B = bright cyan */ put_dword(fp, 0x00FFFF);
00404     /* 0x0C = bright red */ put_dword(fp, 0xFF0000);
00405     /* 0x0D = bright magenta */ put_dword(fp, 0xFF00FF);
00406     /* 0x0E = yellow */ put_dword(fp, 0xFFFF00);
00407     /* 0x0F = white */ put_dword(fp, 0xFFFFFF);
00408     /* pixel data bits */
00409     b = 0;
00410
00412     // for(i=size-1; i>=m; i--){
00413     //     memset(map, 0x0F, size);
00414     //     for (j = 0; j < size; ++j) {
00415     //         b <<= 4;
00416     //         b |= (j < n ? map[j] & 15 : 0);
00417     //         if (j & 1) put_byte(fp, b);
00418     //     }

```

```

00419 // }
00421
00422 for ( i = A->ROW-1; i >=0; i-- ) {
00423     for ( k = A->nb-1; k >=0; k-- ) {
00424
00425         memset(map, 0x0F, size);
00426
00427         for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
00428             for ( l = 0; l < A->nb; l++ ) {
00429
00430                 col = A->JA[j]*nb + l;
00431                 val = A->val[ A->JA[j]*nb2 + k*nb + l];
00432
00433                 if (map[col] != 0x0F)
00434                     map[col] = 0x0F;
00435                 else if ( val > 1e-20)
00436                     map[col] = 0x09; /* bright blue */
00437                 else if ( val < -1e-20)
00438                     map[col] = 0x0C; /* bright red */
00439                 else if (val == 0)
00440                     map[col] = 0x00; /* bright red */
00441                 else
00442                     map[col] = 0x06; /* brown */
00443             } // for l
00444         } // for j
00445
00446
00447         for (j = 0; j < size; ++j) {
00448             b <<= 4;
00449             b |= (j < n ? map[j] & 15 : 0);
00450             if (j & 1) put_byte(fp, b);
00451         }
00452     }
00453 }
00454
00455 }
00456
00457 fflush(fp);
00458 if (ferror(fp)) {
00459     printf("### ERROR: Write error on '%s'! [%s]\n", fname, __FUNCTION__);
00460 }
00461
00462 FINISH: if (fp != NULL) fclose(fp);
00463
00464 fasp_mem_free(map); map = NULL;
00465 }
00466
00478 void fasp_grid2d_plot (pgrid2d pg,
00479                       int level)
00480 {
00481     FILE *datei;
00482     char buf[120];
00483     INT i;
00484     REAL xmid,ymid,xc,yc;
00485
00486     sprintf(buf,"Grid_ref_level%d.eps",level);
00487     datei = fopen(buf,"w");
00488     if(datei==NULL) {
00489         printf("Opening file %s fails!\n", buf);
00490         return;
00491     }
00492
00493     fprintf(datei, "%%!PS-Adobe-2.0 EPSF-2.0\n");
00494     fprintf(datei, "%%BoundingBox: 0 0 550 550\n");
00495     fprintf(datei, "25 dup translate\n");
00496     fprintf(datei, "%f setlinewidth\n",0.2);
00497     fprintf(datei, "/Helvetica findfont %f scalefont setfont\n",64.0*pow(0.5,level));
00498     fprintf(datei, "/b{0 setgray} def\n");
00499     fprintf(datei, "/r{1.0 0.6 0.6 setrgbcolor} def\n");
00500     fprintf(datei, "/u{0.1 0.7 0.1 setrgbcolor} def\n");
00501     fprintf(datei, "/d{0.1 0.1 1.0 setrgbcolor} def\n");
00502     fprintf(datei, "/cs{closepath stroke} def\n");
00503     fprintf(datei, "/m{moveto} def\n");
00504     fprintf(datei, "/l{lineto} def\n");
00505
00506     fprintf(datei,"b\n");
00507     for (i=0; i<pg->triangles; ++i) {
00508         xc = (pg->p[pg->t[i][0]][0]+pg->p[pg->t[i][1]][0]+pg->p[pg->t[i][2]][0])*150.0;
00509         yc = (pg->p[pg->t[i][0]][1]+pg->p[pg->t[i][1]][1]+pg->p[pg->t[i][2]][1])*150.0;
00510
00511         xmid = pg->p[pg->t[i][0]][0]*450.0;

```

```

00512         ymid = pg->p[pg->t[i][0]][1]*450.0;
00513         fprintf(datei,"%1f %1f m ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
00514         xmid = pg->p[pg->t[i][1]][0]*450.0;
00515         ymid = pg->p[pg->t[i][1]][1]*450.0;
00516         fprintf(datei,"%1f %1f l ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
00517         xmid = pg->p[pg->t[i][2]][0]*450.0;
00518         ymid = pg->p[pg->t[i][2]][1]*450.0;
00519         fprintf(datei,"%1f %1f l ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
00520         fprintf(datei,"cs\n");
00521     }
00522     fprintf(datei,"r\n");
00523     for(i=0; i<pg->vertices; ++i) {
00524         xmid = pg->p[i][0]*450.0;
00525         ymid = pg->p[i][1]*450.0;
00526         fprintf(datei,"%1f %1f m ",xmid,ymid);
00527         fprintf(datei,"(%d) show\n ",i);
00528     }
00529     fprintf(datei,"u\n");
00530     for(i=0; i<pg->edges; ++i) {
00531         xmid = 0.5*(pg->p[pg->e[i][0]][0]+pg->p[pg->e[i][1]][0])*450.0;
00532         ymid = 0.5*(pg->p[pg->e[i][0]][1]+pg->p[pg->e[i][1]][1])*450.0;
00533         fprintf(datei,"%1f %1f m ",xmid,ymid);
00534         fprintf(datei,"(%d) show\n ",i);
00535     }
00536     xmid = pg->p[pg->e[i][0]][0]*450.0;
00537     ymid = pg->p[pg->e[i][0]][1]*450.0;
00538     fprintf(datei,"%1f %1f m ",xmid,ymid);
00539     xmid = pg->p[pg->e[i][1]][0]*450.0;
00540     ymid = pg->p[pg->e[i][1]][1]*450.0;
00541     fprintf(datei,"%1f %1f l ",xmid,ymid);
00542     fprintf(datei,"cs\n");
00543 }
00544 fprintf(datei,"d\n");
00545 for(i=0; i<pg->triangles; ++i) {
00546     xmid = (pg->p[pg->t[i][0]][0]+pg->p[pg->t[i][1]][0]+pg->p[pg->t[i][2]][0])*150.0;
00547     ymid = (pg->p[pg->t[i][0]][1]+pg->p[pg->t[i][1]][1]+pg->p[pg->t[i][2]][1])*150.0;
00548     fprintf(datei,"%1f %1f m ",xmid,ymid);
00549     fprintf(datei,"(%d) show\n ",i);
00550 }
00551 fprintf(datei, "showpage\n");
00552 fclose(datei);
00553 }
00554
00555 /*-----*/
00556 /*--      Private Functions      --*/
00557 /*-----*/
00558
00568 static void put_byte (FILE          *fp,
00569                     const int      c)
00570 {
00571     fputc(c, fp);
00572     return;
00573 }
00574
00584 static void put_word (FILE          *fp,
00585                     const int      w)
00586 { /* big endian */
00587     put_byte(fp, w);
00588     put_byte(fp, w >> 8);
00589     return;
00590 }
00591
00601 static void put_dword (FILE          *fp,
00602                     const int      d)
00603 { /* big endian */
00604     put_word(fp, d);
00605     put_word(fp, d >> 16);
00606     return;
00607 }
00608
00674 static int write_bmp16 (const char  *fname,
00675                     const int      m,
00676                     const int      n,
00677                     const char     map[])
00678 {
00679     FILE *fp;
00680     int offset, bmsize, i, j, b, ret = 1;
00681
00682     if (!(1 <= m && m <= 32767))
00683         printf("### ERROR: %s invalid height %d\n", __FUNCTION__, m);
00684

```

```

00685     if (!(1 <= n && n <= 32767))
00686         printf("### ERROR: %s invalid width %d\n", __FUNCTION__, n);
00687
00688     fp = fopen(fname, "wb");
00689     if (fp == NULL) {
00690         printf("### ERROR: %s unable to create '%s'\n", __FUNCTION__, fname);
00691         ret = 0;
00692         goto FINISH;
00693     }
00694     offset = 14 + 40 + 16 * 4;
00695     bmsize = (4 * n + 31) / 32;
00696     /* struct BMPFILEHEADER (14 bytes) */
00697     /* UINT bfType */          put_byte(fp, 'B'); put_byte(fp, 'M');
00698     /* DWORD bfSize */        put_dword(fp, offset + bmsize * 4);
00699     /* UINT bfReserved1 */     put_word(fp, 0);
00700     /* UINT bfReserved2 */     put_word(fp, 0);
00701     /* DWORD bfOffBits */      put_dword(fp, offset);
00702     /* struct BMPINFOHEADER (40 bytes) */
00703     /* DWORD biSize */         put_dword(fp, 40);
00704     /* LONG biWidth */         put_dword(fp, n);
00705     /* LONG biHeight */        put_dword(fp, m);
00706     /* WORD biPlanes */        put_word(fp, 1);
00707     /* WORD biBitCount */      put_word(fp, 4);
00708     /* DWORD biCompression */  put_dword(fp, 0 /* BI_RGB */);
00709     /* DWORD biSizeImage */     put_dword(fp, 0);
00710     /* LONG biXPelsPerMeter */  put_dword(fp, 2953 /* 75 dpi */);
00711     /* LONG biYPelsPerMeter */  put_dword(fp, 2953 /* 75 dpi */);
00712     /* DWORD biClrUsed */      put_dword(fp, 0);
00713     /* DWORD biClrImportant */  put_dword(fp, 0);
00714     /* struct RGBQUAD (16 * 4 = 64 bytes) */
00715     /* CGA-compatible colors: */
00716     /* 0x00 = black */         put_dword(fp, 0x000000);
00717     /* 0x01 = blue */          put_dword(fp, 0x000080);
00718     /* 0x02 = green */         put_dword(fp, 0x008000);
00719     /* 0x03 = cyan */          put_dword(fp, 0x008080);
00720     /* 0x04 = red */           put_dword(fp, 0x800000);
00721     /* 0x05 = magenta */       put_dword(fp, 0x800080);
00722     /* 0x06 = brown */         put_dword(fp, 0x808000);
00723     /* 0x07 = light gray */    put_dword(fp, 0xC0C0C0);
00724     /* 0x08 = dark gray */     put_dword(fp, 0x808080);
00725     /* 0x09 = bright blue */   put_dword(fp, 0x0000FF);
00726     /* 0x0A = bright green */   put_dword(fp, 0x00FF00);
00727     /* 0x0B = bright cyan */   put_dword(fp, 0x00FFFF);
00728     /* 0x0C = bright red */     put_dword(fp, 0xFF0000);
00729     /* 0x0D = bright magenta */ put_dword(fp, 0xFF00FF);
00730     /* 0x0E = yellow */        put_dword(fp, 0xFFFF00);
00731     /* 0x0F = white */         put_dword(fp, 0xFFFFFF);
00732     /* pixel data bits */
00733     b = 0;
00734     for (i = m - 1; i >= 0; i--) {
00735         for (j = 0; j < ((n + 7) / 8) * 8; ++j) {
00736             b <<= 4;
00737             b |= (j < n ? map[i * n + j] & 15 : 0);
00738             if (j & 1) put_byte(fp, b);
00739         }
00740     }
00741     fflush(fp);
00742
00743     if (ferror(fp)) {
00744         printf("### ERROR: %s write error on '%s'\n", __FUNCTION__, fname);
00745         ret = 0;
00746     }
00747
00748 FINISH: if (fp != NULL) fclose(fp);
00749     return ret;
00750 }
00751
00752 /*-----*/
00753 /*--          End of File          --*/
00754 /*-----*/

```

9.29 AuxInput.c File Reference

Read and check input parameters.

```

#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- [SHORT fasp_param_check](#) ([input_param](#) *inparam)
Simple check on input parameters.
- void [fasp_param_input](#) (const char *fname, [input_param](#) *inparam)
Read input parameters from disk file.

9.29.1 Detailed Description

Read and check input parameters.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxMemory.c](#) and [AuxMessage.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxInput.c](#).

9.29.2 Function Documentation

9.29.2.1 fasp_param_check()

```
SHORT fasp_param_check (
    input_param * inparam )
```

Simple check on input parameters.

Parameters

<i>inparam</i>	Input parameters
----------------	------------------

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Chensong Zhang

Date

09/29/2013

Definition at line 33 of file [AuxInput.c](#).

9.29.2.2 fasp_param_input()

```
void fasp_param_input (
    const char * fname,
    input_param * inparam )
```

Read input parameters from disk file.

Parameters

<i>fname</i>	File name for input file
<i>inparam</i>	Input parameters

Author

Chensong Zhang

Date

03/20/2010

Modified by Xiaozhe Hu on 01/23/2011: add AMLI cycle; Modified by Chensong Zhang on 05/10/2013: add a new input;
 Modified by Chensong Zhang on 03/23/2015: skip unknown keyword; Modified by Chensong Zhang on 03/27/2017:
 check unexpected error; Modified by Chensong Zhang on 09/20/2017: new skip the line;
 Definition at line 112 of file [AuxInput.c](#).

9.30 AuxInput.c

[Go to the documentation of this file.](#)

```

00001
00014 #include "fasp.h"
00015 #include "fasp_funcs.h"
00016
00017 /*-----*/
00018 /*--      Public Functions      --*/
00019 /*-----*/
00020
00033 SHORT fasp_param_check (input_param *inparam)
00034 {
00035     SHORT status = FASP_SUCCESS;
00036
00037     if ( inparam->problem_num<0
00038         || inparam->solver_type<0
00039         || inparam->solver_type>50
00040         || inparam->precond_type<0
00041         || inparam->decoup_type<0
00042         || inparam->itsolver_tol<0
00043         || inparam->itsolver_maxit<0
00044         || inparam->stop_type<=0
00045         || inparam->stop_type>3
00046         || inparam->restart<0
00047         || inparam->ILU_type<=0
00048         || inparam->ILU_type>3
00049         || inparam->ILU_lfil<0
00050         || inparam->ILU_droptol<=0
00051         || inparam->ILU_relax<0
00052         || inparam->ILU_permtol<0
00053         || inparam->SWZ_mmsize<0
00054         || inparam->SWZ_maxlvl<0
00055         || inparam->SWZ_type<0
00056         || inparam->SWZ_blksolver<0
00057         || inparam->AMG_type<=0
00058         || inparam->AMG_type>3
00059         || inparam->AMG_cycle_type<=0
00060         || inparam->AMG_levels<0
00061         || inparam->AMG_ILU_levels<0
00062         || inparam->AMG_coarse_dof<=0
00063         || inparam->AMG_tol<0
00064         || inparam->AMG_maxit<0
00065         || inparam->AMG_coarsening_type<=0
00066         || inparam->AMG_coarsening_type>4
00067         || inparam->AMG_coarse_solver<0
00068         || inparam->AMG_interpolation_type<0
00069         || inparam->AMG_interpolation_type>5
00070         || inparam->AMG_smoother<0
00071         || inparam->AMG_smoother>20
00072         || inparam->AMG_strong_threshold<0.0
00073         || inparam->AMG_strong_threshold>0.9999

```



```

00074         || inparam->AMG_truncation_threshold<0.0
00075         || inparam->AMG_truncation_threshold>0.9999
00076         || inparam->AMG_max_row_sum<0.0
00077         || inparam->AMG_presmooth_iter<0
00078         || inparam->AMG_postsmooth_iter<0
00079         || inparam->AMG_amli_degree<0
00080         || inparam->AMG_aggressive_level<0
00081         || inparam->AMG_aggressive_path<0
00082         || inparam->AMG_aggregation_type<0
00083         || inparam->AMG_pair_number<0
00084         || inparam->AMG_strong_coupled<0
00085         || inparam->AMG_max_aggregation<=0
00086         || inparam->AMG_tentative_smooth<0
00087         || inparam->AMG_smooth_filter<0
00088         || inparam->AMG_smooth_restriction<0
00089         || inparam->AMG_smooth_restriction>1
00090     ) status = ERROR_INPUT_PAR;
00091
00092     return status;
00093 }
00094
00112 void fasp_param_input (const char *fname,
00113                       input_param *inparam)
00114 {
00115     char    buffer[STRLEN]; // Note: max number of char for each line!
00116     int     val;
00117     SHORT   status = FASP_SUCCESS;
00118     FILE    *fp;
00119
00120     // set default input parameters
00121     fasp_param_input_init(inparam);
00122
00123     // if input file is not specified, use the default values
00124     if (fname==NULL) return;
00125
00126     fp = fopen(fname,"r");
00127     if (fp==NULL) fasp_chkerr(ERROR_OPEN_FILE, __FUNCTION__);
00128
00129     while ( status == FASP_SUCCESS ) {
00130         int     ibuff;
00131         double  dbuff;
00132         char    sbuff[STRLEN];
00133
00134         val = fscanf(fp,"%s",buffer);
00135         if (val==EOF) break;
00136         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00137         if (buffer[0]==' ' || buffer[0]=='%' || buffer[0]=='|') {
00138             if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00139             continue;
00140         }
00141
00142         // match keyword and scan for value
00143         if (strcmp(buffer,"workdir")==0) {
00144             val = fscanf(fp,"%s",buffer);
00145             if (val!=1 || strcmp(buffer,"")!=0) {
00146                 status = ERROR_INPUT_PAR; break;
00147             }
00148             val = fscanf(fp,"%s",sbuff);
00149             if (val!=1) { status = ERROR_INPUT_PAR; break; }
00150             memcpy(inparam->workdir,sbuff,STRLEN);
00151             if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00152         }
00153
00154         else if (strcmp(buffer,"problem_num")==0) {
00155             val = fscanf(fp,"%s",buffer);
00156             if (val!=1 || strcmp(buffer,"")!=0) {
00157                 status = ERROR_INPUT_PAR; break;
00158             }
00159             val = fscanf(fp,"%d",&ibuff);
00160             if (val!=1) { status = ERROR_INPUT_PAR; break; }
00161             inparam->problem_num=ibuff;
00162             if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00163         }
00164
00165         else if (strcmp(buffer,"print_level")==0) {
00166             val = fscanf(fp,"%s",buffer);
00167             if (val!=1 || strcmp(buffer,"")!=0) {
00168                 status = ERROR_INPUT_PAR; break;
00169             }
00170             val = fscanf(fp,"%d",&ibuff);
00171             if (val!=1) { status = ERROR_INPUT_PAR; break; }

```

```

00172         inparam->print_level = ibuff;
00173         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00174     }
00175
00176     else if (strcmp(buffer,"output_type")==0) {
00177         val = fscanf(fp,"%s",buffer);
00178         if (val!=1 || strcmp(buffer,"")!=0) {
00179             status = ERROR_INPUT_PAR; break;
00180         }
00181         val = fscanf(fp,"%d",&ibuff);
00182         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00183         inparam->output_type = ibuff;
00184         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00185     }
00186
00187     else if (strcmp(buffer,"solver_type")==0) {
00188         val = fscanf(fp,"%s",buffer);
00189         if (val!=1 || strcmp(buffer,"")!=0) {
00190             status = ERROR_INPUT_PAR; break;
00191         }
00192         val = fscanf(fp,"%d",&ibuff);
00193         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00194         inparam->solver_type = ibuff;
00195         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00196     }
00197
00198     else if (strcmp(buffer,"stop_type")==0) {
00199         val = fscanf(fp,"%s",buffer);
00200         if (val!=1 || strcmp(buffer,"")!=0) {
00201             status = ERROR_INPUT_PAR; break;
00202         }
00203         val = fscanf(fp,"%d",&ibuff);
00204         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00205         inparam->stop_type = ibuff;
00206         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00207     }
00208
00209     else if (strcmp(buffer,"decoup_type")==0) {
00210         val = fscanf(fp,"%s",buffer);
00211         if (val!=1 || strcmp(buffer,"")!=0) {
00212             status = ERROR_INPUT_PAR; break;
00213         }
00214         val = fscanf(fp,"%d",&ibuff);
00215         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00216         inparam->decoup_type = ibuff;
00217         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00218     }
00219
00220     else if (strcmp(buffer,"precond_type")==0) {
00221         val = fscanf(fp,"%s",buffer);
00222         if (val!=1 || strcmp(buffer,"")!=0) {
00223             status = ERROR_INPUT_PAR; break;
00224         }
00225         val = fscanf(fp,"%d",&ibuff);
00226         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00227         inparam->precond_type = ibuff;
00228         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00229     }
00230
00231     else if (strcmp(buffer,"itsolver_tol")==0) {
00232         val = fscanf(fp,"%s",buffer);
00233         if (val!=1 || strcmp(buffer,"")!=0) {
00234             status = ERROR_INPUT_PAR; break;
00235         }
00236         val = fscanf(fp,"%lf",&dbuff);
00237         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00238         inparam->itsolver_tol = dbuff;
00239         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00240     }
00241
00242     else if (strcmp(buffer,"itsolver_maxit")==0) {
00243         val = fscanf(fp,"%s",buffer);
00244         if (val!=1 || strcmp(buffer,"")!=0) {
00245             status = ERROR_INPUT_PAR; break;
00246         }
00247         val = fscanf(fp,"%d",&ibuff);
00248         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00249         inparam->itsolver_maxit = ibuff;
00250         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00251     }
00252

```

```

00253     else if (strcmp(buffer,"AMG_ILU_levels")==0) {
00254         val = fscanf(fp,"%s",buffer);
00255         if (val!=1 || strcmp(buffer,"")!=0) {
00256             status = ERROR_INPUT_PAR; break;
00257         }
00258         val = fscanf(fp,"%d",&ibuff);
00259         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00260         inparam->AMG_ILU_levels = ibuff;
00261         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00262     }
00263
00264     else if (strcmp(buffer,"AMG_SWZ_levels")==0) {
00265         val = fscanf(fp,"%s",buffer);
00266         if (val!=1 || strcmp(buffer,"")!=0) {
00267             status = ERROR_INPUT_PAR; break;
00268         }
00269         val = fscanf(fp,"%d",&ibuff);
00270         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00271         inparam->AMG_SWZ_levels = ibuff;
00272         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00273     }
00274
00275     else if (strcmp(buffer,"itsolver_restart")==0) {
00276         val = fscanf(fp,"%s",buffer);
00277         if (val!=1 || strcmp(buffer,"")!=0) {
00278             status = ERROR_INPUT_PAR; break;
00279         }
00280         val = fscanf(fp,"%d",&ibuff);
00281         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00282         inparam->restart = ibuff;
00283         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00284     }
00285
00286     else if (strcmp(buffer,"AMG_type")==0) {
00287         val = fscanf(fp,"%s",buffer);
00288         if (val!=1 || strcmp(buffer,"")!=0) {
00289             status = ERROR_INPUT_PAR; break;
00290         }
00291         val = fscanf(fp,"%s",buffer);
00292         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00293
00294         if ((strcmp(buffer,"C")==0) || (strcmp(buffer,"c")==0))
00295             inparam->AMG_type = CLASSIC_AMG;
00296         else if ((strcmp(buffer,"SA")==0) || (strcmp(buffer,"sa")==0))
00297             inparam->AMG_type = SA_AMG;
00298         else if ((strcmp(buffer,"UA")==0) || (strcmp(buffer,"ua")==0))
00299             inparam->AMG_type = UA_AMG;
00300         else
00301             { status = ERROR_INPUT_PAR; break; }
00302         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00303     }
00304
00305     else if (strcmp(buffer,"AMG_strong_coupled")==0) {
00306         val = fscanf(fp,"%s",buffer);
00307         if (val!=1 || strcmp(buffer,"")!=0) {
00308             status = ERROR_INPUT_PAR; break;
00309         }
00310         val = fscanf(fp,"%lf",&dbuff);
00311         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00312         inparam->AMG_strong_coupled = dbuff;
00313         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00314     }
00315
00316     else if (strcmp(buffer,"AMG_max_aggregation")==0) {
00317         val = fscanf(fp,"%s",buffer);
00318         if (val!=1 || strcmp(buffer,"")!=0) {
00319             status = ERROR_INPUT_PAR; break;
00320         }
00321         val = fscanf(fp,"%d",&ibuff);
00322         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00323         inparam->AMG_max_aggregation = ibuff;
00324         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00325     }
00326
00327     else if (strcmp(buffer,"AMG_tentative_smooth")==0) {
00328         val = fscanf(fp,"%s",buffer);
00329         if (val!=1 || strcmp(buffer,"")!=0) {
00330             status = ERROR_INPUT_PAR; break;
00331         }
00332         val = fscanf(fp,"%lf",&dbuff);
00333         if (val!=1) { status = ERROR_INPUT_PAR; break; }

```

```

00334         inparam->AMG_tentative_smooth = dbuff;
00335         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00336     }
00337
00338     else if (strcmp(buffer,"AMG_smooth_filter")==0) {
00339         val = fscanf(fp,"%s",buffer);
00340         if (val!=1 || strcmp(buffer,"")!=0) {
00341             status = ERROR_INPUT_PAR; break;
00342         }
00343         val = fscanf(fp,"%s",buffer);
00344         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00345
00346         if ((strcmp(buffer,"ON")==0) || (strcmp(buffer,"on")==0) ||
00347             (strcmp(buffer,"On")==0) || (strcmp(buffer,"oN")==0)) {
00348             inparam->AMG_smooth_filter = ON;
00349         }
00350         else if ((strcmp(buffer,"OFF")==0) || (strcmp(buffer,"off")==0) ||
00351             (strcmp(buffer,"oFF")==0) || (strcmp(buffer,"OFF")==0) ||
00352             (strcmp(buffer,"Off")==0) || (strcmp(buffer,"oFF")==0) ||
00353             (strcmp(buffer,"OfF")==0) || (strcmp(buffer,"OFF")==0)) {
00354             inparam->AMG_smooth_filter = OFF;
00355         }
00356         else
00357             { status = ERROR_INPUT_PAR; break; }
00358         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00359     }
00360
00361     else if (strcmp(buffer,"AMG_smooth_restriction")==0) {
00362         val = fscanf(fp,"%s",buffer);
00363         if (val!=1 || strcmp(buffer,"")!=0) {
00364             status = ERROR_INPUT_PAR; break;
00365         }
00366         val = fscanf(fp,"%s",buffer);
00367         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00368
00369         if ((strcmp(buffer,"ON")==0) || (strcmp(buffer,"on")==0) ||
00370             (strcmp(buffer,"On")==0) || (strcmp(buffer,"oN")==0)) {
00371             inparam->AMG_smooth_restriction = ON;
00372         }
00373         else if ((strcmp(buffer,"OFF")==0) || (strcmp(buffer,"off")==0) ||
00374             (strcmp(buffer,"oFF")==0) || (strcmp(buffer,"OFF")==0) ||
00375             (strcmp(buffer,"Off")==0) || (strcmp(buffer,"oFF")==0) ||
00376             (strcmp(buffer,"OfF")==0) || (strcmp(buffer,"OFF")==0)) {
00377             inparam->AMG_smooth_restriction = OFF;
00378         }
00379         else
00380             { status = ERROR_INPUT_PAR; break; }
00381         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00382     }
00383
00384     else if (strcmp(buffer,"AMG_coarse_solver")==0) {
00385         val = fscanf(fp,"%s",buffer);
00386         if (val!=1 || strcmp(buffer,"")!=0) {
00387             status = ERROR_INPUT_PAR; break;
00388         }
00389         val = fscanf(fp,"%d",&ibuff);
00390         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00391         inparam->AMG_coarse_solver = ibuff;
00392         if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00393     }
00394
00395     else if (strcmp(buffer,"AMG_coarse_scaling")==0) {
00396         val = fscanf(fp,"%s",buffer);
00397         if (val!=1 || strcmp(buffer,"")!=0) {
00398             status = ERROR_INPUT_PAR; break;
00399         }
00400         val = fscanf(fp,"%s",buffer);
00401         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00402
00403         if ((strcmp(buffer,"ON")==0) || (strcmp(buffer,"on")==0) ||
00404             (strcmp(buffer,"On")==0) || (strcmp(buffer,"oN")==0)) {
00405             inparam->AMG_coarse_scaling = ON;
00406         }
00407         else if ((strcmp(buffer,"OFF")==0) || (strcmp(buffer,"off")==0) ||
00408             (strcmp(buffer,"oFF")==0) || (strcmp(buffer,"OFF")==0) ||
00409             (strcmp(buffer,"Off")==0) || (strcmp(buffer,"oFF")==0) ||
00410             (strcmp(buffer,"OfF")==0) || (strcmp(buffer,"OFF")==0)) {
00411             inparam->AMG_coarse_scaling = OFF;
00412         }
00413         else
00414             { status = ERROR_INPUT_PAR; break; }

```

```

00415         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00416     }
00417
00418     else if (strcmp(buffer,"AMG_levels")==0) {
00419         val = fscanf(fp,"%s",buffer);
00420         if (val!=1 || strcmp(buffer,"")!=0) {
00421             status = ERROR_INPUT_PAR; break;
00422         }
00423         val = fscanf(fp,"%d",&ibuff);
00424         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00425         inparam->AMG_levels = ibuff;
00426         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00427     }
00428
00429     else if (strcmp(buffer,"AMG_tol")==0) {
00430         val = fscanf(fp,"%s",buffer);
00431         if (val!=1 || strcmp(buffer,"")!=0) {
00432             status = ERROR_INPUT_PAR; break;
00433         }
00434         val = fscanf(fp,"%lf",&dbuff);
00435         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00436         inparam->AMG_tol = dbuff;
00437         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00438     }
00439
00440     else if (strcmp(buffer,"AMG_maxit")==0) {
00441         val = fscanf(fp,"%s",buffer);
00442         if (val!=1 || strcmp(buffer,"")!=0) {
00443             status = ERROR_INPUT_PAR; break;
00444         }
00445         val = fscanf(fp,"%d",&ibuff);
00446         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00447         inparam->AMG_maxit = ibuff;
00448         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00449     }
00450
00451     else if (strcmp(buffer,"AMG_coarse_dof")==0) {
00452         val = fscanf(fp,"%s",buffer);
00453         if (val!=1 || strcmp(buffer,"")!=0) {
00454             status = ERROR_INPUT_PAR; break;
00455         }
00456         val = fscanf(fp,"%d",&ibuff);
00457         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00458         inparam->AMG_coarse_dof = ibuff;
00459         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00460     }
00461
00462     else if (strcmp(buffer,"AMG_cycle_type")==0) {
00463         val = fscanf(fp,"%s",buffer);
00464         if (val!=1 || strcmp(buffer,"")!=0) {
00465             status = ERROR_INPUT_PAR; break;
00466         }
00467         val = fscanf(fp,"%s",buffer);
00468         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00469
00470         if ((strcmp(buffer,"V")==0) || (strcmp(buffer,"v")==0))
00471             inparam->AMG_cycle_type = V_CYCLE;
00472         else if ((strcmp(buffer,"W")==0) || (strcmp(buffer,"w")==0))
00473             inparam->AMG_cycle_type = W_CYCLE;
00474         else if ((strcmp(buffer,"A")==0) || (strcmp(buffer,"a")==0))
00475             inparam->AMG_cycle_type = AMLI_CYCLE;
00476         else if ((strcmp(buffer,"NA")==0) || (strcmp(buffer,"na")==0))
00477             inparam->AMG_cycle_type = NL_AMLI_CYCLE;
00478         else if ((strcmp(buffer,"VW")==0) || (strcmp(buffer,"vw")==0))
00479             inparam->AMG_cycle_type = VW_CYCLE;
00480         else if ((strcmp(buffer,"WV")==0) || (strcmp(buffer,"wv")==0))
00481             inparam->AMG_cycle_type = WV_CYCLE;
00482         else
00483             { status = ERROR_INPUT_PAR; break; }
00484         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00485     }
00486
00487     else if (strcmp(buffer,"AMG_smoother")==0) {
00488         val = fscanf(fp,"%s",buffer);
00489         if (val!=1 || strcmp(buffer,"")!=0) {
00490             status = ERROR_INPUT_PAR; break;
00491         }
00492         val = fscanf(fp,"%s",buffer);
00493         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00494
00495         if ((strcmp(buffer,"JACOBI")==0) || (strcmp(buffer,"jacobi")==0))

```

```

00496         inparam->AMG_smoother = SMOOTHER_JACOBI;
00497     else if ((strcmp(buffer, "GS")==0) || (strcmp(buffer, "gs")==0))
00498         inparam->AMG_smoother = SMOOTHER_GS;
00499     else if ((strcmp(buffer, "SGS")==0) || (strcmp(buffer, "sgs")==0))
00500         inparam->AMG_smoother = SMOOTHER_SGS;
00501     else if ((strcmp(buffer, "CG")==0) || (strcmp(buffer, "cg")==0))
00502         inparam->AMG_smoother = SMOOTHER_CG;
00503     else if ((strcmp(buffer, "SOR")==0) || (strcmp(buffer, "sor")==0))
00504         inparam->AMG_smoother = SMOOTHER_SOR;
00505     else if ((strcmp(buffer, "SSOR")==0) || (strcmp(buffer, "ssor")==0))
00506         inparam->AMG_smoother = SMOOTHER_SSOR;
00507     else if ((strcmp(buffer, "GSOR")==0) || (strcmp(buffer, "gsor")==0))
00508         inparam->AMG_smoother = SMOOTHER_GSOR;
00509     else if ((strcmp(buffer, "SGSOR")==0) || (strcmp(buffer, "sgsor")==0))
00510         inparam->AMG_smoother = SMOOTHER_SGSOR;
00511     else if ((strcmp(buffer, "POLY")==0) || (strcmp(buffer, "poly")==0))
00512         inparam->AMG_smoother = SMOOTHER_POLY;
00513     else if ((strcmp(buffer, "L1DIAG")==0) || (strcmp(buffer, "l1diag")==0))
00514         inparam->AMG_smoother = SMOOTHER_L1DIAG;
00515     else if ((strcmp(buffer, "BLKOIL")==0) || (strcmp(buffer, "blkoil")==0))
00516         inparam->AMG_smoother = SMOOTHER_BLKOil;
00517     else if ((strcmp(buffer, "SPETEN")==0) || (strcmp(buffer, "speten")==0))
00518         inparam->AMG_smoother = SMOOTHER_SPETEN;
00519     else
00520     {
00521         status = ERROR_INPUT_PAR; break;
00522     }
00523     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00524 }
00525
00526 else if (strcmp(buffer, "AMG_smooth_order")==0) {
00527     val = fscanf(fp, "%s", buffer);
00528     if (val!=1 || strcmp(buffer, "=")!=0) {
00529         status = ERROR_INPUT_PAR; break;
00530     }
00531     val = fscanf(fp, "%s", buffer);
00532     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00533
00534     if ((strcmp(buffer, "NO")==0) || (strcmp(buffer, "no")==0))
00535         inparam->AMG_smooth_order = NO_ORDER;
00536     else if ((strcmp(buffer, "CF")==0) || (strcmp(buffer, "cf")==0))
00537         inparam->AMG_smooth_order = CF_ORDER;
00538     else
00539     {
00540         status = ERROR_INPUT_PAR; break;
00541     }
00542     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00543 }
00544
00545 else if (strcmp(buffer, "AMG_coarsening_type")==0) {
00546     val = fscanf(fp, "%s", buffer);
00547     if (val!=1 || strcmp(buffer, "=")!=0) {
00548         status = ERROR_INPUT_PAR; break;
00549     }
00550     val = fscanf(fp, "%d", &ibuff);
00551     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00552     inparam->AMG_coarsening_type = ibuff;
00553     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00554 }
00555
00556 else if (strcmp(buffer, "AMG_interpolation_type")==0) {
00557     val = fscanf(fp, "%s", buffer);
00558     if (val!=1 || strcmp(buffer, "=")!=0) {
00559         status = ERROR_INPUT_PAR; break;
00560     }
00561     val = fscanf(fp, "%d", &ibuff);
00562     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00563     inparam->AMG_interpolation_type = ibuff;
00564     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00565 }
00566
00567 else if (strcmp(buffer, "AMG_aggregation_type")==0) {
00568     val = fscanf(fp, "%s", buffer);
00569     if (val!=1 || strcmp(buffer, "=")!=0) {
00570         status = ERROR_INPUT_PAR; break;
00571     }
00572     val = fscanf(fp, "%d", &ibuff);
00573     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00574     inparam->AMG_aggregation_type = ibuff;
00575     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00576 }
00577
00578 else if (strcmp(buffer, "AMG_pair_number")==0) {
00579     val = fscanf(fp, "%s", buffer);
00580     if (val!=1 || strcmp(buffer, "=")!=0) {

```

```

00577         status = ERROR_INPUT_PAR; break;
00578     }
00579     val = fscanf(fp,"%d",&ibuff);
00580     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00581     inparam->AMG_pair_number = ibuff;
00582     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00583 }
00584
00585 else if (strcmp(buffer,"AMG_quality_bound")==0) {
00586     val = fscanf(fp,"%s",buffer);
00587     if (val!=1 || strcmp(buffer,"")!=0) {
00588         status = ERROR_INPUT_PAR; break;
00589     }
00590     val = fscanf(fp,"%lf",&dbuff);
00591     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00592     inparam->AMG_quality_bound = dbuff;
00593     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00594 }
00595
00596 else if (strcmp(buffer,"AMG_aggressive_level")==0) {
00597     val = fscanf(fp,"%s",buffer);
00598     if (val!=1 || strcmp(buffer,"")!=0) {
00599         status = ERROR_INPUT_PAR; break;
00600     }
00601     val = fscanf(fp,"%d",&ibuff);
00602     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00603     inparam->AMG_aggressive_level = ibuff;
00604     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00605 }
00606
00607 else if (strcmp(buffer,"AMG_aggressive_path")==0) {
00608     val = fscanf(fp,"%s",buffer);
00609     if (val!=1 || strcmp(buffer,"")!=0) {
00610         status = ERROR_INPUT_PAR; break;
00611     }
00612     val = fscanf(fp,"%d",&ibuff);
00613     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00614     inparam->AMG_aggressive_path = ibuff;
00615     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00616 }
00617
00618 else if (strcmp(buffer,"AMG_presmooth_iter")==0) {
00619     val = fscanf(fp,"%s",buffer);
00620     if (val!=1 || strcmp(buffer,"")!=0) {
00621         status = ERROR_INPUT_PAR; break;
00622     }
00623     val = fscanf(fp,"%d",&ibuff);
00624     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00625     inparam->AMG_presmooth_iter = ibuff;
00626     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00627 }
00628
00629 else if (strcmp(buffer,"AMG_postsmooth_iter")==0) {
00630     val = fscanf(fp,"%s",buffer);
00631     if (val!=1 || strcmp(buffer,"")!=0) {
00632         status = ERROR_INPUT_PAR; break;
00633     }
00634     val = fscanf(fp,"%d",&ibuff);
00635     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00636     inparam->AMG_postsmooth_iter = ibuff;
00637     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00638 }
00639
00640 else if (strcmp(buffer,"AMG_relaxation")==0) {
00641     val = fscanf(fp,"%s",buffer);
00642     if (val!=1 || strcmp(buffer,"")!=0) {
00643         status = ERROR_INPUT_PAR; break;
00644     }
00645     val = fscanf(fp,"%lf",&dbuff);
00646     if (val!=1) { status = ERROR_INPUT_PAR; break; }
00647     inparam->AMG_relaxation=dbuff;
00648     if (fscanf(fp, "%*[^\\n]") { /* skip rest of line and do nothing */ };
00649 }
00650
00651 else if (strcmp(buffer,"AMG_polynomial_degree")==0) {
00652     val = fscanf(fp,"%s",buffer);
00653     if (val!=1 || strcmp(buffer,"")!=0) {
00654         status = ERROR_INPUT_PAR; break;
00655     }
00656     val = fscanf(fp,"%d",&ibuff);
00657     if (val!=1) { status = ERROR_INPUT_PAR; break; }

```

```

00658         inparam->AMG_polynomial_degree = ibuff;
00659         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00660     }
00661
00662     else if (strcmp(buffer,"AMG_strong_threshold")==0) {
00663         val = fscanf(fp,"%s",buffer);
00664         if (val!=1 || strcmp(buffer,"")!=0) {
00665             status = ERROR_INPUT_PAR; break;
00666         }
00667         val = fscanf(fp,"%lf",&dbuff);
00668         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00669         inparam->AMG_strong_threshold = dbuff;
00670         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00671     }
00672
00673     else if (strcmp(buffer,"AMG_truncation_threshold")==0) {
00674         val = fscanf(fp,"%s",buffer);
00675         if (val!=1 || strcmp(buffer,"")!=0) {
00676             status = ERROR_INPUT_PAR; break;
00677         }
00678         val = fscanf(fp,"%lf",&dbuff);
00679         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00680         inparam->AMG_truncation_threshold = dbuff;
00681         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00682     }
00683
00684     else if (strcmp(buffer,"AMG_max_row_sum")==0) {
00685         val = fscanf(fp,"%s",buffer);
00686         if (val!=1 || strcmp(buffer,"")!=0) {
00687             status = ERROR_INPUT_PAR; break;
00688         }
00689         val = fscanf(fp,"%lf",&dbuff);
00690         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00691         inparam->AMG_max_row_sum = dbuff;
00692         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00693     }
00694
00695     else if (strcmp(buffer,"AMG_amli_degree")==0) {
00696         val = fscanf(fp,"%s",buffer);
00697         if (val!=1 || strcmp(buffer,"")!=0) {
00698             status = ERROR_INPUT_PAR; break;
00699         }
00700         val = fscanf(fp,"%d",&ibuff);
00701         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00702         inparam->AMG_amli_degree = ibuff;
00703         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00704     }
00705
00706     else if (strcmp(buffer,"AMG_nl_amli_krylov_type")==0) {
00707         val = fscanf(fp,"%s",buffer);
00708         if (val!=1 || strcmp(buffer,"")!=0) {
00709             status = ERROR_INPUT_PAR; break;
00710         }
00711         val = fscanf(fp,"%d",&ibuff);
00712         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00713         inparam->AMG_nl_amli_krylov_type = ibuff;
00714         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00715     }
00716
00717     else if (strcmp(buffer,"ILU_type")==0) {
00718         val = fscanf(fp,"%s",buffer);
00719         if (val!=1 || strcmp(buffer,"")!=0) {
00720             status = ERROR_INPUT_PAR; break;
00721         }
00722         val = fscanf(fp,"%d",&ibuff);
00723         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00724         inparam->ILU_type = ibuff;
00725         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00726     }
00727
00728     else if (strcmp(buffer,"ILU_lfil")==0) {
00729         val = fscanf(fp,"%s",buffer);
00730         if (val!=1 || strcmp(buffer,"")!=0) {
00731             status = ERROR_INPUT_PAR; break;
00732         }
00733         val = fscanf(fp,"%d",&ibuff);
00734         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00735         inparam->ILU_lfil = ibuff;
00736         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00737     }
00738

```



```

00739     else if (strcmp(buffer,"ILU_droptol")==0) {
00740         val = fscanf(fp,"%s",buffer);
00741         if (val!=1 || strcmp(buffer,"")!=0) {
00742             status = ERROR_INPUT_PAR; break;
00743         }
00744         val = fscanf(fp,"%lf",&dbuff);
00745         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00746         inparam->ILU_droptol = dbuff;
00747         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00748     }
00749
00750     else if (strcmp(buffer,"ILU_relax")==0) {
00751         val = fscanf(fp,"%s",buffer);
00752         if (val!=1 || strcmp(buffer,"")!=0) {
00753             status = ERROR_INPUT_PAR; break;
00754         }
00755         val = fscanf(fp,"%lf",&dbuff);
00756         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00757         inparam->ILU_relax = dbuff;
00758         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00759     }
00760
00761     else if (strcmp(buffer,"ILU_permtol")==0) {
00762         val = fscanf(fp,"%s",buffer);
00763         if (val!=1 || strcmp(buffer,"")!=0) {
00764             status = ERROR_INPUT_PAR; break;
00765         }
00766         val = fscanf(fp,"%lf",&dbuff);
00767         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00768         inparam->ILU_permtol = dbuff;
00769         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00770     }
00771
00772     else if (strcmp(buffer,"SWZ_mmsize")==0) {
00773         val = fscanf(fp,"%s",buffer);
00774         if (val!=1 || strcmp(buffer,"")!=0) {
00775             status = ERROR_INPUT_PAR; break;
00776         }
00777         val = fscanf(fp,"%d",&ibuff);
00778         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00779         inparam->SWZ_mmsize = ibuff;
00780         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00781     }
00782
00783     else if (strcmp(buffer,"SWZ_maxlvl")==0) {
00784         val = fscanf(fp,"%s",buffer);
00785         if (val!=1 || strcmp(buffer,"")!=0) {
00786             status = ERROR_INPUT_PAR; break;
00787         }
00788         val = fscanf(fp,"%d",&ibuff);
00789         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00790         inparam->SWZ_maxlvl = ibuff;
00791         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00792     }
00793
00794     else if (strcmp(buffer,"SWZ_type")==0) {
00795         val = fscanf(fp,"%s",buffer);
00796         if (val!=1 || strcmp(buffer,"")!=0) {
00797             status = ERROR_INPUT_PAR; break;
00798         }
00799         val = fscanf(fp,"%d",&ibuff);
00800         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00801         inparam->SWZ_type = ibuff;
00802         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00803     }
00804
00805     else if (strcmp(buffer,"SWZ_blksolver")==0) {
00806         val = fscanf(fp,"%s",buffer);
00807         if (val!=1 || strcmp(buffer,"")!=0) {
00808             status = ERROR_INPUT_PAR; break;
00809         }
00810         val = fscanf(fp,"%d",&ibuff);
00811         if (val!=1) { status = ERROR_INPUT_PAR; break; }
00812         inparam->SWZ_blksolver = ibuff;
00813         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00814     }
00815
00816     else {
00817         printf("### WARNING: Unknown input keyword %s!\\n", buffer);
00818         if (fscanf(fp, "%*[^\\n]") /* skip rest of line and do nothing */ );
00819     }

```

```

00820     }
00821
00822     fclose(fp);
00823
00824     // if meet unexpected input, stop the program
00825     fasp_chkerr(status, __FUNCTION__);
00826
00827     // sanity checks
00828     status = fasp_param_check(inparam);
00829
00830     #if DEBUG_MODE > 1
00831     printf("### DEBUG: Reading input status = %d\n", status);
00832     #endif
00833
00834     fasp_chkerr(status, __FUNCTION__);
00835 }
00836
00837 /*-----*/
00838 /*--      End of File      --*/
00839 /*-----*/

```

9.31 AuxMemory.c File Reference

Memory allocation and deallocation subroutines.

```

#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void * [fasp_mem_calloc](#) (const unsigned int size, const unsigned int type)
Allocate, initiate, and check memory.
- void * [fasp_mem_realloc](#) (void *oldmem, const [LONGLONG](#) tsize)
Reallocate, initiate, and check memory.
- void [fasp_mem_free](#) (void *mem)
Free up previous allocated memory body and set pointer to NULL.
- void [fasp_mem_usage](#) (void)
Show total allocated memory currently.
- [SHORT fasp_mem_iludata_check](#) (const [ILU_data](#) *iludata)
Check whether a [ILU_data](#) has enough work space.

Variables

- const int [Million](#) = 1048576

9.31.1 Detailed Description

Memory allocation and deallocation subroutines.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxMemory.c](#).

9.31.2 Function Documentation

9.31.2.1 fasp_mem_calloc()

```
void * fasp_mem_calloc (
    const unsigned int size,
    const unsigned int type )
```

Allocate, initiate, and check memory.

Parameters

<i>size</i>	Number of memory blocks
<i>type</i>	Size of memory blocks

Returns

Void pointer to the allocated memory

Author

Chensong Zhang

Date

2010/08/12

Modified by Chensong Zhang on 07/30/2013: print warnings if failed
Definition at line 65 of file [AuxMemory.c](#).

9.31.2.2 fasp_mem_free()

```
void fasp_mem_free (
    void * mem )
```

Free up previous allocated memory body and set pointer to NULL.

Parameters

<i>mem</i>	Pointer to the memory body need to be freed
------------	---

Author

Chensong Zhang

Date

2010/12/24

Modified on 2018/01/10 by Chensong: Add output when mem is NULL
Definition at line 155 of file [AuxMemory.c](#).

9.31.2.3 fasp_mem_iludata_check()

```
SHORT fasp_mem_iludata_check (
    const ILU_data * iludata )
```

Check whether a [ILU_data](#) has enough work space.

Parameters

<i>iludata</i>	Pointer to be checked
----------------	-----------------------

Returns

FASP_SUCCESS if success, else ERROR (negative value)

Author

Xiaozhe Hu, Chensong Zhang

Date

11/27/09

Definition at line 205 of file [AuxMemory.c](#).

9.31.2.4 fasp_mem_realloc()

```
void * fasp_mem_realloc (
    void * oldmem,
    const LONGLONG tsize )
```

Reallocate, initiate, and check memory.

Parameters

<i>oldmem</i>	Pointer to the existing mem block
<i>tsize</i>	Size of memory blocks

Returns

Void pointer to the reallocated memory

Author

Chensong Zhang

Date

2010/08/12

Modified by Chensong Zhang on 07/30/2013: print error if failed

Definition at line 114 of file [AuxMemory.c](#).

9.31.2.5 fasp_mem_usage()

```
void fasp_mem_usage (
    void )
```

Show total allocated memory currently.

Author

Chensong Zhang

Date

2010/08/12

Definition at line 185 of file [AuxMemory.c](#).

9.31.3 Variable Documentation

9.31.3.1 Million

```
const int Million = 1048576
```

1M = 1024*1024

Definition at line 44 of file [AuxMemory.c](#).

9.32 AuxMemory.c

[Go to the documentation of this file.](#)

```
00001
00013 /*-----*/
00014 /*-- Declare External Functions --*/
00015 /*-----*/
00016
00017 #include "fasp.h"
00018 #include "fasp_funcs.h"
00019
00020 #if DLMALLOC
00021 #include "dlmalloc.h"
00022 #elif NEDMALLOC
00023 #include "nedmalloc.h"
00024 #ifdef __cplusplus
00025 extern "C" {
00026 #endif
00027     void * nedcalloc(size_t no, size_t size);
00028     void * nedrealloc(void *mem, size_t size);
00029     void  nedfree(void *mem);
00030 #ifdef __cplusplus
00031 }
00032 #endif
00033 #endif
00034
00035 #if DEBUG_MODE > 1
00036 extern unsigned long total_alloc_mem;
00037 extern unsigned long total_alloc_count;
00038 #endif
00039
00040 /*-----*/
00041 /*--      Global Variables      --*/
00042 /*-----*/
00043
00044 const int Million = 1048576;
00045
00046 /*-----*/
00047 /*--      Public Functions      --*/
00048 /*-----*/
00049
00065 void * fasp_mem_calloc (const unsigned int  size,
00066                        const unsigned int  type)
00067 {
00068     const LONGLONG tsize = size*type;
00069     void * mem = NULL;
00070
00071 #if DEBUG_MODE > 1
00072     printf("### DEBUG: Trying to allocate %.3lfMB RAM!\n", (REAL)tsize/Million);
```

```

00073 #endif
00074
00075     if ( tsize > 0 ) {
00076
00077 #if DLMALLOC
00078         mem = dlcalloc(size,type);
00079 #elif NEDMALLOC
00080         mem = nedcalloc(size,type);
00081 #else
00082         mem = calloc(size,type);
00083 #endif
00084
00085 #if DEBUG_MODE > 1
00086         total_alloc_mem += tsize;
00087         total_alloc_count++;
00088 #endif
00089     }
00090
00091     if ( mem == NULL ) {
00092         printf("### WARNING: Trying to allocate %lldB RAM...\n", tsize);
00093         printf("### WARNING: Cannot allocate %.4fMB RAM!\n", (REAL)tsize/Million);
00094     }
00095
00096     return mem;
00097 }
00098
00114 void * fasp_mem_realloc (void          *oldmem,
00115                        const LONGLONG tsize)
00116 {
00117     void * mem = NULL;
00118
00119 #if DEBUG_MODE > 1
00120     printf("### DEBUG: Trying to allocate %.3fMB RAM!\n", (REAL)tsize/Million);
00121 #endif
00122
00123     if ( tsize > 0 ) {
00124
00125 #if DLMALLOC
00126         mem = dlrealloc(oldmem,tsize);
00127 #elif NEDMALLOC
00128         mem = nedrealloc(oldmem,tsize);
00129 #else
00130         mem = realloc(oldmem,tsize);
00131 #endif
00132
00133     }
00134
00135     if ( mem == NULL ) {
00136         printf("### WARNING: Trying to allocate %lldB RAM!\n", tsize);
00137         printf("### WARNING: Cannot allocate %.3fMB RAM!\n", (REAL)tsize/Million);
00138     }
00139
00140     return mem;
00141 }
00142
00155 void fasp_mem_free (void *mem)
00156 {
00157     if ( mem ) {
00158 #if DLMALLOC
00159         dlfree(mem);
00160 #elif NEDMALLOC
00161         nedfree(mem);
00162 #else
00163         free(mem);
00164 #endif
00165
00166 #if DEBUG_MODE > 1
00167         total_alloc_count--;
00168 #endif
00169     }
00170     else {
00171 #if DEBUG_MODE > 1
00172         printf("### WARNING: Trying to free an empty pointer!\n");
00173 #endif
00174     }
00175 }
00176
00185 void fasp_mem_usage ( void )
00186 {
00187 #if DEBUG_MODE > 1
00188     printf("### DEBUG: Number of alloc = %ld, allocated memory = %.3fMB.\n",

```

```

00189         total_alloc_count, (REAL)total_alloc_mem/Million);
00190 #endif
00191 }
00192
00205 SHORT fasp_mem_iludata_check (const ILU_data *iludata)
00206 {
00207     const INT memneed = 2*iludata->row; // estimated memory usage
00208
00209     if ( iludata->nwork >= memneed ) {
00210         return FASP_SUCCESS;
00211     }
00212     else {
00213         printf("### ERROR: ILU needs %d RAM, only %d available!\n",
00214             memneed, iludata->nwork);
00215         return ERROR_ALLOC_MEM;
00216     }
00217 }
00218
00219 /*-----*/
00220 /*--      End of File      --*/
00221 /*-----*/

```

9.33 AuxMessage.c File Reference

Output some useful messages.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void `fasp_itinfo` (const `INT` ptrlvl, const `INT` stop_type, const `INT` iter, const `REAL` relres, const `REAL` absres, const `REAL` factor)
Print out iteration information for iterative solvers.
- void `fasp_amgcomplexity` (const `AMG_data` *mgl, const `SHORT` ptrlvl)
Print level and complexity information of AMG.
- void `fasp_amgcomplexity_bsr` (const `AMG_data_bsr` *mgl, const `SHORT` ptrlvl)
Print complexities of AMG method for BSR matrices.
- void `fasp_cputime` (const char *message, const `REAL` cputime)
Print CPU walltime.
- void `fasp_message` (const `INT` ptrlvl, const char *message)
Print output information if necessary.
- void `fasp_chkerr` (const `SHORT` status, const char *fctname)
Check error status and print out error messages before quit.

9.33.1 Detailed Description

Output some useful messages.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxMessage.c](#).

9.33.2 Function Documentation

9.33.2.1 fasp_amgcomplexity()

```
void void fasp_amgcomplexity (
    const AMG_data * mgl,
    const SHORT prtlvl )
```

Print level and complexity information of AMG.

Parameters

<i>mgl</i>	Multilevel hierachy for AMG
<i>prtlvl</i>	How much information to print

Author

Chensong Zhang

Date

11/16/2009

Definition at line 84 of file [AuxMessage.c](#).

9.33.2.2 fasp_amgcomplexity_bsr()

```
void void fasp_amgcomplexity_bsr (
    const AMG_data_bsr * mgl,
    const SHORT prtlvl )
```

Print complexities of AMG method for BSR matrices.

Parameters

<i>mgl</i>	Multilevel hierachy for AMG
<i>prtlvl</i>	How much information to print

Author

Chensong Zhang

Date

05/10/2013

Definition at line 136 of file [AuxMessage.c](#).

9.33.2.3 fasp_chkerr()

```
void fasp_chkerr (
    const SHORT status,
    const char * fctname )
```


Check error status and print out error messages before quit.

Parameters

<i>status</i>	Error status
<i>fctname</i>	Function name where this routine is called

Author

Chensong Zhang

Date

01/10/2012

Definition at line 213 of file [AuxMessage.c](#).

9.33.2.4 fasp_cputime()

```
void void fasp_cputime (
    const char * message,
    const REAL cputime )
```

Print CPU walltime.

Parameters

<i>message</i>	Some string to print out
<i>cputime</i>	Walltime since start to end

Author

Chensong Zhang

Date

04/10/2012

Definition at line 179 of file [AuxMessage.c](#).

9.33.2.5 fasp_itinfo()

```
void fasp_itinfo (
    const INT ptrlvl,
    const INT stop_type,
    const INT iter,
    const REAL relres,
    const REAL absres,
    const REAL factor )
```

Print out iteration information for iterative solvers.

Parameters

<i>ptrlvl</i>	Level for output
<i>stop_type</i>	Type of stopping criteria
<i>iter</i>	Number of iterations

Parameters

<i>relres</i>	Relative residual of different kinds
<i>absres</i>	Absolute residual of different kinds
<i>factor</i>	Contraction factor

Author

Chensong Zhang

Date

11/16/2009

Modified by Chensong Zhang on 03/28/2013: Output initial guess Modified by Chensong Zhang on 04/05/2013: Fix a typo

Definition at line 41 of file [AuxMessage.c](#).

9.33.2.6 fasp_message()

```
void fasp_message (
    const INT ptrlvl,
    const char * message )
```

Print output information if necessary.

Parameters

<i>ptrlvl</i>	Level for output
<i>message</i>	Error message to print

Author

Chensong Zhang

Date

11/16/2009

Definition at line 196 of file [AuxMessage.c](#).

9.34 AuxMessage.c

[Go to the documentation of this file.](#)

```
00001
00013 #include <math.h>
00014
00015 #include "fasp.h"
00016 #include "fasp_functs.h"
00017
00018 /*-----*/
00019 /*--      Public Functions      --*/
00020 /*-----*/
00021
00041 void fasp_itinfo (const INT   ptrlvl,
00042                  const INT   stop_type,
00043                  const INT   iter,
00044                  const REAL   relres,
```

```

00045         const REAL absres,
00046         const REAL factor)
00047 {
00048     if ( prtlvl >= PRINT_SOME ) {
00049
00050         if ( iter > 0 ) {
00051             printf("%6d | %13.6e | %13.6e | %10.4f\n", iter, relres, absres, factor);
00052         }
00053         else { // iter = 0: initial guess
00054             printf("-----\n");
00055             switch (stop_type) {
00056                 case STOP_REL_RES:
00057                     printf("It Num | ||r||/||b|| | ||r|| | Conv. Factor\n");
00058                     break;
00059                 case STOP_REL_PRECRES:
00060                     printf("It Num | ||r||_B/||b||_B | ||r||_B | Conv. Factor\n");
00061                     break;
00062                 case STOP_MOD_REL_RES:
00063                     printf("It Num | ||r||/||x|| | ||r|| | Conv. Factor\n");
00064                     break;
00065             }
00066             printf("-----\n");
00067             printf("%6d | %13.6e | %13.6e | -.- \n", iter, relres, absres);
00068         } // end if iter
00069     } // end if prtlvl
00070 }
00071 }
00072
00084 void fasp_amgcomplexity (const AMG_data *mgl,
00085                         const SHORT prtlvl)
00086 {
00087     const SHORT max_levels = mgl->num_levels;
00088     SHORT level;
00089     REAL gridcom = 0.0, opcom = 0.0;
00090
00091     if ( prtlvl >= PRINT_SOME ) {
00092
00093         printf("-----\n");
00094         printf(" Level Num of rows Num of nonzeros Avg. NNZ / row \n");
00095         printf("-----\n");
00096
00097         for ( level = 0; level < max_levels; ++level) {
00098             const REAL AvgNNZ = (REAL) mgl[level].A.nnz/mgl[level].A.row;
00099             printf("%5d %13d %17d %14.2f\n",
00100                 level, mgl[level].A.row, mgl[level].A.nnz, AvgNNZ);
00101             gridcom += mgl[level].A.row;
00102             opcom += mgl[level].A.nnz;
00103
00104             #if 0 // Save coarser linear systems for debugging purposes --Chensong
00105                 char matA[max_levels], rhsb[max_levels];
00106                 if (level > 0) {
00107                     sprintf(matA, "A%d.coo", level);
00108                     sprintf(rhsb, "b%d.coo", level);
00109                     fasp_dcsrvec_write2(matA, rhsb, &(mgl[level].A), &(mgl[level].b));
00110                 }
00111             #endif
00112         }
00113         printf("-----\n");
00114
00115         gridcom /= mgl[0].A.row;
00116         opcom /= mgl[0].A.nnz;
00117         printf(" Grid complexity = %.3f |", gridcom);
00118         printf(" Operator complexity = %.3f\n", opcom);
00119
00120         printf("-----\n");
00121     }
00122 }
00123
00136 void fasp_amgcomplexity_bsr (const AMG_data_bsr *mgl,
00137                             const SHORT prtlvl)
00138 {
00139     const SHORT max_levels = mgl->num_levels;
00140     SHORT level;
00141     REAL gridcom = 0.0, opcom = 0.0;
00142
00143     if ( prtlvl >= PRINT_SOME ) {
00144
00145         printf("-----\n");
00146         printf(" Level Num of rows Num of nonzeros Avg. NNZ / row \n");
00147         printf("-----\n");
00148

```

```

00149     for ( level = 0; level < max_levels; ++level ) {
00150         const REAL AvgNNZ = (REAL) mgl[level].A.NNZ/mgl[level].A.ROW;
00151         printf("%5d %13d %17d %14.2f\n",
00152             level,mgl[level].A.ROW, mgl[level].A.NNZ, AvgNNZ);
00153         gridcom += mgl[level].A.ROW;
00154         opcom    += mgl[level].A.NNZ;
00155     }
00156     printf("-----\n");
00157
00158     gridcom /= mgl[0].A.ROW;
00159     opcom    /= mgl[0].A.NNZ;
00160     printf(" Grid complexity = %.3f |", gridcom);
00161     printf(" Operator complexity = %.3f\n", opcom);
00162
00163     printf("-----\n");
00164 }
00165 }
00166 }
00167
00179 void fasp_cputime (const char *message,
00180                  const REAL cputime)
00181 {
00182     printf("%s costs %.4f seconds\n", message, cputime);
00183 }
00184
00196 void fasp_message (const INT ptrlvl,
00197                   const char *message)
00198 {
00199     if ( ptrlvl > PRINT_NONE ) printf("%s", message);
00200 }
00201
00213 void fasp_chkerr (const SHORT status,
00214                  const char *fctname)
00215 {
00216     if ( status >= 0 ) return; // No error found!!!
00217
00218     switch ( status ) {
00219     case ERROR_READ_FILE:
00220         printf("### ERROR: Cannot read file! [%s]\n", fctname);
00221         break;
00222     case ERROR_OPEN_FILE:
00223         printf("### ERROR: Cannot open file! [%s]\n", fctname);
00224         break;
00225     case ERROR_WRONG_FILE:
00226         printf("### ERROR: Unknown file format! [%s]\n", fctname);
00227         break;
00228     case ERROR_INPUT_PAR:
00229         printf("### ERROR: Unknown input argument! [%s]\n", fctname);
00230         break;
00231     case ERROR_REGRESS:
00232         printf("### ERROR: Regression test failed! [%s]\n", fctname);
00233         break;
00234     case ERROR_ALLOC_MEM:
00235         printf("### ERROR: Cannot allocate memory! [%s]\n", fctname);
00236         break;
00237     case ERROR_NUM_BLOCKS:
00238         printf("### ERROR: Unexpected number of blocks! [%s]\n", fctname);
00239         break;
00240     case ERROR_DATA_STRUCTURE:
00241         printf("### ERROR: Wrong data structure! [%s]\n", fctname);
00242         break;
00243     case ERROR_DATA_ZERODIAG:
00244         printf("### ERROR: Matrix has zero diagonal entries! [%s]\n", fctname);
00245         break;
00246     case ERROR_DUMMY_VAR:
00247         printf("### ERROR: Unknown input argument! [%s]\n", fctname);
00248         break;
00249     case ERROR_AMG_INTERP_TYPE:
00250         printf("### ERROR: Unknown AMG interpolation type! [%s]\n", fctname);
00251         break;
00252     case ERROR_AMG_COARSE_TYPE:
00253         printf("### ERROR: Unknown AMG coarsening type! [%s]\n", fctname);
00254         break;
00255     case ERROR_AMG_SMOOTH_TYPE:
00256         printf("### ERROR: Unknown AMG smoother type! [%s]\n", fctname);
00257         break;
00258     case ERROR_SOLVER_TYPE:
00259         printf("### ERROR: Unknown solver type! [%s]\n", fctname);
00260         break;
00261     case ERROR_SOLVER_PRECTYPE:
00262         printf("### ERROR: Unknown preconditioner type! [%s]\n", fctname);

```

```

00263         break;
00264     case ERROR_SOLVER_STAG:
00265         printf("### ERROR: Solver stagnation! [%s]\n", fctname);
00266         break;
00267     case ERROR_SOLVER_SOLSTAG:
00268         printf("### ERROR: Solution close to zero! [%s]\n", fctname);
00269         break;
00270     case ERROR_SOLVER_TOLSMALL:
00271         printf("### ERROR: Convergence tolerance too small! [%s]\n", fctname);
00272         break;
00273     case ERROR_SOLVER_ILUSETUP:
00274         printf("### ERROR: ILU setup failed! [%s]\n", fctname);
00275         break;
00276     case ERROR_SOLVER_MAXIT:
00277         printf("### ERROR: Max iteration number reached! [%s]\n", fctname);
00278         break;
00279     case ERROR_SOLVER_EXIT:
00280         printf("### ERROR: Iterative solver failed! [%s]\n", fctname);
00281         break;
00282     case ERROR_SOLVER_MISC:
00283         printf("### ERROR: Unknown solver runtime error! [%s]\n", fctname);
00284         break;
00285     case ERROR_MISC:
00286         printf("### ERROR: Miscellaneous error! [%s]\n", fctname);
00287         break;
00288     case ERROR_QUAD_TYPE:
00289         printf("### ERROR: Unknown quadrature rules! [%s]\n", fctname);
00290         break;
00291     case ERROR_QUAD_DIM:
00292         printf("### ERROR: Num of quad points not supported! [%s]\n", fctname);
00293         break;
00294     case ERROR_UNKNOWN:
00295         printf("### ERROR: Unknown error! [%s]\n", fctname);
00296         break;
00297     default:
00298         break;
00299 }
00300
00301 exit(status);
00302 }
00303
00304 /*-----*/
00305 /*--      End of File      --*/
00306 /*-----*/

```

9.35 AuxParam.c File Reference

Initialize, set, or print input data and parameters.

```

#include <stdio.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_param_set` (const int argc, const char *argv[], `input_param` *iniparam)
Read input from command-line arguments.
- void `fasp_param_init` (const `input_param` *iniparam, `ITS_param` *itsparam, `AMG_param` *amgparam, `ILU_param` *iluparam, `SWZ_param` *swzparam)
Initialize parameters, global variables, etc.
- void `fasp_param_input_init` (`input_param` *iniparam)
Initialize input parameters.
- void `fasp_param_amg_init` (`AMG_param` *amgparam)
Initialize AMG parameters.
- void `fasp_param_solver_init` (`ITS_param` *itsparam)
Initialize ITS_param.

- void [fasp_param_ilu_init](#) ([ILU_param](#) *iluparam)
Initialize ILU parameters.
- void [fasp_param_swz_init](#) ([SWZ_param](#) *swzparam)
Initialize Schwarz parameters.
- void [fasp_param_amg_set](#) ([AMG_param](#) *param, const [input_param](#) *iniparam)
Set AMG_param from INPUT.
- void [fasp_param_ilu_set](#) ([ILU_param](#) *iluparam, const [input_param](#) *iniparam)
Set ILU_param with INPUT.
- void [fasp_param_swz_set](#) ([SWZ_param](#) *swzparam, const [input_param](#) *iniparam)
Set SWZ_param with INPUT.
- void [fasp_param_solver_set](#) ([ITS_param](#) *itsparam, const [input_param](#) *iniparam)
Set ITS_param with INPUT.
- void [fasp_param_amg_to_prec](#) ([precond_data](#) *pcdata, const [AMG_param](#) *amgparam)
Set precondition_data with AMG_param.
- void [fasp_param_prec_to_amg](#) ([AMG_param](#) *amgparam, const [precond_data](#) *pcdata)
Set AMG_param with precondition_data.
- void [fasp_param_amg_to_precbsr](#) ([precond_data_bsr](#) *pcdata, const [AMG_param](#) *amgparam)
Set precondition_data_bsr with AMG_param.
- void [fasp_param_precbsr_to_amg](#) ([AMG_param](#) *amgparam, const [precond_data_bsr](#) *pcdata)
Set AMG_param with precondition_data.
- void [fasp_param_amg_print](#) (const [AMG_param](#) *param)
Print out AMG parameters.
- void [fasp_param_ilu_print](#) (const [ILU_param](#) *param)
Print out ILU parameters.
- void [fasp_param_swz_print](#) (const [SWZ_param](#) *param)
Print out Schwarz parameters.
- void [fasp_param_solver_print](#) (const [ITS_param](#) *param)
Print out itsolver parameters.

9.35.1 Detailed Description

Initialize, set, or print input data and parameters.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxInput.c](#) and [AuxMessage.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxParam.c](#).

9.35.2 Function Documentation

9.35.2.1 [fasp_param_amg_init\(\)](#)

```
void fasp_param_amg_init (
    AMG\_param * amgparam )
```

Initialize AMG parameters.

Parameters

<i>amgparam</i>	Parameters for AMG
-----------------	--------------------

Author

Chensong Zhang

Date

2010/04/03

Definition at line 407 of file [AuxParam.c](#).

9.35.2.2 fasp_param_amg_print()

```
void fasp_param_amg_print (
    const AMG\_param * param )
```

Print out AMG parameters.

Parameters

<i>param</i>	Parameters for AMG
--------------	--------------------

Author

Chensong Zhang

Date

2010/03/22

Definition at line 820 of file [AuxParam.c](#).

9.35.2.3 fasp_param_amg_set()

```
void fasp_param_amg_set (
    AMG\_param * param,
    const input\_param * iniparam )
```

Set [AMG_param](#) from INPUT.

Parameters

<i>param</i>	Parameters for AMG
<i>iniparam</i>	Input parameters

Author

Chensong Zhang

Date

2010/03/23

Definition at line 537 of file [AuxParam.c](#).**9.35.2.4 fasp_param_amg_to_prec()**

```
void fasp_param_amg_to_prec (
    precondition_data * pcd_data,
    const AMG_param * amgparam )
```

Set [precond_data](#) with [AMG_param](#).

Parameters

<i>pcdata</i>	Preconditioning data structure
<i>amgparam</i>	Parameters for AMG

Author

Chensong Zhang

Date

2011/01/10

Definition at line 687 of file [AuxParam.c](#).**9.35.2.5 fasp_param_amg_to_precbsr()**

```
void fasp_param_amg_to_precbsr (
    precondition_data_bsr * pcd_data,
    const AMG_param * amgparam )
```

Set [precond_data_bsr](#) with [AMG_param](#).

Parameters

<i>pcdata</i>	Preconditioning data structure
<i>amgparam</i>	Parameters for AMG

Author

Xiaozhe Hu

Date

02/06/2012

Definition at line 755 of file [AuxParam.c](#).**9.35.2.6 fasp_param_ilu_init()**

```
void fasp_param_ilu_init (
    ILU_param * iluparam )
```

Initialize ILU parameters.

Parameters

<i>iluparam</i>	Parameters for ILU
-----------------	--------------------

Author

Chensong Zhang

Date

2010/04/06

Definition at line 495 of file [AuxParam.c](#).

9.35.2.7 fasp_param_ilu_print()

```
void fasp_param_ilu_print (
    const ILU_param * param )
```

Print out ILU parameters.

Parameters

<i>param</i>	Parameters for ILU
--------------	--------------------

Author

Chensong Zhang

Date

2011/12/20

Definition at line 943 of file [AuxParam.c](#).

9.35.2.8 fasp_param_ilu_set()

```
void fasp_param_ilu_set (
    ILU_param * iluparam,
    const input_param * iniparam )
```

Set [ILU_param](#) with INPUT.

Parameters

<i>iluparam</i>	Parameters for ILU
<i>iniparam</i>	Input parameters

Author

Chensong Zhang

Date

2010/04/03

Definition at line 612 of file [AuxParam.c](#).**9.35.2.9 fasp_param_init()**

```
void fasp_param_init (
    const input\_param * iniparam,
    ITS\_param * itsparam,
    AMG\_param * amgparam,
    ILU\_param * iluparam,
    SWZ\_param * swzparam )
```

Initialize parameters, global variables, etc.

Parameters

<i>iniparam</i>	Input parameters
<i>itsparam</i>	Iterative solver parameters
<i>amgparam</i>	AMG parameters
<i>iluparam</i>	ILU parameters
<i>swzparam</i>	Schwarz parameters

Author

Chensong Zhang

Date

2010/08/12

Modified by Chensong Zhang (12/29/2013): rewritten

Definition at line 283 of file [AuxParam.c](#).**9.35.2.10 fasp_param_input_init()**

```
void fasp_param_input_init (
    input\_param * iniparam )
```

Initialize input parameters.

Parameters

<i>iniparam</i>	Input parameters
-----------------	------------------

Author

Chensong Zhang

Date

2010/03/20

Definition at line 325 of file [AuxParam.c](#).**9.35.2.11 fasp_param_prec_to_amg()**

```
void fasp_param_prec_to_amg (
    AMG_param * amgparam,
    const precondition_data * pcddata )
```

Set [AMG_param](#) with [precond_data](#).

Parameters

<i>amgparam</i>	Parameters for AMG
<i>pcdata</i>	Preconditioning data structure

Author

Chensong Zhang

Date

2011/01/10

Definition at line 722 of file [AuxParam.c](#).**9.35.2.12 fasp_param_precbsr_to_amg()**

```
void fasp_param_precbsr_to_amg (
    AMG_param * amgparam,
    const precondition_data_bsr * pcddata )
```

Set [AMG_param](#) with [precond_data](#).

Parameters

<i>amgparam</i>	Parameters for AMG
<i>pcdata</i>	Preconditioning data structure

Author

Xiaozhe Hu

Date

02/06/2012

Definition at line 790 of file [AuxParam.c](#).**9.35.2.13 fasp_param_set()**

```
void fasp_param_set (
    const int argc,
```

```
const char * argv[],  
input_param * iniparam )
```

Read input from command-line arguments.

Parameters

<i>argc</i>	Number of arg input
<i>argv</i>	Input arguments
<i>iniparam</i>	Parameters to be set

Author

Chensong Zhang

Date

12/29/2013

Definition at line 41 of file [AuxParam.c](#).

9.35.2.14 fasp_param_solver_init()

```
void fasp_param_solver_init (  
    ITS_param * itsparam )
```

Initialize [ITS_param](#).

Parameters

<i>itsparam</i>	Parameters for iterative solvers
-----------------	----------------------------------

Author

Chensong Zhang

Date

2010/03/23

Definition at line 473 of file [AuxParam.c](#).

9.35.2.15 fasp_param_solver_print()

```
void fasp_param_solver_print (  
    const ITS_param * param )
```

Print out itsolver parameters.

Parameters

<i>param</i>	Paramters for iterative solvers
--------------	---------------------------------

Author

Chensong Zhang

Date

2011/12/20

Definition at line 1002 of file [AuxParam.c](#).**9.35.2.16 fasp_param_solver_set()**

```
void fasp_param_solver_set (
    ITS_param * itsparam,
    const input_param * iniparam )
```

Set [ITS_param](#) with INPUT.

Parameters

<i>itsparam</i>	Parameters for iterative solvers
<i>iniparam</i>	Input parameters

Author

Chensong Zhang

Date

2010/03/23

Definition at line 656 of file [AuxParam.c](#).**9.35.2.17 fasp_param_swz_init()**

```
void fasp_param_swz_init (
    SWZ_param * swzparam )
```

Initialize Schwarz parameters.

Parameters

<i>swzparam</i>	Parameters for Schwarz method
-----------------	-------------------------------

Author

Xiaozhe Hu

Date

05/22/2012

Modified by Chensong Zhang on 10/10/2014: Add block solver type

Definition at line 517 of file [AuxParam.c](#).

9.35.2.18 fasp_param_swz_print()

```
void fasp_param_swz_print (
    const SWZ_param * param )
```

Print out Schwarz parameters.

Parameters

<i>param</i>	Parameters for Schwarz
--------------	------------------------

Author

Xiaozhe Hu

Date

05/22/2012

Definition at line 973 of file [AuxParam.c](#).

9.35.2.19 fasp_param_swz_set()

```
void fasp_param_swz_set (
    SWZ_param * swzparam,
    const input_param * iniparam )
```

Set [SWZ_param](#) with INPUT.

Parameters

<i>swzparam</i>	Parameters for Schwarz method
<i>iniparam</i>	Input parameters

Author

Xiaozhe Hu

Date

05/22/2012

Definition at line 634 of file [AuxParam.c](#).

9.36 AuxParam.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <stdio.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_funcs.h"
00018
00019 #if DEBUG_MODE > 1
00020 unsigned long total_alloc_mem;
00021 unsigned long total_alloc_count;
00022 #endif
00023
00024 /*-----*/
```

```

00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00041 void fasp_param_set (const int    argc,
00042                    const char  *argv[],
00043                    input_param *iniparam)
00044 {
00045     int    arg_index    = 1;
00046     int    print_usage  = FALSE;
00047     SHORT  status       = FASP_SUCCESS;
00048
00049     // Option 1. set default input parameters
00050     fasp_param_input_init(iniparam);
00051
00052     while ( arg_index < argc ) {
00053
00054         if ( strcmp(argv[arg_index], "-help") == 0 ) {
00055             print_usage = TRUE; break;
00056         }
00057
00058         // Option 2. Get parameters from an ini file
00059         else if ( strcmp(argv[arg_index], "-ini") == 0 ) {
00060             arg_index++;
00061             if ( arg_index >= argc ) {
00062                 printf("### ERROR: Missing ini filename! [%s]\n", __FUNCTION__);
00063                 print_usage = TRUE; break;
00064             }
00065             strcpy(iniparam->inifile, argv[arg_index]);
00066             fasp_param_input(iniparam->inifile, iniparam);
00067             if ( ++arg_index >= argc ) break;
00068         }
00069
00070         // Option 3. Get parameters from command line input
00071         else if ( strcmp(argv[arg_index], "-print") == 0 ) {
00072             arg_index++;
00073             if ( arg_index >= argc ) {
00074                 printf("### ERROR: Expecting print level (from 0 to 10).\n");
00075                 print_usage = TRUE; break;
00076             }
00077             iniparam->print_level = atoi(argv[arg_index]);
00078             if ( ++arg_index >= argc ) break;
00079         }
00080
00081         else if ( strcmp(argv[arg_index], "-output") == 0 ) {
00082             arg_index++;
00083             if ( arg_index >= argc ) {
00084                 printf("### ERROR: Expecting output type (0 or 1).\n");
00085                 print_usage = TRUE; break;
00086             }
00087             iniparam->output_type = atoi(argv[arg_index]);
00088             if ( ++arg_index >= argc ) break;
00089         }
00090
00091         else if ( strcmp(argv[arg_index], "-solver") == 0 ) {
00092             arg_index++;
00093             if ( arg_index >= argc ) {
00094                 printf("### ERROR: Expecting solver type.\n");
00095                 print_usage = TRUE; break;
00096             }
00097             iniparam->solver_type = atoi(argv[arg_index]);
00098             if ( ++arg_index >= argc ) break;
00099         }
00100
00101         else if ( strcmp(argv[arg_index], "-precond") == 0 ) {
00102             arg_index++;
00103             if ( arg_index >= argc ) {
00104                 printf("### ERROR: Expecting preconditioner type.\n");
00105                 print_usage = TRUE; break;
00106             }
00107             iniparam->precond_type = atoi(argv[arg_index]);
00108             if ( ++arg_index >= argc ) break;
00109         }
00110
00111         else if ( strcmp(argv[arg_index], "-maxit") == 0 ) {
00112             arg_index++;
00113             if ( arg_index >= argc ) {
00114                 printf("### ERROR: Expecting max number of iterations.\n");
00115                 print_usage = TRUE; break;
00116             }
00117             iniparam->itsolver_maxit = atoi(argv[arg_index]);
00118             if ( ++arg_index >= argc ) break;

```



```

00119     }
00120
00121     else if ( strcmp(argv[arg_index], "-tol") == 0 ) {
00122         arg_index++;
00123         if ( arg_index >= argc ) {
00124             printf("### ERROR: Expecting tolerance for itsolver.\n");
00125             print_usage = TRUE; break;
00126         }
00127         iniparam->itsolver_tol = atof(argv[arg_index]);
00128         if ( ++arg_index >= argc ) break;
00129     }
00130
00131     else if ( strcmp(argv[arg_index], "-amgmaxit") == 0 ) {
00132         arg_index++;
00133         if ( arg_index >= argc ) {
00134             printf("### ERROR: Expecting max num of iterations for AMG.\n");
00135             print_usage = TRUE; break;
00136         }
00137         iniparam->AMG_maxit = atoi(argv[arg_index]);
00138         if ( ++arg_index >= argc ) break;
00139     }
00140
00141     else if ( strcmp(argv[arg_index], "-amgtol") == 0 ) {
00142         arg_index++;
00143         if ( arg_index >= argc ) {
00144             printf("### ERROR: Expecting tolerance for AMG.\n");
00145             print_usage = TRUE; break;
00146         }
00147         iniparam->AMG_tol = atof(argv[arg_index]);
00148         if ( ++arg_index >= argc ) break;
00149     }
00150
00151     else if ( strcmp(argv[arg_index], "-amgtype") == 0 ) {
00152         arg_index++;
00153         if ( arg_index >= argc ) {
00154             printf("### ERROR: Expecting AMG type (1, 2, 3).\n");
00155             print_usage = TRUE; break;
00156         }
00157         iniparam->AMG_type = atoi(argv[arg_index]);
00158         if ( ++arg_index >= argc ) break;
00159     }
00160
00161     else if ( strcmp(argv[arg_index], "-amgcycle") == 0 ) {
00162         arg_index++;
00163         if ( arg_index >= argc ) {
00164             printf("### ERROR: Expecting AMG cycle type (1, 2, 3, 12, 21).\n");
00165             print_usage = TRUE; break;
00166         }
00167         iniparam->AMG_cycle_type = atoi(argv[arg_index]);
00168         if ( ++arg_index >= argc ) break;
00169     }
00170
00171     else if ( strcmp(argv[arg_index], "-amgcoarsening") == 0 ) {
00172         arg_index++;
00173         if ( arg_index >= argc ) {
00174             printf("### ERROR: Expecting AMG coarsening type.\n");
00175             print_usage = TRUE; break;
00176         }
00177         iniparam->AMG_coarsening_type = atoi(argv[arg_index]);
00178         if ( ++arg_index >= argc ) break;
00179     }
00180
00181     else if ( strcmp(argv[arg_index], "-amginterplation") == 0 ) {
00182         arg_index++;
00183         if ( arg_index >= argc ) {
00184             printf("### ERROR: Expecting AMG interpolation type.\n");
00185             print_usage = TRUE; break;
00186         }
00187         iniparam->AMG_interpolation_type = atoi(argv[arg_index]);
00188         if ( ++arg_index >= argc ) break;
00189     }
00190
00191     else if ( strcmp(argv[arg_index], "-amgsmoother") == 0 ) {
00192         arg_index++;
00193         if ( arg_index >= argc ) {
00194             printf("### ERROR: Expecting AMG smoother type.\n");
00195             print_usage = TRUE; break;
00196         }
00197         iniparam->AMG_smoother = atoi(argv[arg_index]);
00198         if ( ++arg_index >= argc ) break;
00199     }

```

```

00200
00201     else if ( strcmp(argv[arg_index], "-amgsthreshold") == 0 ) {
00202         arg_index++;
00203         if ( arg_index >= argc ) {
00204             printf("### ERROR: Expecting AMG strong threshold.\n");
00205             print_usage = TRUE; break;
00206         }
00207         iniparam->AMG_strong_threshold = atof(argv[arg_index]);
00208         if ( ++arg_index >= argc ) break;
00209     }
00210
00211     else if ( strcmp(argv[arg_index], "-amgscouple") == 0 ) {
00212         arg_index++;
00213         if ( arg_index >= argc ) {
00214             printf("### ERROR: Expecting AMG strong coupled threshold.\n");
00215             print_usage = TRUE; break;
00216         }
00217         iniparam->AMG_strong_coupled = atof(argv[arg_index]);
00218         if ( ++arg_index >= argc ) break;
00219     }
00220
00221     else {
00222         print_usage = TRUE;
00223         break;
00224     }
00225 }
00226
00227 if ( print_usage ) {
00228     printf("FASP command line options:\n");
00229
00230     printf("=====\n");
00231     printf(" -ini           [CharValue] : Ini file name\n");
00232     printf(" -print        [IntValue]  : Print level\n");
00233     printf(" -output       [IntValue]  : Output to screen or a log file\n");
00234     printf(" -solver       [IntValue]  : Solver type\n");
00235     printf(" -precond      [IntValue]  : Preconditioner type\n");
00236     printf(" -maxit        [IntValue]  : Max number of iterations\n");
00237     printf(" -tol          [RealValue] : Tolerance for iterative solvers\n");
00238     printf(" -amgmaxit     [IntValue]  : Max number of AMG iterations\n");
00239     printf(" -amgtol       [RealValue] : Tolerance for AMG methods\n");
00240     printf(" -amgtype      [IntValue]  : AMG type\n");
00241     printf(" -amgcycle     [IntValue]  : AMG cycle type\n");
00242     printf(" -amgcoarsening [IntValue] : AMG coarsening type\n");
00243     printf(" -amginterpolation [IntValue] : AMG interpolation type\n");
00244     printf(" -amgsmoother  [IntValue]  : AMG smoother type\n");
00245     printf(" -amgsthreshold [RealValue] : AMG strong threshold\n");
00246     printf(" -amgscoupled  [RealValue] : AMG strong coupled threshold\n");
00247     printf(" -help         : Brief help messages\n");
00248
00249     exit(ERROR_INPUT_PAR);
00250 }
00251
00252 // sanity checks
00253 status = fasp_param_check(iniparam);
00254
00255 // if meet unexpected input, stop the program
00256 fasp_chkerr(status, __FUNCTION__);
00257
00258 }
00259
00260 void fasp_param_init (const input_param *iniparam,
00261                     ITS_param *itsparam,
00262                     AMG_param *amgparam,
00263                     ILU_param *iluparam,
00264                     SWZ_param *swzparam)
00265 {
00266     #if DEBUG_MODE > 1
00267         total_alloc_mem = 0; // initialize total memory amount
00268         total_alloc_count = 0; // initialize alloc count
00269     #endif
00270
00271     if (itsparam) fasp_param_solver_init(itsparam);
00272     if (amgparam) fasp_param_amg_init(amgparam);
00273     if (iluparam) fasp_param_ilu_init(iluparam);
00274     if (swzparam) fasp_param_swz_init(swzparam);
00275
00276     if (iniparam) {
00277         if (itsparam) fasp_param_solver_set(itsparam, iniparam);
00278     }
00279 }

```

```

00301         if (amgparam) fasp_param_amg_set(amgparam,iniparam);
00302         if (iluparam) fasp_param_ilu_set(iluparam,iniparam);
00303         if (swzparam) fasp_param_swz_set(swzparam,iniparam);
00304     }
00305     else {
00306         printf("### WARNING: No input given! Use default values instead.\n");
00307     }
00308
00309     // if using AMG as a solver, set min num of iterations = 50
00310     if ( (itsparam == NULL) && (amgparam != NULL) ) {
00311         amgparam->maxit = MAX(amgparam->maxit, 50);
00312     }
00313 }
00314
00325 void fasp_param_input_init (input_param *iniparam)
00326 {
00327     strcpy(iniparam->workdir, "../data/");
00328
00329     // Input/output
00330     iniparam->print_level      = PRINT_SOME;
00331     iniparam->output_type      = 0;
00332
00333     // Problem information
00334     iniparam->problem_num      = 10;
00335     iniparam->solver_type       = SOLVER_CG;
00336     iniparam->decoup_type       = 1;
00337     iniparam->precond_type      = PREC_AMG;
00338     iniparam->stop_type         = STOP_REL_RES;
00339
00340     // Solver parameters
00341     iniparam->itsolver_tol       = 1e-6;
00342     iniparam->itsolver_maxit     = 500;
00343     iniparam->restart            = 25;
00344
00345     // ILU method parameters
00346     iniparam->ILU_type           = ILUk;
00347     iniparam->ILU_lfil           = 0;
00348     iniparam->ILU_droptol        = 0.001;
00349     iniparam->ILU_relax          = 0;
00350     iniparam->ILU_permtol        = 0.0;
00351
00352     // Schwarz method parameters
00353     iniparam->SWZ_mmsize         = 200;
00354     iniparam->SWZ_maxlvl         = 2;
00355     iniparam->SWZ_type           = 1;
00356     iniparam->SWZ_blksolver      = SOLVER_DEFAULT;
00357
00358     // AMG method parameters
00359     iniparam->AMG_type            = CLASSIC_AMG;
00360     iniparam->AMG_levels          = 20;
00361     iniparam->AMG_cycle_type      = V_CYCLE;
00362     iniparam->AMG_smoother        = SMOOTHER_GS;
00363     iniparam->AMG_smooth_order    = CF_ORDER;
00364     iniparam->AMG_presmooth_iter  = 1;
00365     iniparam->AMG_postsmooth_iter = 1;
00366     iniparam->AMG_relaxation       = 1.0;
00367     iniparam->AMG_coarse_dof      = 500;
00368     iniparam->AMG_coarse_solver   = 0;
00369     iniparam->AMG_tol              = 1e-6;
00370     iniparam->AMG_maxit           = 1;
00371     iniparam->AMG_ILU_levels       = 0;
00372     iniparam->AMG_SWZ_levels       = 0;
00373     iniparam->AMG_coarse_scaling   = OFF; // Require investigation --Chensong
00374     iniparam->AMG_amli_degree      = 1;
00375     iniparam->AMG_nl_amli_krylov_type = 2;
00376
00377     // Classical AMG specific
00378     iniparam->AMG_coarsening_type  = 1;
00379     iniparam->AMG_interpolation_type = 1;
00380     iniparam->AMG_max_row_sum       = 0.9;
00381     iniparam->AMG_strong_threshold  = 0.3;
00382     iniparam->AMG_truncation_threshold = 0.2;
00383     iniparam->AMG_aggressive_level  = 0;
00384     iniparam->AMG_aggressive_path   = 1;
00385
00386     // Aggregation AMG specific
00387     iniparam->AMG_aggregation_type  = PAIRWISE;
00388     iniparam->AMG_quality_bound      = 8.0;
00389     iniparam->AMG_pair_number        = 2;
00390     iniparam->AMG_strong_coupled     = 0.25;
00391     iniparam->AMG_max_aggregation   = 9;

```

```

00392     iniparam->AMG_tentative_smooth    = 0.67;
00393     iniparam->AMG_smooth_filter        = ON;
00394     iniparam->AMG_smooth_restriction   = ON;
00395 }
00396
00407 void fasp_param_amg_init (AMG_param *amgparam)
00408 {
00409     // General AMG parameters
00410     amgparam->AMG_type                  = CLASSIC_AMG;
00411     amgparam->print_level                = PRINT_NONE;
00412     amgparam->maxit                     = 1;
00413     amgparam->tol                       = 1e-6;
00414     amgparam->max_levels                 = 20;
00415     amgparam->coarse_dof                 = 500;
00416     amgparam->cycle_type                 = V_CYCLE;
00417     amgparam->smoother                   = SMOOTHER_GS;
00418     amgparam->smooth_order               = CF_ORDER;
00419     amgparam->presmooth_iter             = 1;
00420     amgparam->postsmooth_iter            = 1;
00421     amgparam->coarse_solver              = SOLVER_DEFAULT;
00422     amgparam->relaxation                 = 1.0;
00423     amgparam->polynomial_degree           = 3;
00424     amgparam->coarse_scaling             = OFF;
00425     amgparam->amli_degree                = 2;
00426     amgparam->amli_coef                  = NULL;
00427     amgparam->nl_amli_krylov_type        = SOLVER_GCG;
00428
00429     // Classical AMG specific
00430     amgparam->coarsening_type            = COARSE_RS;
00431     amgparam->interpolation_type         = INTERP_DIR;
00432     amgparam->max_row_sum                 = 0.9;
00433     amgparam->strong_threshold            = 0.3;
00434     amgparam->truncation_threshold        = 0.2;
00435     amgparam->aggressive_level            = 0;
00436     amgparam->aggressive_path            = 1;
00437
00438     // Aggregation AMG specific
00439     amgparam->aggregation_type            = PAIRWISE;
00440     amgparam->quality_bound               = 10.0;
00441     amgparam->pair_number                 = 2;
00442     amgparam->strong_coupled              = 0.08;
00443     amgparam->max_aggregation             = 20;
00444     amgparam->tentative_smooth            = 0.67;
00445     amgparam->smooth_filter               = ON;
00446     amgparam->smooth_restriction          = ON;
00447
00448     // ILU smoother parameters
00449     amgparam->ILU_type                   = ILUK;
00450     amgparam->ILU_levels                  = 0;
00451     amgparam->ILU_lfil                    = 0;
00452     amgparam->ILU_droptol                 = 0.001;
00453     amgparam->ILU_relax                   = 0;
00454
00455     // Schwarz smoother parameters
00456     amgparam->SWZ_levels                  = 0; // levels will use Schwarz smoother
00457     amgparam->SWZ_mmsize                  = 200;
00458     amgparam->SWZ_maxlvl                  = 3; // vertices with smaller distance
00459     amgparam->SWZ_type                    = 1;
00460     amgparam->SWZ_bksolver                 = SOLVER_DEFAULT;
00461 }
00462
00473 void fasp_param_solver_init (ITS_param *itsparam)
00474 {
00475     itsparam->print_level                = PRINT_NONE;
00476     itsparam->itsolver_type               = SOLVER_CG;
00477     itsparam->decoup_type                 = 1;
00478     itsparam->precond_type                = PREC_AMG;
00479     itsparam->stop_type                   = STOP_REL_RES;
00480     itsparam->maxit                       = 500;
00481     itsparam->restart                     = 25;
00482     itsparam->tol                         = 1e-6;
00483 }
00484
00495 void fasp_param_ilu_init (ILU_param *iluparam)
00496 {
00497     iluparam->print_level                = PRINT_NONE;
00498     iluparam->ILU_type                   = ILUK;
00499     iluparam->ILU_lfil                    = 2;
00500     iluparam->ILU_droptol                 = 0.001;
00501     iluparam->ILU_relax                   = 0;
00502     iluparam->ILU_permtol                 = 0.01;

```

```

00503 }
00504
00517 void fasp_param_swz_init (SWZ_param *swzparam)
00518 {
00519     swzparam->print_level = PRINT_NONE;
00520     swzparam->SWZ_type     = 3;
00521     swzparam->SWZ_maxlvl   = 2;
00522     swzparam->SWZ_mmsize   = 200;
00523     swzparam->SWZ_blksolver = 0;
00524 }
00525
00537 void fasp_param_amg_set (AMG_param      *param,
00538                        const input_param *iniparam)
00539 {
00540     param->AMG_type = iniparam->AMG_type;
00541     param->print_level = iniparam->print_level;
00542
00543     if (iniparam->solver_type == SOLVER_AMG) {
00544         param->maxit = iniparam->itsolver_maxit;
00545         param->tol   = iniparam->itsolver_tol;
00546     }
00547     else if (iniparam->solver_type == SOLVER_FMG) {
00548         param->maxit = iniparam->itsolver_maxit;
00549         param->tol   = iniparam->itsolver_tol;
00550     }
00551     else {
00552         param->maxit = iniparam->AMG_maxit;
00553         param->tol   = iniparam->AMG_tol;
00554     }
00555
00556     param->max_levels      = iniparam->AMG_levels;
00557     param->cycle_type      = iniparam->AMG_cycle_type;
00558     param->smoother        = iniparam->AMG_smoother;
00559     param->smooth_order    = iniparam->AMG_smooth_order;
00560     param->relaxation       = iniparam->AMG_relaxation;
00561     param->coarse_solver   = iniparam->AMG_coarse_solver;
00562     param->polynomial_degree = iniparam->AMG_polynomial_degree;
00563     param->presmooth_iter  = iniparam->AMG_presmooth_iter;
00564     param->postsmooth_iter = iniparam->AMG_postsmooth_iter;
00565     param->coarse_dof      = iniparam->AMG_coarse_dof;
00566     param->coarse_scaling  = iniparam->AMG_coarse_scaling;
00567     param->amli_degree     = iniparam->AMG_amli_degree;
00568     param->amli_coef       = NULL;
00569     param->nl_amli_krylov_type = iniparam->AMG_nl_amli_krylov_type;
00570
00571     param->coarsening_type = iniparam->AMG_coarsening_type;
00572     param->interpolation_type = iniparam->AMG_interpolation_type;
00573     param->strong_threshold = iniparam->AMG_strong_threshold;
00574     param->truncation_threshold = iniparam->AMG_truncation_threshold;
00575     param->max_row_sum      = iniparam->AMG_max_row_sum;
00576     param->aggressive_level = iniparam->AMG_aggressive_level;
00577     param->aggressive_path  = iniparam->AMG_aggressive_path;
00578
00579     param->aggregation_type = iniparam->AMG_aggregation_type;
00580     param->pair_number      = iniparam->AMG_pair_number;
00581     param->quality_bound    = iniparam->AMG_quality_bound;
00582     param->strong_coupled   = iniparam->AMG_strong_coupled;
00583     param->max_aggregation  = iniparam->AMG_max_aggregation;
00584     param->tentative_smooth = iniparam->AMG_tentative_smooth;
00585     param->smooth_filter    = iniparam->AMG_smooth_filter;
00586     param->smooth_restriction = iniparam->AMG_smooth_restriction;
00587
00588     param->ILU_levels      = iniparam->AMG_ILU_levels;
00589     param->ILU_type        = iniparam->ILU_type;
00590     param->ILU_lfil        = iniparam->ILU_lfil;
00591     param->ILU_droptol     = iniparam->ILU_droptol;
00592     param->ILU_relax       = iniparam->ILU_relax;
00593     param->ILU_permtol     = iniparam->ILU_permtol;
00594
00595     param->SWZ_levels      = iniparam->AMG_SWZ_levels;
00596     param->SWZ_mmsize     = iniparam->SWZ_mmsize;
00597     param->SWZ_maxlvl     = iniparam->SWZ_maxlvl;
00598     param->SWZ_type       = iniparam->SWZ_type;
00599 }
00600
00612 void fasp_param_ilu_set (ILU_param      *iluparam,
00613                        const input_param *iniparam)
00614 {
00615     iluparam->print_level = iniparam->print_level;
00616     iluparam->ILU_type    = iniparam->ILU_type;
00617     iluparam->ILU_lfil    = iniparam->ILU_lfil;

```

```

00618     iluparam->ILU_droptol = iniparam->ILU_droptol;
00619     iluparam->ILU_relax    = iniparam->ILU_relax;
00620     iluparam->ILU_permtol = iniparam->ILU_permtol;
00621 }
00622
00634 void fasp_param_swz_set (SWZ_param      *swzparam,
00635                          const input_param *iniparam)
00636 {
00637     swzparam->print_level = iniparam->print_level;
00638     swzparam->SWZ_type    = iniparam->SWZ_type;
00639     swzparam->SWZ_maxlvl  = iniparam->SWZ_maxlvl;
00640     swzparam->SWZ_mmsize  = iniparam->SWZ_mmsize;
00641     swzparam->SWZ_blkssolver = iniparam->SWZ_blkssolver;
00642 }
00643
00656 void fasp_param_solver_set (ITS_param      *itsparam,
00657                             const input_param *iniparam)
00658 {
00659     itsparam->print_level = iniparam->print_level;
00660     itsparam->itsolver_type = iniparam->solver_type;
00661     itsparam->decoup_type  = iniparam->decoup_type;
00662     itsparam->precond_type = iniparam->precond_type;
00663     itsparam->stop_type    = iniparam->stop_type;
00664     itsparam->restart      = iniparam->restart;
00665
00666     if ( itsparam->itsolver_type == SOLVER_AMG ) {
00667         itsparam->tol = iniparam->AMG_tol;
00668         itsparam->maxit = iniparam->AMG_maxit;
00669     }
00670     else {
00671         itsparam->tol = iniparam->itsolver_tol;
00672         itsparam->maxit = iniparam->itsolver_maxit;
00673     }
00674 }
00675
00687 void fasp_param_amg_to_prec (precond_data *pcdata,
00688                             const AMG_param *amgparam)
00689 {
00690     pcdata->AMG_type = amgparam->AMG_type;
00691     pcdata->print_level = amgparam->print_level;
00692     pcdata->maxit = amgparam->maxit;
00693     pcdata->max_levels = amgparam->max_levels;
00694     pcdata->tol = amgparam->tol;
00695     pcdata->cycle_type = amgparam->cycle_type;
00696     pcdata->smoother = amgparam->smoother;
00697     pcdata->smooth_order = amgparam->smooth_order;
00698     pcdata->presmooth_iter = amgparam->presmooth_iter;
00699     pcdata->postsmooth_iter = amgparam->postsmooth_iter;
00700     pcdata->coarsening_type = amgparam->coarsening_type;
00701     pcdata->coarse_solver = amgparam->coarse_solver;
00702     pcdata->relaxation = amgparam->relaxation;
00703     pcdata->polynomial_degree = amgparam->polynomial_degree;
00704     pcdata->coarse_scaling = amgparam->coarse_scaling;
00705     pcdata->amli_degree = amgparam->amli_degree;
00706     pcdata->amli_coef = amgparam->amli_coef;
00707     pcdata->nl_amli_krylov_type = amgparam->nl_amli_krylov_type;
00708     pcdata->tentative_smooth = amgparam->tentative_smooth;
00709 }
00710
00722 void fasp_param_prec_to_amg (AMG_param      *amgparam,
00723                             const precond_data *pcdata)
00724 {
00725     amgparam->AMG_type = pcdata->AMG_type;
00726     amgparam->print_level = pcdata->print_level;
00727     amgparam->cycle_type = pcdata->cycle_type;
00728     amgparam->smoother = pcdata->smoother;
00729     amgparam->smooth_order = pcdata->smooth_order;
00730     amgparam->presmooth_iter = pcdata->presmooth_iter;
00731     amgparam->postsmooth_iter = pcdata->postsmooth_iter;
00732     amgparam->relaxation = pcdata->relaxation;
00733     amgparam->polynomial_degree = pcdata->polynomial_degree;
00734     amgparam->coarse_solver = pcdata->coarse_solver;
00735     amgparam->coarse_scaling = pcdata->coarse_scaling;
00736     amgparam->amli_degree = pcdata->amli_degree;
00737     amgparam->amli_coef = pcdata->amli_coef;
00738     amgparam->nl_amli_krylov_type = pcdata->nl_amli_krylov_type;
00739     amgparam->tentative_smooth = pcdata->tentative_smooth;
00740     amgparam->ILU_levels = pcdata->mgl_data->ILU_levels;
00741 }
00742
00755 void fasp_param_amg_to_precbsr (precond_data_bsr *pcdata,

```

```

00756                                     const AMG_param *amgparam)
00757 {
00758     pcddata->AMG_type                = amgparam->AMG_type;
00759     pcddata->print_level              = amgparam->print_level;
00760     pcddata->maxit                   = amgparam->maxit;
00761     pcddata->max_levels               = amgparam->max_levels;
00762     pcddata->tol                     = amgparam->tol;
00763     pcddata->cycle_type               = amgparam->cycle_type;
00764     pcddata->smoother                 = amgparam->smoother;
00765     pcddata->smooth_order              = amgparam->smooth_order;
00766     pcddata->presmooth_iter           = amgparam->presmooth_iter;
00767     pcddata->postsmooth_iter          = amgparam->postsmooth_iter;
00768     pcddata->coarse_solver             = amgparam->coarse_solver;
00769     pcddata->coarsening_type           = amgparam->coarsening_type;
00770     pcddata->relaxation                = amgparam->relaxation;
00771     pcddata->coarse_scaling            = amgparam->coarse_scaling;
00772     pcddata->amli_degree               = amgparam->amli_degree;
00773     pcddata->amli_coef                = amgparam->amli_coef;
00774     pcddata->nl_amli_krylov_type       = amgparam->nl_amli_krylov_type;
00775     pcddata->tentative_smooth          = amgparam->tentative_smooth;
00776 }
00777
00790 void fasp_param_precbsr_to_amg (AMG_param *amgparam,
00791                                const precond_data_bsr *pcdata)
00792 {
00793     amgparam->AMG_type                = pcddata->AMG_type;
00794     amgparam->print_level              = pcddata->print_level;
00795     amgparam->cycle_type               = pcddata->cycle_type;
00796     amgparam->smoother                 = pcddata->smoother;
00797     amgparam->smooth_order              = pcddata->smooth_order;
00798     amgparam->presmooth_iter           = pcddata->presmooth_iter;
00799     amgparam->postsmooth_iter          = pcddata->postsmooth_iter;
00800     amgparam->relaxation                = pcddata->relaxation;
00801     amgparam->coarse_solver             = pcddata->coarse_solver;
00802     amgparam->coarse_scaling            = pcddata->coarse_scaling;
00803     amgparam->amli_degree               = pcddata->amli_degree;
00804     amgparam->amli_coef                = pcddata->amli_coef;
00805     amgparam->nl_amli_krylov_type       = pcddata->nl_amli_krylov_type;
00806     amgparam->tentative_smooth          = pcddata->tentative_smooth;
00807     amgparam->ILU_levels                = pcddata->mgl_data->ILU_levels;
00808 }
00809
00820 void fasp_param_amg_print (const AMG_param *param)
00821 {
00822     if ( param ) {
00823         printf("\n      Parameters in AMG_param\n");
00824         printf("-----\n");
00825
00826         printf("AMG print level:                %d\n", param->print_level);
00827         printf("AMG max num of iter:            %d\n", param->maxit);
00828         printf("AMG type:                      %d\n", param->AMG_type);
00829         printf("AMG tolerance:                  %.2e\n", param->tol);
00830         printf("AMG max levels:                 %d\n", param->max_levels);
00831         printf("AMG cycle type:                 %d\n", param->cycle_type);
00832         printf("AMG coarse solver type:         %d\n", param->coarse_solver);
00833         printf("AMG scaling of coarse correction: %d\n", param->coarse_scaling);
00834         printf("AMG smoother type:             %d\n", param->smoother);
00835         printf("AMG smoother order:            %d\n", param->smooth_order);
00836         printf("AMG num of presmoothing:        %d\n", param->presmooth_iter);
00837         printf("AMG num of postsmoothing:       %d\n", param->postsmooth_iter);
00838
00839         if ( param->smoother == SMOOTHER_SOR ||
00840             param->smoother == SMOOTHER_SSOR ||
00841             param->smoother == SMOOTHER_GSOR ||
00842             param->smoother == SMOOTHER_SGSOR ) {
00843             printf("AMG relax factor:                %.4f\n",
00844                 param->relaxation);
00845         }
00846
00847         if ( param->smoother == SMOOTHER_POLY ) {
00848             printf("AMG polynomial smoother degree: %d\n",
00849                 param->polynomial_degree);
00850         }
00851
00852         if ( param->cycle_type == AMLI_CYCLE ) {
00853             printf("AMG AMLI degree of polynomial: %d\n",
00854                 param->amli_degree);
00855         }
00856     }
00857 }
00858

```

```

00859     if ( param->cycle_type == NL_AMLI_CYCLE ) {
00860         printf("AMG Nonlinear AMLI Krylov type:          %d\n",
00861             param->nl_amli_krylov_type);
00862     }
00863
00864     switch (param->AMG_type) {
00865     case CLASSIC_AMG:
00866         printf("AMG coarsening type:                    %d\n",
00867             param->coarsening_type);
00868         printf("AMG interpolation type:                    %d\n",
00869             param->interpolation_type);
00870         printf("AMG dof on coarsest grid:                %d\n",
00871             param->coarse_dof);
00872         printf("AMG strong threshold:                        %.4f\n",
00873             param->strong_threshold);
00874         printf("AMG truncation threshold:                    %.4f\n",
00875             param->truncation_threshold);
00876         printf("AMG max row sum:                            %.4f\n",
00877             param->max_row_sum);
00878         printf("AMG aggressive levels:                      %d\n",
00879             param->aggressive_level);
00880         printf("AMG aggressive path:                        %d\n",
00881             param->aggressive_path);
00882         break;
00883
00884     default: // SA_AMG or UA_AMG
00885         printf("Aggregation type:                            %d\n",
00886             param->aggregation_type);
00887         if ( param->aggregation_type == PAIRWISE ) {
00888             printf("Aggregation number of pairs:                %d\n",
00889                 param->pair_number);
00890             printf("Aggregation quality bound:                    %.2f\n",
00891                 param->quality_bound);
00892         }
00893         if ( param->aggregation_type == VMB ) {
00894             printf("Aggregation strong coupling:                %.4f\n",
00895                 param->strong_coupled);
00896             printf("Aggregation max aggregation:                %d\n",
00897                 param->max_aggregation);
00898             printf("Aggregation tentative smooth:                %.4f\n",
00899                 param->tentative_smooth);
00900             printf("Aggregation smooth filter:                  %d\n",
00901                 param->smooth_filter);
00902             printf("Aggregation smooth restriction:              %d\n",
00903                 param->smooth_restriction);
00904         }
00905         break;
00906     }
00907
00908     if (param->ILU_levels>0) {
00909         printf("AMG ILU smoother level:                    %d\n", param->ILU_levels);
00910         printf("AMG ILU type:                              %d\n", param->ILU_type);
00911         printf("AMG ILU level of fill-in:                  %d\n", param->ILU_lfil);
00912         printf("AMG ILU drop tol:                          %e\n", param->ILU_droptol);
00913         printf("AMG ILU relaxation:                        %f\n", param->ILU_relax);
00914     }
00915
00916     if (param->SWZ_levels>0) {
00917         printf("AMG Schwarz smoother level:                %d\n", param->SWZ_levels);
00918         printf("AMG Schwarz type:                          %d\n", param->SWZ_type);
00919         printf("AMG Schwarz forming block level:            %d\n", param->SWZ_maxlvl);
00920         printf("AMG Schwarz maximal block size:             %d\n", param->SWZ_mmsize);
00921     }
00922
00923     printf("-----\n\n");
00924 }
00925
00926 else {
00927     printf("### WARNING: AMG_param has not been set!\n");
00928 } // end if (param)
00929
00930
00931 }
00932
00943 void fasp_param_ilu_print (const ILU_param *param)
00944 {
00945     if ( param ) {
00946
00947         printf("\n          Parameters in ILU_param\n");
00948         printf("-----\n");
00949         printf("ILU print level:                            %d\n", param->print_level);

```



```

00950         printf("ILU type:                                %d\n", param->ILU_type);
00951         printf("ILU level of fill-in:                    %d\n", param->ILU_lfil);
00952         printf("ILU relaxation factor:                  %.4f\n", param->ILU_relax);
00953         printf("ILU drop tolerance:                      %.2e\n", param->ILU_droptol);
00954         printf("ILU permutation tolerance:                %.2e\n", param->ILU_permtol);
00955         printf("-----\n\n");
00956     }
00957     else {
00958         printf("### WARNING: ILU_param has not been set!\n");
00959     }
00960 }
00961 }
00962
00973 void fasp_param_swz_print (const SWZ_param *param)
00974 {
00975     if ( param ) {
00976
00977         printf("\n          Parameters in SWZ_param\n");
00978         printf("-----\n");
00979         printf("Schwarz print level:                                %d\n", param->print_level);
00980         printf("Schwarz type:                                        %d\n", param->SWZ_type);
00981         printf("Schwarz forming block level:                        %d\n", param->SWZ_maxlvl);
00982         printf("Schwarz maximal block size:                        %d\n", param->SWZ_mmsize);
00983         printf("Schwarz block solver type:                          %d\n", param->SWZ_blksolver);
00984         printf("-----\n\n");
00985     }
00986     else {
00987         printf("### WARNING: SWZ_param has not been set!\n");
00988     }
00989 }
00990 }
00991
01002 void fasp_param_solver_print (const ITS_param *param)
01003 {
01004     if ( param ) {
01005
01006         printf("\n          Parameters in ITS_param\n");
01007         printf("-----\n");
01008
01009         printf("Solver print level:                                %d\n", param->print_level);
01010         printf("Solver type:                                        %d\n", param->itsolver_type);
01011         printf("Solver precondition type:                          %d\n", param->precond_type);
01012         printf("Solver max num of iter:                            %d\n", param->maxit);
01013         printf("Solver tolerance:                                  %.2e\n", param->tol);
01014         printf("Solver stopping type:                              %d\n", param->stop_type);
01015
01016         if (param->itsolver_type==SOLVER_GMRES ||
01017             param->itsolver_type==SOLVER_VGMRES) {
01018             printf("Solver restart number:                            %d\n", param->restart);
01019         }
01020
01021         printf("-----\n\n");
01022     }
01023     else {
01024         printf("### WARNING: ITS_param has not been set!\n");
01025     }
01026 }
01027 }
01028
01029 /*-----*/
01030 /*--          End of File          --*/
01031 /*-----*/

```

9.37 AuxSort.c File Reference

Array sorting/merging and removing duplicated integers.

```

#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- [INT fasp_aux_BiSearch](#) (const [INT](#) nlist, const [INT](#) *list, const [INT](#) value)
Binary Search.

- `INT fasp_aux_unique` (`INT` numbers[], `const INT` size)
Remove duplicates in an sorted (ascending order) array.
- `void fasp_aux_merge` (`INT` numbers[], `INT` work[], `INT` left, `INT` mid, `INT` right)
Merge two sorted arrays.
- `void fasp_aux_msort` (`INT` numbers[], `INT` work[], `INT` left, `INT` right)
Sort the INT array in ascending order with the merge sort algorithm.
- `void fasp_aux_iQuickSort` (`INT` *a, `INT` left, `INT` right)
Sort the array (INT type) in ascending order with the quick sorting algorithm.
- `void fasp_aux_dQuickSort` (`REAL` *a, `INT` left, `INT` right)
Sort the array (REAL type) in ascending order with the quick sorting algorithm.
- `void fasp_aux_iQuickSortIndex` (`INT` *a, `INT` left, `INT` right, `INT` *index)
Reorder the index of (INT type) so that 'a' is in ascending order.
- `void fasp_aux_dQuickSortIndex` (`REAL` *a, `INT` left, `INT` right, `INT` *index)
Reorder the index of (REAL type) so that 'a' is ascending in such order.

9.37.1 Detailed Description

Array sorting/merging and removing duplicated integers.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxMemory.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxSort.c](#).

9.37.2 Function Documentation

9.37.2.1 fasp_aux_BiSearch()

```
INT fasp_aux_BiSearch (
    const INT nlist,
    const INT * list,
    const INT value )
```

Binary Search.

Parameters

<i>nlist</i>	Length of the array list
<i>list</i>	Pointer to a set of values
<i>value</i>	The target

Returns

The location of value in array list if succeeded; otherwise, return -1.

Author

Chunsheng Feng

Date

03/01/2011

Definition at line 42 of file [AuxSort.c](#).**9.37.2.2 fasp_aux_dQuickSort()**

```
void fasp_aux_dQuickSort (
    REAL * a,
    INT left,
    INT right )
```

Sort the array (REAL type) in ascending order with the quick sorting algorithm.

Parameters

<i>a</i>	Pointer to the array needed to be sorted
<i>left</i>	Starting index
<i>right</i>	Ending index

Author

Zhiyang Zhou

Date

2009/11/28

Note

'left' and 'right' are usually set to be 0 and n-1, respectively where n is the length of 'a'.

Definition at line 246 of file [AuxSort.c](#).**9.37.2.3 fasp_aux_dQuickSortIndex()**

```
void fasp_aux_dQuickSortIndex (
    REAL * a,
    INT left,
    INT right,
    INT * index )
```

Reorder the index of (REAL type) so that 'a' is ascending in such order.

Parameters

<i>a</i>	Pointer to the array
<i>left</i>	Starting index
<i>right</i>	Ending index
<i>index</i>	Index of 'a' (out)

Author

Zhiyang Zhou

Date

2009/12/02

Note

'left' and 'right' are usually set to be 0 and n-1, respectively, where n is the length of 'a'. 'index' should be initialized in the nature order and it has the same length as 'a'.

Definition at line 327 of file [AuxSort.c](#).

9.37.2.4 fasp_aux_iQuickSort()

```
void fasp_aux_iQuickSort (
    INT * a,
    INT left,
    INT right )
```

Sort the array (INT type) in ascending order with the quick sorting algorithm.

Parameters

<i>a</i>	Pointer to the array needed to be sorted
<i>left</i>	Starting index
<i>right</i>	Ending index

Author

Zhiyang Zhou

Date

11/28/2009

Note

'left' and 'right' are usually set to be 0 and n-1, respectively where n is the length of 'a'.

Definition at line 208 of file [AuxSort.c](#).

9.37.2.5 fasp_aux_iQuickSortIndex()

```
void fasp_aux_iQuickSortIndex (
    INT * a,
    INT left,
    INT right,
    INT * index )
```

Reorder the index of (INT type) so that 'a' is in ascending order.

Parameters

<i>a</i>	Pointer to the array
<i>left</i>	Starting index
<i>right</i>	Ending index
<i>index</i>	Index of 'a' (out)

Author

Zhiyang Zhou

Date

2009/12/02

Note

'left' and 'right' are usually set to be 0 and n-1, respectively, where n is the length of 'a'. 'index' should be initialized in the nature order and it has the same length as 'a'.

Definition at line 286 of file [AuxSort.c](#).

9.37.2.6 fasp_aux_merge()

```
void fasp_aux_merge (
    INT numbers[],
    INT work[],
    INT left,
    INT mid,
    INT right )
```

Merge two sorted arrays.

Parameters

<i>numbers</i>	Pointer to the array needed to be sorted
<i>work</i>	Pointer to the work array with same size as numbers
<i>left</i>	Starting index of array 1
<i>mid</i>	Starting index of array 2
<i>right</i>	Ending index of array 1 and 2

Author

Chensong Zhang

Date

11/21/2010

Note

Both arrays are stored in numbers! Arrays should be pre-sorted!

Definition at line 115 of file [AuxSort.c](#).

9.37.2.7 fasp_aux_msort()

```
void fasp_aux_msort (
    INT numbers[],
    INT work[],
    INT left,
    INT right )
```

Sort the INT array in ascending order with the merge sort algorithm.

Parameters

<i>numbers</i>	Pointer to the array needed to be sorted
<i>work</i>	Pointer to the work array with same size as numbers
<i>left</i>	Starting index
<i>right</i>	Ending index

Author

Chensong Zhang

Date

11/21/2010

Note

'left' and 'right' are usually set to be 0 and n-1, respectively

Definition at line 177 of file [AuxSort.c](#).

9.37.2.8 fasp_aux_unique()

```
INT fasp_aux_unique (
    INT numbers[],
    const INT size )
```

Remove duplicates in an sorted (ascending order) array.

Parameters

<i>numbers</i>	Pointer to the array needed to be sorted (in/out)
<i>size</i>	Length of the target array

Returns

New size after removing duplicates

Author

Chensong Zhang

Date

11/21/2010

Note

Operation is in place. Does not use any extra or temporary storage.

Definition at line 82 of file [AuxSort.c](#).

9.38 AuxSort.c

[Go to the documentation of this file.](#)

```

00001
00014 #include "fasp.h"
00015 #include "fasp_funcs.h"
00016
00017 /*-----*/
00018 /*--  Declare Private Functions  --*/
00019 /*-----*/
00020
00021 static void dSwapping (REAL *w, const INT i, const INT j);
00022 static void iSwapping (INT *w, const INT i, const INT j);
00023
00024 /*-----*/
00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00042 INT fasp_aux_BiSearch (const INT  nlist,
00043                       const INT *list,
00044                       const INT  value)
00045 {
00046     INT low, high, m;
00047
00048     low = 0;
00049     high = nlist - 1;
00050
00051     while (low <= high) {
00052         m = (low + high) / 2;
00053         if (value < list[m]) {
00054             high = m - 1;
00055         }
00056         else if (value > list[m]) {
00057             low = m + 1;
00058         }
00059         else {
00060             return m;
00061         }
00062     }
00063
00064     return -1;
00065 }
00066
00082 INT fasp_aux_unique (INT    numbers[],
00083                     const INT size)
00084 {
00085     INT i, newsize;
00086
00087     if ( size == 0 ) return(0);
00088
00089     for ( newsize = 0, i = 1; i < size; ++i ) {
00090         if ( numbers[newsize] < numbers[i] ) {
00091             newsize++;
00092             numbers[newsize] = numbers[i];
00093         }
00094     }
00095
00096     return(newsize+1);
00097 }
00098
00115 void fasp_aux_merge (INT  numbers[],
00116                     INT  work[],
00117                     INT  left,
00118                     INT  mid,
00119                     INT  right)
00120 {
00121     INT i, left_end, num_elements, tmp_pos;
00122
00123     left_end = mid - 1;
00124     tmp_pos = left;
00125     num_elements = right - left + 1;
00126

```

```

00127     while ((left <= left_end) && (mid <= right)) {
00128
00129         if (numbers[left] <= numbers[mid]) // first branch <=
00130         {
00131             work[tmp_pos] = numbers[left];
00132             tmp_pos = tmp_pos + 1;
00133             left = left + 1;
00134         }
00135         else // second branch >
00136         {
00137             work[tmp_pos] = numbers[mid];
00138             tmp_pos = tmp_pos + 1;
00139             mid = mid + 1;
00140         }
00141     }
00142
00143     while (left <= left_end) {
00144         work[tmp_pos] = numbers[left];
00145         left = left + 1;
00146         tmp_pos = tmp_pos + 1;
00147     }
00148
00149     while (mid <= right) {
00150         work[tmp_pos] = numbers[mid];
00151         mid = mid + 1;
00152         tmp_pos = tmp_pos + 1;
00153     }
00154
00155     for (i = 0; i < num_elements; ++i) {
00156         numbers[right] = work[right];
00157         right = right - 1;
00158     }
00159 }
00160 }
00161
00177 void fasp_aux_msort (INT  numbers[],
00178                     INT  work[],
00179                     INT  left,
00180                     INT  right)
00181 {
00182     INT mid;
00183
00184     if (right > left) {
00185         mid = (right + left) / 2;
00186         fasp_aux_msort(numbers, work, left, mid);
00187         fasp_aux_msort(numbers, work, mid+1, right);
00188         fasp_aux_merge(numbers, work, left, mid+1, right);
00189     }
00190
00191 }
00192
00208 void fasp_aux_iQuickSort (INT  *a,
00209                          INT  left,
00210                          INT  right)
00211 {
00212     INT i, last;
00213
00214     if (left >= right) return;
00215
00216     iSwapping(a, left, (left+right)/2);
00217
00218     last = left;
00219     for (i = left+1; i <= right; ++i) {
00220         if (a[i] < a[left]) {
00221             iSwapping(a, ++last, i);
00222         }
00223     }
00224
00225     iSwapping(a, left, last);
00226
00227     fasp_aux_iQuickSort(a, left, last-1);
00228     fasp_aux_iQuickSort(a, last+1, right);
00229 }
00230
00246 void fasp_aux_dQuickSort (REAL *a,
00247                          INT  left,
00248                          INT  right)
00249 {
00250     INT i, last;
00251
00252     if (left >= right) return;

```



```

00253
00254     dSwapping(a, left, (left+right)/2);
00255
00256     last = left;
00257     for (i = left+1; i <= right; ++i) {
00258         if (a[i] < a[left]) {
00259             dSwapping(a, ++last, i);
00260         }
00261     }
00262
00263     dSwapping(a, left, last);
00264
00265     fasp_aux_dQuickSort(a, left, last-1);
00266     fasp_aux_dQuickSort(a, last+1, right);
00267 }
00268
00286 void fasp_aux_iQuickSortIndex (INT    *a,
00287                                INT    left,
00288                                INT    right,
00289                                INT    *index)
00290 {
00291     INT i, last;
00292
00293     if (left >= right) return;
00294
00295     iSwapping(index, left, (left+right)/2);
00296
00297     last = left;
00298     for (i = left+1; i <= right; ++i) {
00299         if (a[index[i]] < a[index[left]]) {
00300             iSwapping(index, ++last, i);
00301         }
00302     }
00303
00304     iSwapping(index, left, last);
00305
00306     fasp_aux_iQuickSortIndex(a, left, last-1, index);
00307     fasp_aux_iQuickSortIndex(a, last+1, right, index);
00308 }
00309
00327 void fasp_aux_dQuickSortIndex (REAL    *a,
00328                                INT    left,
00329                                INT    right,
00330                                INT    *index)
00331 {
00332     INT i, last;
00333
00334     if (left >= right) return;
00335
00336     iSwapping(index, left, (left+right)/2);
00337
00338     last = left;
00339     for (i = left+1; i <= right; ++i) {
00340         if (a[index[i]] < a[index[left]]) {
00341             iSwapping(index, ++last, i);
00342         }
00343     }
00344
00345     iSwapping(index, left, last);
00346
00347     fasp_aux_dQuickSortIndex(a, left, last-1, index);
00348     fasp_aux_dQuickSortIndex(a, last+1, right, index);
00349 }
00350
00351 /*-----*/
00352 /*--      Private Functions      --*/
00353 /*-----*/
00354
00367 static void iSwapping (INT    *w,
00368                        const INT i,
00369                        const INT j)
00370 {
00371     const INT temp = w[i];
00372     w[i] = w[j]; w[j] = temp;
00373 }
00374
00387 static void dSwapping (REAL    *w,
00388                        const INT i,
00389                        const INT j)
00390 {
00391     const REAL temp = w[i];

```

```

00392      w[i] = w[j]; w[j] = temp;
00393  }
00394
00395  /*-----*/
00396  /*--      End of File      --*/
00397  /*-----*/

```

9.39 AuxThreads.c File Reference

Get and set number of threads and assign work load for each thread.

```

#include <stdio.h>
#include <stdlib.h>
#include "fasp.h"

```

Functions

- void [fasp_get_start_end](#) (const [INT](#) procid, const [INT](#) nprocs, const [INT](#) n, [INT](#) *start, [INT](#) *end)
Assign Load to each thread.
- void [fasp_set_gs_threads](#) (const [INT](#) mythreads, const [INT](#) its)
Set threads for CPR. Please add it at the begin of Krylov OpenMP method function and after iter++.

Variables

- [INT](#) THDs_AMG_GS =0
- [INT](#) THDs_CPR_IGS =0
- [INT](#) THDs_CPR_gGS =0

9.39.1 Detailed Description

Get and set number of threads and assign work load for each thread.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxThreads.c](#).

9.39.2 Function Documentation

9.39.2.1 [fasp_get_start_end\(\)](#)

```

void fasp_get_start_end (
    const INT procid,
    const INT nprocs,
    const INT n,
    INT * start,
    INT * end )

```

Assign Load to each thread.

Parameters

<i>procid</i>	Index of thread
<i>nprocs</i>	Number of threads
<i>n</i>	Total workload
<i>start</i>	Pointer to the begin of each thread in total workload
<i>end</i>	Pointer to the end of each thread in total workload

Author

Chunsheng Feng, Xiaoqiang Yue and Zheng Li

Date

June/25/2012

Definition at line 92 of file [AuxThreads.c](#).

9.39.2.2 fasp_set_gs_threads()

```
void fasp_set_gs_threads (
    const INT mythreads,
    const INT its )
```

Set threads for CPR. Please add it at the begin of Krylov OpenMP method function and after iter++.

Parameters

<i>mythreads</i>	Total threads of solver
<i>its</i>	Current iteration number in the Krylov methods

Author

Feng Chunsheng, Yue Xiaoqiang

Date

03/20/2011

Definition at line 132 of file [AuxThreads.c](#).

9.39.3 Variable Documentation**9.39.3.1 THDs_AMG_GS**

```
INT THDs_AMG_GS =0
```

AMG GS smoothing threads

Definition at line 116 of file [AuxThreads.c](#).

9.39.3.2 THDs_CPR_gGS

`INT THDs_CPR_gGS =0`

global matrix GS smoothing threads

Definition at line 118 of file [AuxThreads.c](#).

9.39.3.3 THDs_CPR_lGS

`INT THDs_CPR_lGS =0`

reservoir GS smoothing threads

Definition at line 117 of file [AuxThreads.c](#).

9.40 AuxThreads.c

[Go to the documentation of this file.](#)

```

00001
00013 #include <stdio.h>
00014 #include <stdlib.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021
00022 /*-----*/
00023 /*--      Public Functions      --*/
00024 /*-----*/
00025
00026 #ifdef _OPENMP
00027
00028 INT thread_ini_flag = 0;
00029
00040 INT fasp_get_num_threads ( void )
00041 {
00042     static INT nthreads;
00043
00044     if ( thread_ini_flag == 0 ) {
00045         nthreads = 1;
00046 #pragma omp parallel
00047         nthreads = omp_get_num_threads();
00048
00049         printf("\nFASP is running on %d thread(s).\n\n", nthreads);
00050         thread_ini_flag = 1;
00051     }
00052
00053     return nthreads;
00054 }
00055
00068 INT fasp_set_num_threads (const INT nthreads)
00069 {
00070     omp_set_num_threads( nthreads );
00071
00072     return nthreads;
00073 }
00074
00075 #endif
00076
00092 void fasp_get_start_end (const INT  procid,
00093                         const INT  nprocs,
00094                         const INT  n,
00095                         INT         *start,
00096                         INT         *end)
00097 {
00098     INT chunk_size = n / nprocs;
00099     INT mod = n % nprocs;
00100     INT start_loc, end_loc;
00101
00102     if ( procid < mod ) {
00103         end_loc = chunk_size + 1;
00104         start_loc = end_loc * procid;

```

```

00105     }
00106     else {
00107         end_loc = chunk_size;
00108         start_loc = end_loc * procid + mod;
00109     }
00110     end_loc = end_loc + start_loc;
00111
00112     *start = start_loc;
00113     *end = end_loc;
00114 }
00115
00116 INT THDs_AMG_GS=0;
00117 INT THDs_CPR_lGS=0;
00118 INT THDs_CPR_gGS=0;
00132 void fasp_set_gs_threads (const INT mythreads,
00133                          const INT its)
00134 {
00135     #ifdef _OPENMP
00136     #if 1
00137     #if 1
00138         if (its <=8) {
00139             THDs_AMG_GS = mythreads;
00140             THDs_CPR_lGS = mythreads ;
00141             THDs_CPR_gGS = mythreads ;
00142         }
00143         else if (its <=12) {
00144             THDs_AMG_GS = mythreads;
00145             THDs_CPR_lGS = (6 < mythreads) ? 6 : mythreads;
00146             THDs_CPR_gGS = (4 < mythreads) ? 4 : mythreads;
00147         }
00148         else if (its <=15) {
00149             THDs_AMG_GS = (3 < mythreads) ? 3 : mythreads;
00150             THDs_CPR_lGS = (3 < mythreads) ? 3 : mythreads;
00151             THDs_CPR_gGS = (2 < mythreads) ? 2 : mythreads;
00152         }
00153         else if (its <=18) {
00154             THDs_AMG_GS = (2 < mythreads) ? 2 : mythreads;
00155             THDs_CPR_lGS = (2 < mythreads) ? 2 : mythreads;
00156             THDs_CPR_gGS = (1 < mythreads) ? 1 : mythreads;
00157         }
00158         else {
00159             THDs_AMG_GS = 1;
00160             THDs_CPR_lGS = 1;
00161             THDs_CPR_gGS = 1;
00162         }
00163     }
00164     #else
00165         THDs_AMG_GS = mythreads;
00166         THDs_CPR_lGS = mythreads ;
00167         THDs_CPR_gGS = mythreads ;
00168     #endif
00169 }
00170 #endif // _OPENMP
00171 }
00172
00173 /*-----*/
00174 /*--      End of File      --*/
00175 /*-----*/

```

9.41 AuxTiming.c File Reference

Timing subroutines.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_gettime](#) (REAL *time)

Get system time.

9.41.1 Detailed Description

Timing subroutines.

Note

This file contains Level-0 (Aux) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxTiming.c](#).

9.41.2 Function Documentation

9.41.2.1 fasp_gettime()

```
void fasp_gettime (
    REAL * time )
```

Get system time.

Author

Chunsheng Feng, Zheng LI

Date

11/10/2012

Modified by Chensong Zhang on 09/22/2014: Use CLOCKS_PER_SEC for cross-platform
Definition at line 36 of file [AuxTiming.c](#).

9.42 AuxTiming.c

[Go to the documentation of this file.](#)

```
00001
00013 #include <time.h>
00014
00015 #ifdef _OPENMP
00016 #include <omp.h>
00017 #endif
00018
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--      Public Functions      --*/
00024 /*-----*/
00025
00036 void fasp_gettime (REAL *time)
00037 {
00038     if ( time != NULL ) {
00039 #ifdef _OPENMP
00040         *time = omp_get_wtime();
00041 #else
00042         *time = (REAL) clock() / CLOCKS_PER_SEC;
00043 #endif
00044     }
00045 }
00046
00047 /*-----*/
00048 /*--      End of File      --*/
00049 /*-----*/
```

9.43 AuxVector.c File Reference

Simple vector operations – init, set, copy, etc.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- [SHORT fasp_dvec_isnan](#) (const [dvector](#) *u)
Check a dvector whether there is NAN.
- [dvector fasp_dvec_create](#) (const [INT](#) m)
Create dvector data space of REAL type.
- [ivector fasp_ivec_create](#) (const [INT](#) m)
Create vector data space of INT type.
- void [fasp_dvec_alloc](#) (const [INT](#) m, [dvector](#) *u)
Create dvector data space of REAL type.
- void [fasp_ivec_alloc](#) (const [INT](#) m, [ivector](#) *u)
Create vector data space of INT type.
- void [fasp_dvec_free](#) ([dvector](#) *u)
Free vector data space of REAL type.
- void [fasp_ivec_free](#) ([ivector](#) *u)
Free vector data space of INT type.
- void [fasp_dvec_rand](#) (const [INT](#) n, [dvector](#) *x)
Generate fake random REAL vector in the range from 0 to 1.
- void [fasp_dvec_set](#) ([INT](#) n, [dvector](#) *x, const [REAL](#) val)
Initialize dvector x[i]=val for i=0:n-1.
- void [fasp_ivec_set](#) ([INT](#) n, [ivector](#) *u, const [INT](#) m)
Set ivector value to be m.
- void [fasp_dvec_cp](#) (const [dvector](#) *x, [dvector](#) *y)
Copy dvector x to dvector y.
- [REAL fasp_dvec_maxdiff](#) (const [dvector](#) *x, const [dvector](#) *y)
Maximal difference of two dvector x and y.
- void [fasp_dvec_symdiagscale](#) ([dvector](#) *b, const [dvector](#) *diag)
Symmetric diagonal scaling $D^{-1/2}b$.

9.43.1 Detailed Description

Simple vector operations – init, set, copy, etc.

Note

This file contains Level-0 (Aux) functions. It requires: [AuxThreads.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [AuxVector.c](#).

9.43.2 Function Documentation

9.43.2.1 fasp_dvec_alloc()

```
void fasp_dvec_alloc (
    const INT m,
    dvector * u )
```

Create dvector data space of REAL type.

Parameters

<i>m</i>	Number of rows
<i>u</i>	Pointer to dvector (OUTPUT)

Author

Chensong Zhang

Date

2010/04/06

Definition at line 105 of file [AuxVector.c](#).

9.43.2.2 fasp_dvec_cp()

```
void fasp_dvec_cp (
    const dvector * x,
    dvector * y )
```

Copy dvector x to dvector y.

Parameters

<i>x</i>	Pointer to dvector
<i>y</i>	Pointer to dvector (MODIFIED)

Author

Chensong Zhang

Date

11/16/2009

Definition at line 334 of file [AuxVector.c](#).

9.43.2.3 fasp_dvec_create()

```
dvector fasp_dvec_create (
    const INT m )
```

Create dvector data space of REAL type.

Parameters

<i>m</i>	Number of rows
----------	----------------

Returns

u The new dvector

Author

Chensong Zhang

Date

2010/04/06

Definition at line 62 of file [AuxVector.c](#).

9.43.2.4 fasp_dvec_free()

```
void fasp_dvec_free (  
    dvector * u )
```

Free vector data space of REAL type.

Parameters

<i>u</i>	Pointer to dvector which needs to be deallocated
----------	--

Author

Chensong Zhang

Date

2010/04/03

Definition at line 145 of file [AuxVector.c](#).

9.43.2.5 fasp_dvec_isnan()

```
SHORT fasp_dvec_isnan (  
    const dvector * u )
```

Check a dvector whether there is NAN.

Parameters

<i>u</i>	Pointer to dvector
----------	--------------------

Returns

Return TRUE if there is NAN

Author

Chensong Zhang

Date

2013/03/31

Definition at line 39 of file [AuxVector.c](#).**9.43.2.6 fasp_dvec_maxdiff()**

```
REAL fasp_dvec_maxdiff (
    const dvector * x,
    const dvector * y )
```

Maximal difference of two dvector x and y.

Parameters

<i>x</i>	Pointer to dvector
<i>y</i>	Pointer to dvector

Returns

Maximal norm of x-y

Author

Chensong Zhang

Date

11/16/2009

Modified by chunsheng Feng, Zheng Li

Date

06/30/2012

Definition at line 357 of file [AuxVector.c](#).**9.43.2.7 fasp_dvec_rand()**

```
void fasp_dvec_rand (
    const INT n,
    dvector * x )
```

Generate fake random REAL vector in the range from 0 to 1.

Parameters

<i>n</i>	Size of the vector
<i>x</i>	Pointer to dvector

Note

Sample usage:

```
dvector xapp;
```

```
fasp_dvec_create(100,&xapp);
```

```
fasp_dvec_rand(100,&xapp);
```

```
fasp_dvec_print(100,&xapp);
```

Author

Chensong Zhang

Date

11/16/2009

Definition at line 192 of file [AuxVector.c](#).

9.43.2.8 fasp_dvec_set()

```
void fasp_dvec_set (
    INT n,
    dvector * x,
    const REAL val )
```

Initialize dvector $x[i]=val$ for $i=0:n-1$.

Parameters

<i>n</i>	Number of variables
<i>x</i>	Pointer to dvector
<i>val</i>	Initial value for the vector

Author

Chensong Zhang

Date

11/16/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 222 of file [AuxVector.c](#).

9.43.2.9 fasp_dvec_symdiagscale()

```
void fasp_dvec_symdiagscale (
    dvector * b,
    const dvector * diag )
```

Symmetric diagonal scaling $D^{-1/2}b$.

Parameters

<i>b</i>	Pointer to dvector
<i>diag</i>	Pointer to dvector: the diagonal entries

Author

Xiaozhe Hu

Date

01/31/2011

Definition at line 410 of file [AuxVector.c](#).

9.43.2.10 fasp_ivec_alloc()

```
void fasp_ivec_alloc (
    const INT m,
    ivec * u )
```

Create vector data space of INT type.

Parameters

<i>m</i>	Number of rows
<i>u</i>	Pointer to ivec (OUTPUT)

Author

Chensong Zhang

Date

2010/04/06

Definition at line 125 of file [AuxVector.c](#).

9.43.2.11 fasp_ivec_create()

```
ivec fasp_ivec_create (
    const INT m )
```

Create vector data space of INT type.

Parameters

<i>m</i>	Number of rows
----------	----------------

Returns

u The new ivector

Author

Chensong Zhang

Date

2010/04/06

Definition at line 84 of file [AuxVector.c](#).

9.43.2.12 fasp_ivec_free()

```
void fasp_ivec_free (
    ivector * u )
```

Free vector data space of INT type.

Parameters

<i>u</i>	Pointer to ivector which needs to be deallocated
----------	--

Author

Chensong Zhang

Date

2010/04/03

Note

This function is same as `fasp_dvec_free` except input type.

Definition at line 164 of file [AuxVector.c](#).

9.43.2.13 fasp_ivec_set()

```
void fasp_ivec_set (
    INT n,
    ivector * u,
    const INT m )
```

Set ivector value to be *m*.

Parameters

<i>n</i>	Number of variables
<i>m</i>	Integer value of ivector
<i>u</i>	Pointer to ivector (MODIFIED)

Author

Chensong Zhang

Date

04/03/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 291 of file [AuxVector.c](#).

9.44 AuxVector.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <math.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00039 SHORT fasp_dvec_isnan (const dvector *u)
00040 {
00041     INT i;
00042
00043     for ( i = 0; i < u->row; i++ ) {
00044         if ( isnan(u->val[i]) ) return TRUE;
00045     }
00046
00047     return FALSE;
00048 }
00049
00062 dvector fasp_dvec_create (const INT m)
00063 {
00064     dvector u;
00065
00066     u.row = m;
00067     u.val = (REAL *)fasp_mem_calloc(m, sizeof(REAL));
00068
00069     return u;
00070 }
00071
00084 ivector fasp_ivec_create (const INT m)
00085 {
00086     ivector u;
00087
00088     u.row = m;
00089     u.val = (INT *)fasp_mem_calloc(m, sizeof(INT));
00090
00091     return u;
00092 }
00093
00105 void fasp_dvec_alloc (const INT  m,
00106                      dvector  *u)
00107 {
00108     u->row = m;
00109     u->val = (REAL*)fasp_mem_calloc(m, sizeof(REAL));
00110
00111     return;
00112 }
00113
00125 void fasp_ivec_alloc (const INT  m,
00126                      ivector  *u)
00127 {
00128

```

```

00129     u->row = m;
00130     u->val = (INT*) fasp_mem_calloc(m, sizeof(INT));
00131
00132     return;
00133 }
00134
00145 void fasp_dvec_free (dvector *u)
00146 {
00147     if ( u == NULL ) return;
00148     fasp_mem_free(u->val); u->val = NULL; u->row = 0;
00149 }
00150
00151
00164 void fasp_ivec_free (ivector *u)
00165 {
00166     if ( u == NULL ) return;
00167     fasp_mem_free(u->val); u->val = NULL; u->row = 0;
00168 }
00169
00170
00192 void fasp_dvec_rand (const INT  n,
00193                     dvector  *x)
00194 {
00195     const INT va = 0;
00196     const INT vb = n;
00197
00198     INT s=1, i,j;
00199
00200     srand(s);
00201     for ( i = 0; i < n; ++i ) {
00202         j = 1 + (INT) (((REAL)n)*rand()/(RAND_MAX+1.0));
00203         x->val[i] = (((REAL)j)-va)/(vb-vb);
00204     }
00205     x->row = n;
00206 }
00207
00222 void fasp_dvec_set (INT      n,
00223                    dvector  *x,
00224                    const REAL val)
00225 {
00226     INT i;
00227     REAL *xpt = x->val;
00228
00229     if ( n > 0 ) x->row = n;
00230     else n = x->row;
00231
00232 #ifdef _OPENMP
00233     // variables for OpenMP
00234     INT myid, mybegin, myend;
00235     INT nthreads = fasp_get_num_threads();
00236 #endif
00237
00238     if (val == 0.0) {
00239
00240 #ifdef _OPENMP
00241         if (n > OPENMP_HOLDS) {
00242             #pragma omp parallel for private(myid, mybegin, myend)
00243             for (myid = 0; myid < nthreads; myid++) {
00244                 fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00245                 memset(&xpt[mybegin], 0x0, sizeof(REAL)*(myend-mybegin));
00246             }
00247         }
00248         else {
00249 #endif
00250             memset(xpt, 0x0, sizeof(REAL)*n);
00251 #ifdef _OPENMP
00252         }
00253 #endif
00254     }
00255
00256     else {
00257
00258 #ifdef _OPENMP
00259         if (n > OPENMP_HOLDS) {
00260             #pragma omp parallel for private(myid, mybegin, myend)
00261             for (myid = 0; myid < nthreads; myid++) {
00262                 fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00263                 for (i=mybegin; i<myend; ++i) xpt[i]=val;
00264             }
00265         }
00266     }

```

```

00267         else {
00268 #endif
00269             for (i=0; i<n; ++i) xpt[i]=val;
00270 #ifdef _OPENMP
00271         }
00272 #endif
00273     }
00274 }
00275 }
00276
00291 void fasp_ivec_set (INT      n,
00292                    ivector  *u,
00293                    const INT m)
00294 {
00295     SHORT nthreads = 1, use_openmp = FALSE;
00296     INT i;
00297
00298     if ( n > 0 ) u->row = n;
00299     else n = u->row;
00300
00301 #ifdef _OPENMP
00302     if ( n > OPENMP_HOLDS ) {
00303         use_openmp = TRUE;
00304         nthreads = fasp_get_num_threads();
00305     }
00306 #endif
00307
00308     if (use_openmp) {
00309         INT mybegin, myend, myid;
00310 #ifdef _OPENMP
00311 #pragma omp parallel for private(myid, mybegin, myend, i)
00312 #endif
00313         for (myid = 0; myid < nthreads; myid++ ) {
00314             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00315             for (i=mybegin; i<myend; ++i) u->val[i] = m;
00316         }
00317     }
00318     else {
00319         for (i=0; i<n; ++i) u->val[i] = m;
00320     }
00321 }
00322
00334 void fasp_dvec_cp (const dvector *x,
00335                   dvector *y)
00336 {
00337     y->row = x->row;
00338     memcpy(y->val, x->val, x->row*sizeof(REAL));
00339 }
00340
00357 REAL fasp_dvec_maxdiff (const dvector *x,
00358                        const dvector *y)
00359 {
00360     const INT length = x->row;
00361     const REAL *xpt = x->val, *ypt = y->val;
00362
00363     SHORT use_openmp = FALSE;
00364     INT i;
00365     REAL Linf = 0.0, diffi = 0.0;
00366
00367 #ifdef _OPENMP
00368     INT myid, mybegin, myend, nthreads;
00369     if ( length > OPENMP_HOLDS ) {
00370         use_openmp = TRUE;
00371         nthreads = fasp_get_num_threads();
00372     }
00373 #endif
00374
00375     if (use_openmp) {
00376 #ifdef _OPENMP
00377         REAL temp = 0.;
00378 #pragma omp parallel firstprivate(temp) private(myid, mybegin, myend, i, diffi)
00379         {
00380             myid = omp_get_thread_num();
00381             fasp_get_start_end(myid, nthreads, length, &mybegin, &myend);
00382             for(i=mybegin; i<myend; i++) {
00383                 if ((diffi = ABS(xpt[i]-ypt[i])) > temp) temp = diffi;
00384             }
00385 #pragma omp critical
00386             if(temp > Linf) Linf = temp;
00387         }
00388 #endif

```



```

00389     }
00390     else {
00391         for (i=0; i<length; ++i) {
00392             if ((diffi = ABS(xpt[i]-ypt[i])) > Linf) Linf = diffi;
00393         }
00394     }
00395     return Linf;
00396 }
00397
00398 void fasp_dvec_symdiagscale (dvector *b,
00411                             const dvector *diag)
00412 {
00413     // information about dvector
00414     const INT n = b->row;
00415     REAL *val = b->val;
00416
00417     // local variables
00418     SHORT use_omp = FALSE;
00419     INT i;
00420
00421     if (diag->row != n) {
00422         printf("### ERROR: Sizes of diag = %d != dvector = %d!", diag->row, n);
00423         fasp_chkerr(ERROR_MISC, __FUNCTION__);
00424     }
00425
00426 #ifdef _OPENMP
00427     INT mybegin, myend, myid, nthreads;
00428     if (n > OPENMP_HOLDS){
00429         use_omp = TRUE;
00430         nthreads = fasp_get_num_threads();
00431     }
00432 #endif
00433
00434     if (use_omp) {
00435 #ifdef _OPENMP
00436 #pragma omp parallel for private(myid, mybegin, myend)
00437         for (myid = 0; myid < nthreads; myid++) {
00438             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00439             for (i=mybegin; i<myend; ++i) val[i] = val[i]/sqrt(diag->val[i]);
00440         }
00441 #endif
00442     }
00443     else {
00444         for (i=0; i<n; ++i) val[i] = val[i]/sqrt(diag->val[i]);
00445     }
00446     return;
00447 }
00448
00449
00450 /*-----*/
00451 /*--      End of File      --*/
00452 /*-----*/

```

9.45 BlaArray.c File Reference

BLAS1 operations for arrays.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_blas_darray_ax` (const INT n, const REAL a, REAL *x)

$$x = a * x$$
- void `fasp_blas_darray_axpy` (const INT n, const REAL a, const REAL *x, REAL *y)

$$y = a * x + y$$
- void `fasp_blas_darray_axpy_nc2` (const REAL a, const REAL *x, REAL *y)

$$y = a * x + y, \text{ length of } x \text{ and } y \text{ should be } 2$$

- void `fasp_blas_darray_axpy_nc3` (const `REAL` a, const `REAL` *x, `REAL` *y)
*y = a*x + y, length of x and y should be 3*
- void `fasp_blas_darray_axpy_nc5` (const `REAL` a, const `REAL` *x, `REAL` *y)
*y = a*x + y, length of x and y should be 5*
- void `fasp_blas_darray_axpy_nc7` (const `REAL` a, const `REAL` *x, `REAL` *y)
*y = a*x + y, length of x and y should be 7*
- void `fasp_blas_darray_axpyz` (const `INT` n, const `REAL` a, const `REAL` *x, const `REAL` *y, `REAL` *z)
*z = a*x + y*
- void `fasp_blas_darray_axpyz_nc2` (const `REAL` a, const `REAL` *x, const `REAL` *y, `REAL` *z)
*z = a*x + y, length of x, y and z should be 2*
- void `fasp_blas_darray_axpyz_nc3` (const `REAL` a, const `REAL` *x, const `REAL` *y, `REAL` *z)
*z = a*x + y, length of x, y and z should be 3*
- void `fasp_blas_darray_axpyz_nc5` (const `REAL` a, const `REAL` *x, const `REAL` *y, `REAL` *z)
*z = a*x + y, length of x, y and z should be 5*
- void `fasp_blas_darray_axpyz_nc7` (const `REAL` a, const `REAL` *x, const `REAL` *y, `REAL` *z)
*z = a*x + y, length of x, y and z should be 7*
- void `fasp_blas_darray_axpby` (const `INT` n, const `REAL` a, const `REAL` *x, const `REAL` b, `REAL` *y)
*y = a*x + b*y*
- `REAL fasp_blas_darray_norm1` (const `INT` n, const `REAL` *x)
L1 norm of array x.
- `REAL fasp_blas_darray_norm2` (const `INT` n, const `REAL` *x)
L2 norm of array x.
- `REAL fasp_blas_darray_norminf` (const `INT` n, const `REAL` *x)
Linf norm of array x.
- `REAL fasp_blas_darray_dotprod` (const `INT` n, const `REAL` *x, const `REAL` *y)
Inner product of two arrays x and y.

9.45.1 Detailed Description

BLAS1 operations for arrays.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxThreads.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaArray.c](#).

9.45.2 Function Documentation

9.45.2.1 `fasp_blas_darray_ax()`

```
void fasp_blas_darray_ax (
    const INT n,
    const REAL a,
    REAL * x )
```

`x = a*x`

Parameters

<i>n</i>	Number of variables
<i>a</i>	Factor a
<i>x</i>	Pointer to x

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Warning

x is reused to store the resulting array!

Definition at line 43 of file [BlaArray.c](#).

9.45.2.2 fasp_blas_darray_axpby()

```
void fasp_blas_darray_axpby (
    const INT n,
    const REAL a,
    const REAL * x,
    const REAL b,
    REAL * y )
```

$y = a*x + b*y$

Parameters

<i>n</i>	Number of variables
<i>a</i>	Factor a
<i>x</i>	Pointer to x
<i>b</i>	Factor b
<i>y</i>	Pointer to y, reused to store the resulting array

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 580 of file [BlaArray.c](#).

9.45.2.3 fasp_blas_darray_axpy()

```
void fasp_blas_darray_axpy (
```

```

const INT n,
const REAL a,
const REAL * x,
REAL * y )

```

$y = a*x + y$

Parameters

n	Number of variables
a	Factor a
x	Pointer to x
y	Pointer to y, reused to store the resulting array

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 93 of file [BlaArray.c](#).

9.45.2.4 fasp_blas_darray_axpy_nc2()

```

void fasp_blas_darray_axpy_nc2 (
    const REAL a,
    const REAL * x,
    REAL * y )

```

$y = a*x + y$, length of x and y should be 2

Parameters

a	REAL factor a
x	Pointer to the original array
y	Pointer to the destination array

Author

Xiaozhe Hu

Date

18/11/2011

Definition at line 170 of file [BlaArray.c](#).

9.45.2.5 fasp_blas_darray_axpy_nc3()

```

void fasp_blas_darray_axpy_nc3 (
    const REAL a,

```

```
const REAL * x,  
REAL * y )
```

y = a*x + y, length of x and y should be 3

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array
<i>y</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 193 of file [BlaArray.c](#).

9.45.2.6 fasp_blas_darray_axpy_nc5()

```
void fasp_blas_darray_axpy_nc5 (  
    const REAL a,  
    const REAL * x,  
    REAL * y )
```

y = a*x + y, length of x and y should be 5

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array
<i>y</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 222 of file [BlaArray.c](#).

9.45.2.7 fasp_blas_darray_axpy_nc7()

```
void fasp_blas_darray_axpy_nc7 (  
    const REAL a,  
    const REAL * x,  
    REAL * y )
```

y = a*x + y, length of x and y should be 7

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array
<i>y</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 269 of file [BlaArray.c](#).

9.45.2.8 fasp_blas_darray_axpyz()

```
void fasp_blas_darray_axpyz (
    const INT n,
    const REAL a,
    const REAL * x,
    const REAL * y,
    REAL * z )
```

$z = a*x + y$

Parameters

<i>n</i>	Number of variables
<i>a</i>	Factor a
<i>x</i>	Pointer to x
<i>y</i>	Pointer to y
<i>z</i>	Pointer to z

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 347 of file [BlaArray.c](#).

9.45.2.9 fasp_blas_darray_axpyz_nc2()

```
void fasp_blas_darray_axpyz_nc2 (
    const REAL a,
    const REAL * x,
    const REAL * y,
    REAL * z )
```

$z = a*x + y$, length of x, y and z should be 2

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array 1
<i>y</i>	Pointer to the original array 2
<i>z</i>	Pointer to the destination array

Author

Xiaozhe Hu

Date

18/11/2011

Definition at line 393 of file [BlaArray.c](#).**9.45.2.10 fasp_blas_darray_axpyz_nc3()**

```
void fasp_blas_darray_axpyz_nc3 (  
    const REAL a,  
    const REAL * x,  
    const REAL * y,  
    REAL * z )
```

$z = a*x + y$, length of x , y and z should be 3

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array 1
<i>y</i>	Pointer to the original array 2
<i>z</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 419 of file [BlaArray.c](#).**9.45.2.11 fasp_blas_darray_axpyz_nc5()**

```
void fasp_blas_darray_axpyz_nc5 (  
    const REAL a,  
    const REAL * x,  
    const REAL * y,  
    REAL * z )
```

$z = a*x + y$, length of x , y and z should be 5

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array 1
<i>y</i>	Pointer to the original array 2
<i>z</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 451 of file [BlaArray.c](#).

9.45.2.12 fasp_blas_darray_axpyz_nc7()

```
void fasp_blas_darray_axpyz_nc7 (
    const REAL a,
    const REAL * x,
    const REAL * y,
    REAL * z )
```

$z = a*x + y$, length of x , y and z should be 7

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to the original array 1
<i>y</i>	Pointer to the original array 2
<i>z</i>	Pointer to the destination array

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 501 of file [BlaArray.c](#).

9.45.2.13 fasp_blas_darray_dotprod()

```
REAL fasp_blas_darray_dotprod (
    const INT n,
    const REAL * x,
    const REAL * y )
```

Inner product of two arraies x and y .

Parameters

n	Number of variables
x	Pointer to x
y	Pointer to y

Returns

Inner product (x,y)

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 741 of file [BlaArray.c](#).

9.45.2.14 fasp_blas_darray_norm1()

```
REAL fasp_blas_darray_norm1 (  
    const INT n,  
    const REAL * x )
```

L1 norm of array x.

Parameters

n	Number of variables
x	Pointer to x

Returns

L1 norm of x

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 628 of file [BlaArray.c](#).

9.45.2.15 fasp_blas_darray_norm2()

```
REAL fasp_blas_darray_norm2 (  
    const INT n,  
    const REAL * x )
```

L2 norm of array x.

Parameters

n	Number of variables
x	Pointer to x

Returns

L2 norm of x

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 657 of file [BlaArray.c](#).

9.45.2.16 fasp_blas_darray_norminf()

```
REAL fasp_blas_darray_norminf (
    const INT n,
    const REAL * x )
```

Linf norm of array x.

Parameters

n	Number of variables
x	Pointer to x

Returns

L_inf norm of x

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Zheng Li on 06/28/2012
Definition at line 686 of file [BlaArray.c](#).

9.46 BlaArray.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <math.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
```

```

00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_functs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00043 void fasp_blas_darray_ax (const INT    n,
00044                          const REAL  a,
00045                          REAL        *x)
00046 {
00047     if ( a == 1.0 ) return; // do nothing
00048
00049     {
00050         SHORT use_omp = FALSE;
00051         INT    i;
00052
00053 #ifdef _OPENMP
00054         INT myid, mybegin, myend, nthreads;
00055         if ( n > OPENMP_HOLDS ) {
00056             use_omp = TRUE;
00057             nthreads = fasp_get_num_threads();
00058         }
00059 #endif
00060
00061         if ( use_omp ) {
00062 #ifdef _OPENMP
00063             #pragma omp parallel private(myid, mybegin, myend, i)
00064             {
00065                 myid = omp_get_thread_num();
00066                 fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00067                 for ( i = mybegin; i < myend; ++i ) x[i] *= a;
00068             }
00069 #endif
00070         }
00071         else {
00072             for ( i = 0; i < n; ++i ) x[i] *= a;
00073         }
00074     }
00075 }
00076
00093 void fasp_blas_darray_axpy (const INT    n,
00094                           const REAL  a,
00095                           const REAL  *x,
00096                           REAL        *y)
00097 {
00098     SHORT use_omp = FALSE;
00099     INT    i;
00100
00101 #ifdef _OPENMP
00102     INT myid, mybegin, myend, nthreads;
00103     if ( n > OPENMP_HOLDS ) {
00104         use_omp = TRUE;
00105         nthreads = fasp_get_num_threads();
00106     }
00107 #endif
00108
00109     if ( a == 1.0 ) {
00110         if ( use_omp ) {
00111 #ifdef _OPENMP
00112             #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00113             {
00114                 myid = omp_get_thread_num();
00115                 fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00116                 for ( i = mybegin; i < myend; ++i ) y[i] += x[i];
00117             }
00118 #endif
00119         }
00120         else {
00121             for ( i = 0; i < n; ++i ) y[i] += x[i];
00122         }
00123     }
00124
00125     else if ( a == -1.0 ) {
00126         if ( use_omp ) {
00127 #ifdef _OPENMP
00128             #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00129             {
00130                 myid = omp_get_thread_num();

```

```

00131         fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00132         for ( i = mybegin; i < myend; ++i ) y[i] -= x[i];
00133     }
00134 #endif
00135 }
00136 else {
00137     for ( i = 0; i < n; ++i ) y[i] -= x[i];
00138 }
00139 }
00140
00141 else {
00142     if ( use_omp ) {
00143 #ifdef _OPENMP
00144 #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00145     {
00146         myid = omp_get_thread_num();
00147         fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00148         for ( i = mybegin; i < myend; ++i ) y[i] += a*x[i];
00149     }
00150 #endif
00151     }
00152     else {
00153         for ( i = 0; i < n; ++i ) y[i] += a*x[i];
00154     }
00155 }
00156 }
00157
00170 void fasp_blas_darray_axpy_nc2 (const REAL  a,
00171                                const REAL  *x,
00172                                REAL        *y)
00173 {
00174     y[0] += a*x[0];
00175     y[1] += a*x[1];
00176
00177     y[2] += a*x[2];
00178     y[3] += a*x[3];
00179 }
00180
00193 void fasp_blas_darray_axpy_nc3 (const REAL  a,
00194                                const REAL  *x,
00195                                REAL        *y)
00196 {
00197     y[0] += a*x[0];
00198     y[1] += a*x[1];
00199     y[2] += a*x[2];
00200
00201     y[3] += a*x[3];
00202     y[4] += a*x[4];
00203     y[5] += a*x[5];
00204
00205     y[6] += a*x[6];
00206     y[7] += a*x[7];
00207     y[8] += a*x[8];
00208 }
00209
00222 void fasp_blas_darray_axpy_nc5 (const REAL  a,
00223                                const REAL  *x,
00224                                REAL        *y)
00225 {
00226     y[0] += a*x[0];
00227     y[1] += a*x[1];
00228     y[2] += a*x[2];
00229     y[3] += a*x[3];
00230     y[4] += a*x[4];
00231
00232     y[5] += a*x[5];
00233     y[6] += a*x[6];
00234     y[7] += a*x[7];
00235     y[8] += a*x[8];
00236     y[9] += a*x[9];
00237
00238     y[10] += a*x[10];
00239     y[11] += a*x[11];
00240     y[12] += a*x[12];
00241     y[13] += a*x[13];
00242     y[14] += a*x[14];
00243
00244     y[15] += a*x[15];
00245     y[16] += a*x[16];
00246     y[17] += a*x[17];
00247     y[18] += a*x[18];

```

```

00248     y[19] += a*x[19];
00249
00250     y[20] += a*x[20];
00251     y[21] += a*x[21];
00252     y[22] += a*x[22];
00253     y[23] += a*x[23];
00254     y[24] += a*x[24];
00255 }
00256
00269 void fasp_blas_darray_axpy_nc7 (const REAL    a,
00270                                const REAL    *x,
00271                                REAL          *y)
00272 {
00273     y[0] += a*x[0];
00274     y[1] += a*x[1];
00275     y[2] += a*x[2];
00276     y[3] += a*x[3];
00277     y[4] += a*x[4];
00278     y[5] += a*x[5];
00279     y[6] += a*x[6];
00280
00281     y[7] += a*x[7];
00282     y[8] += a*x[8];
00283     y[9] += a*x[9];
00284     y[10] += a*x[10];
00285     y[11] += a*x[11];
00286     y[12] += a*x[12];
00287     y[13] += a*x[13];
00288
00289     y[14] += a*x[14];
00290     y[15] += a*x[15];
00291     y[16] += a*x[16];
00292     y[17] += a*x[17];
00293     y[18] += a*x[18];
00294     y[19] += a*x[19];
00295     y[20] += a*x[20];
00296
00297     y[21] += a*x[21];
00298     y[22] += a*x[22];
00299     y[23] += a*x[23];
00300     y[24] += a*x[24];
00301     y[25] += a*x[25];
00302     y[26] += a*x[26];
00303     y[27] += a*x[27];
00304
00305     y[28] += a*x[28];
00306     y[29] += a*x[29];
00307     y[30] += a*x[30];
00308     y[31] += a*x[31];
00309     y[32] += a*x[32];
00310     y[33] += a*x[33];
00311     y[34] += a*x[34];
00312
00313     y[35] += a*x[35];
00314     y[36] += a*x[36];
00315     y[37] += a*x[37];
00316     y[38] += a*x[38];
00317     y[39] += a*x[39];
00318     y[40] += a*x[40];
00319     y[41] += a*x[41];
00320
00321     y[42] += a*x[42];
00322     y[43] += a*x[43];
00323     y[44] += a*x[44];
00324     y[45] += a*x[45];
00325     y[46] += a*x[46];
00326     y[47] += a*x[47];
00327     y[48] += a*x[48];
00328 }
00329
00347 void fasp_blas_darray_axpyz (const INT    n,
00348                              const REAL    a,
00349                              const REAL    *x,
00350                              const REAL    *y,
00351                              REAL          *z)
00352 {
00353     SHORT use_omp = FALSE;
00354     INT    i;
00355
00356 #ifdef _OPENMP
00357     INT myid, mybegin, myend, nthreads;

```

```

00358     if ( n > OPENMP_HOLDS ) {
00359         use_omp = TRUE;
00360         nthreads = fasp_get_num_threads();
00361     }
00362 #endif
00363
00364     if ( use_omp ) {
00365 #ifdef _OPENMP
00366 #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00367 {
00368     myid = omp_get_thread_num();
00369     fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00370     for ( i = mybegin; i < myend; ++i ) z[i] = a*x[i] + y[i];
00371 }
00372 #endif
00373     }
00374     else {
00375         for ( i = 0; i < n; ++i ) z[i] = a*x[i] + y[i];
00376     }
00377 }
00378
00393 void fasp_blas_darray_axpyz_nc2 (const REAL a,
00394                                  const REAL *x,
00395                                  const REAL *y,
00396                                  REAL *z)
00397 {
00398     z[0] = a*x[0] + y[0];
00399     z[1] = a*x[1] + y[1];
00400
00401     z[2] = a*x[2] + y[2];
00402     z[3] = a*x[3] + y[3];
00403 }
00404
00419 void fasp_blas_darray_axpyz_nc3 (const REAL a,
00420                                  const REAL *x,
00421                                  const REAL *y,
00422                                  REAL *z)
00423 {
00424     z[0] = a*x[0] + y[0];
00425     z[1] = a*x[1] + y[1];
00426     z[2] = a*x[2] + y[2];
00427
00428     z[3] = a*x[3] + y[3];
00429     z[4] = a*x[4] + y[4];
00430     z[5] = a*x[5] + y[5];
00431
00432     z[6] = a*x[6] + y[6];
00433     z[7] = a*x[7] + y[7];
00434     z[8] = a*x[8] + y[8];
00435 }
00436
00451 void fasp_blas_darray_axpyz_nc5 (const REAL a,
00452                                  const REAL *x,
00453                                  const REAL *y,
00454                                  REAL *z)
00455 {
00456     z[0] = a*x[0] + y[0];
00457     z[1] = a*x[1] + y[1];
00458     z[2] = a*x[2] + y[2];
00459     z[3] = a*x[3] + y[3];
00460     z[4] = a*x[4] + y[4];
00461
00462     z[5] = a*x[5] + y[5];
00463     z[6] = a*x[6] + y[6];
00464     z[7] = a*x[7] + y[7];
00465     z[8] = a*x[8] + y[8];
00466     z[9] = a*x[9] + y[9];
00467
00468     z[10] = a*x[10] + y[10];
00469     z[11] = a*x[11] + y[11];
00470     z[12] = a*x[12] + y[12];
00471     z[13] = a*x[13] + y[13];
00472     z[14] = a*x[14] + y[14];
00473
00474     z[15] = a*x[15] + y[15];
00475     z[16] = a*x[16] + y[16];
00476     z[17] = a*x[17] + y[17];
00477     z[18] = a*x[18] + y[18];
00478     z[19] = a*x[19] + y[19];
00479
00480     z[20] = a*x[20] + y[20];

```

```

00481     z[21] = a*x[21] + y[21];
00482     z[22] = a*x[22] + y[22];
00483     z[23] = a*x[23] + y[23];
00484     z[24] = a*x[24] + y[24];
00485 }
00486
00501 void fasp_blas_darray_axpyz_nc7 (const REAL  a,
00502                                  const REAL  *x,
00503                                  const REAL  *y,
00504                                  REAL        *z)
00505 {
00506     z[0] = a*x[0] + y[0];
00507     z[1] = a*x[1] + y[1];
00508     z[2] = a*x[2] + y[2];
00509     z[3] = a*x[3] + y[3];
00510     z[4] = a*x[4] + y[4];
00511     z[5] = a*x[5] + y[5];
00512     z[6] = a*x[6] + y[6];
00513
00514     z[7] = a*x[7] + y[7];
00515     z[8] = a*x[8] + y[8];
00516     z[9] = a*x[9] + y[9];
00517     z[10] = a*x[10] + y[10];
00518     z[11] = a*x[11] + y[11];
00519     z[12] = a*x[12] + y[12];
00520     z[13] = a*x[13] + y[13];
00521
00522     z[14] = a*x[14] + y[14];
00523     z[15] = a*x[15] + y[15];
00524     z[16] = a*x[16] + y[16];
00525     z[17] = a*x[17] + y[17];
00526     z[18] = a*x[18] + y[18];
00527     z[19] = a*x[19] + y[19];
00528     z[20] = a*x[20] + y[20];
00529
00530     z[21] = a*x[21] + y[21];
00531     z[22] = a*x[22] + y[22];
00532     z[23] = a*x[23] + y[23];
00533     z[24] = a*x[24] + y[24];
00534     z[25] = a*x[25] + y[25];
00535     z[26] = a*x[26] + y[26];
00536     z[27] = a*x[27] + y[27];
00537
00538     z[28] = a*x[28] + y[28];
00539     z[29] = a*x[29] + y[29];
00540     z[30] = a*x[30] + y[30];
00541     z[31] = a*x[31] + y[31];
00542     z[32] = a*x[32] + y[32];
00543     z[33] = a*x[33] + y[33];
00544     z[34] = a*x[34] + y[34];
00545
00546     z[35] = a*x[35] + y[35];
00547     z[36] = a*x[36] + y[36];
00548     z[37] = a*x[37] + y[37];
00549     z[38] = a*x[38] + y[38];
00550     z[39] = a*x[39] + y[39];
00551     z[40] = a*x[40] + y[40];
00552     z[41] = a*x[41] + y[41];
00553
00554     z[42] = a*x[42] + y[42];
00555     z[43] = a*x[43] + y[43];
00556     z[44] = a*x[44] + y[44];
00557     z[45] = a*x[45] + y[45];
00558     z[46] = a*x[46] + y[46];
00559     z[47] = a*x[47] + y[47];
00560     z[48] = a*x[48] + y[48];
00561 }
00562
00580 void fasp_blas_darray_axpby (const INT  n,
00581                              const REAL  a,
00582                              const REAL  *x,
00583                              const REAL  b,
00584                              REAL        *y)
00585 {
00586     SHORT use_omp = FALSE;
00587     INT    i;
00588
00589     #ifdef _OPENMP
00590         INT myid, mybegin, myend, nthreads;
00591         if ( n > OPENMP_HOLDS ) {
00592             use_omp = TRUE;

```

```

00593         nthreads = fasp_get_num_threads();
00594     }
00595 #endif
00596
00597     if (use_openmp) {
00598 #ifdef _OPENMP
00599 #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00600     {
00601         myid = omp_get_thread_num();
00602         fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00603         for ( i = mybegin; i < myend; ++i ) y[i] = a*x[i] + b*y[i];
00604     }
00605 #endif
00606     }
00607     else {
00608         for ( i = 0; i < n; ++i ) y[i] = a*x[i] + b*y[i];
00609     }
00610 }
00611 }
00612
00628 REAL fasp_blas_darray_norm1 (const INT    n,
00629                             const REAL  *x)
00630 {
00631     register REAL onenorm = 0.0;
00632     INT i;
00633
00634 #ifdef _OPENMP
00635 #pragma omp parallel for reduction(+:onenorm) private(i)
00636 #endif
00637     for ( i = 0; i < n; ++i ) onenorm += ABS(x[i]);
00638
00639     return onenorm;
00640 }
00641
00657 REAL fasp_blas_darray_norm2 (const INT    n,
00658                             const REAL  *x)
00659 {
00660     register REAL twonorm = 0.0;
00661     INT i;
00662
00663 #ifdef _OPENMP
00664 #pragma omp parallel for reduction(+:twonorm) private(i)
00665 #endif
00666     for ( i = 0; i < n; ++i ) twonorm += x[i] * x[i];
00667
00668     return sqrt(twonorm);
00669 }
00670
00686 REAL fasp_blas_darray_norminf (const INT    n,
00687                               const REAL  *x)
00688 {
00689     SHORT use_openmp = FALSE;
00690     register REAL infnorm = 0.0;
00691     INT i;
00692
00693 #ifdef _OPENMP
00694     INT myid, mybegin, myend, nthreads;
00695     if ( n > OPENMP_HOLDS ) {
00696         use_openmp = TRUE;
00697         nthreads = fasp_get_num_threads();
00698     }
00699 #endif
00700
00701     if ( use_openmp ) {
00702 #ifdef _OPENMP
00703         REAL infnorm_loc = 0.0;
00704 #pragma omp parallel firstprivate(infnorm_loc) private(myid, mybegin, myend, i)
00705         {
00706             myid = omp_get_thread_num();
00707             fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
00708             for ( i = mybegin; i < myend; ++i )
00709                 infnorm_loc = MAX( infnorm_loc, ABS(x[i]) );
00710
00711             if ( infnorm_loc > infnorm ) {
00712 #pragma omp critical
00713                 infnorm = MAX( infnorm_loc, infnorm );
00714             }
00715         }
00716 #endif
00717     }
00718     else {

```



```

00719         for ( i = 0; i < n; ++i ) infnorm = MAX( infnorm, ABS(x[i]) );
00720     }
00721
00722     return infnorm;
00723 }
00724
00741 REAL fasp_blas_darray_dotprod (const INT    n,
00742                               const REAL  *x,
00743                               const REAL  *y)
00744 {
00745     SHORT use_omp = FALSE;
00746     register REAL value = 0.0;
00747     INT i;
00748
00749 #ifdef _OPENMP
00750     if ( n > OPENMP_HOLDS ) use_omp = TRUE;
00751 #endif
00752
00753     if ( use_omp ) {
00754 #ifdef _OPENMP
00755 #pragma omp parallel for reduction(+:value) private(i)
00756 #endif
00757         for ( i = 0; i < n; ++i ) value += x[i]*y[i];
00758     }
00759     else {
00760         for ( i = 0; i < n; ++i ) value += x[i]*y[i];
00761     }
00762
00763     return value;
00764 }
00765
00766 /*-----*/
00767 /*--      End of File      --*/
00768 /*-----*/

```

9.47 BlaEigen.c File Reference

Computing the extreme eigenvalues.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [REAL fasp_dcsr_max eig](#) (const [dCSRmat](#) *A, const [REAL](#) tol, const [INT](#) maxit)
Approximate the largest eigenvalue of A by the power method.

9.47.1 Detailed Description

Computing the extreme eigenvalues.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvCSR.c](#), and [BlaVector.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaEigen.c](#).

9.47.2 Function Documentation

9.47.2.1 fasp_dcsr_maxeig()

```
REAL fasp_dcsr_maxeig (
    const dCSRmat * A,
    const REAL tol,
    const INT maxit )
```

Approximate the largest eigenvalue of A by the power method.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>tol</i>	Tolerance for stopping the power method
<i>maxit</i>	Max number of iterations

Returns

Largest eigenvalue

Author

Xiaozhe Hu

Date

01/25/2011

Definition at line 37 of file [BlaEigen.c](#).

9.48 BlaEigen.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <math.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_funcs.h"
00018
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00037 REAL fasp_dcsr_maxeig (const dCSRmat *A,
00038                      const REAL    tol,
00039                      const INT     maxit)
00040 {
00041     REAL eigenvalue = 0.0, temp = 1.0, L2_norm_y;
00042     dvector x, y;
00043     int i;
00044
00045     fasp_dvec_alloc(A->row, &x);
00046     fasp_dvec_rand(A->row, &x);
00047     fasp_blas_darray_ax(A->row, 1.0/fasp_blas_dvec_norm2(&x), x.val);
00048
00049     fasp_dvec_alloc(A->row, &y);
00050
00051     for ( i = maxit; i--; ) {
00052         // y = Ax;
00053         fasp_blas_dcsr_mmv(A, x.val, y.val);
00054
00055         // y/||y||
00056         L2_norm_y = fasp_blas_dvec_norm2(&y);
00057         fasp_blas_darray_ax(A->row, 1.0/L2_norm_y, y.val);
00058
00059         // eigenvalue = y'Ay;
00060         eigenvalue = fasp_blas_dcsr_vmv(A, y.val, y.val);
00061     }
```

```

00062         // convergence test
00063         if ( (ABS(eigenvalue - temp)/ABS(temp)) < tol ) break;
00064
00065         fasp_dvec_cp(&y, &x);
00066         temp = eigenvalue;
00067     }
00068
00069     // clean up memory
00070     fasp_dvec_free(&x);
00071     fasp_dvec_free(&y);
00072
00073     return eigenvalue;
00074 }
00075
00076 /*-----*/
00077 /*--      End of File      --*/
00078 /*-----*/

```

9.49 BlaFormat.c File Reference

Subroutines for matrix format conversion.

```

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_format_dcoo_dcsr](#) (const [dCOOmat](#) *A, [dCSRmat](#) *B)
Transform a REAL matrix from its IJ format to its CSR format.
- [SHORT fasp_format_dcsr_dcoo](#) (const [dCSRmat](#) *A, [dCOOmat](#) *B)
Transform a REAL matrix from its CSR format to its IJ format.
- [SHORT fasp_format_dstr_dcsr](#) (const [dSTRmat](#) *A, [dCSRmat](#) *B)
Transfer a 'dSTRmat' type matrix into a 'dCSRmat' type matrix.
- [dCSRmat fasp_format_dblc_dcsr](#) (const [dBLCmat](#) *Ab)
Form the whole dCSRmat A using blocks given in Ab.
- [dCSRmat fasp_format_dcsr_dcsr](#) (const [dCSRmat](#) *A)
Convert a dCSRmat into a dCSRmat.
- [dCSRmat fasp_format_dbsr_dcsr](#) (const [dBSRmat](#) *B)
Transfer a 'dBSRmat' type matrix into a dCSRmat.
- [dBSRmat fasp_format_dcsr_dbsr](#) (const [dCSRmat](#) *A, const [INT](#) nb)
Transfer a dCSRmat type matrix into a dBSRmat.
- [dBSRmat fasp_format_dstr_dbsr](#) (const [dSTRmat](#) *B)
Transfer a 'dSTRmat' type matrix to a 'dBSRmat' type matrix.
- [dCOOmat * fasp_format_dbsr_dcoo](#) (const [dBSRmat](#) *B)
Transfer a 'dBSRmat' type matrix to a 'dCOOmat' type matrix.

9.49.1 Detailed Description

Subroutines for matrix format conversion.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), and [BlaSparseCSRL.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaFormat.c](#).

9.49.2 Function Documentation

9.49.2.1 fasp_format_dblc_dcsr()

```
dCSRmat fasp_format_dblc_dcsr (
    const dBLCmat * Ab )
```

Form the whole [dCSRmat](#) A using blocks given in Ab.

Parameters

<i>Ab</i>	Pointer to dBLCmat matrix
-----------	---

Returns

[dCSRmat](#) matrix if succeed, NULL if fail

Author

Shiquan Zhang

Date

08/10/2010

Definition at line [294](#) of file [BlaFormat.c](#).

9.49.2.2 fasp_format_dbsr_dcoo()

```
dCOOmat * fasp_format_dbsr_dcoo (
    const dBSRmat * B )
```

Transfer a '[dBSRmat](#)' type matrix to a '[dCOOmat](#)' type matrix.

Parameters

<i>B</i>	Pointer to dBSRmat matrix
----------	---

Returns

Pointer to [dCOOmat](#) matrix

Author

Zhiyang Zhou

Date

2010/10/26

Definition at line [948](#) of file [BlaFormat.c](#).

9.49.2.3 fasp_format_dbsr_dcsr()

```
dCSRmat fasp_format_dbsr_dcsr (
    const dBSRmat * B )
```

Transfer a 'dBSRmat' type matrix into a dCSRmat.

Parameters

<i>B</i>	Pointer to dBSRmat matrix
----------	---------------------------

Returns

dCSRmat matrix

Author

Zhiyang Zhou

Date

10/23/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/24/2012

Note

Works for general nb (Xiaozhe)

Definition at line 497 of file BlaFormat.c.

9.49.2.4 fasp_format_dcoo_dcsr()

```
SHORT fasp_format_dcoo_dcsr (
    const dCOOmat * A,
    dCSRmat * B )
```

Transform a REAL matrix from its IJ format to its CSR format.

Parameters

<i>A</i>	Pointer to dCOOmat matrix
<i>B</i>	Pointer to dCSRmat matrix

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xuehai Huang

Date

08/10/2009

Definition at line 36 of file BlaFormat.c.

9.49.2.5 fasp_format_dcsr_dbsr()

```
dBSRmat fasp_format_dcsr_dbsr (
    const dCSRmat * A,
    const INT nb )
```

Transfer a [dCSRmat](#) type matrix into a [dBSRmat](#).

Parameters

<i>A</i>	Pointer to the dCSRmat type matrix
<i>nb</i>	size of each block

Returns

[dBSRmat](#) matrix

Author

Zheng Li

Date

03/27/2014

Note

modified by Xiaozhe Hu to avoid potential memory leakage problem

Definition at line [723](#) of file [BlaFormat.c](#).

9.49.2.6 fasp_format_dcsr_dcoo()

```
SHORT fasp_format_dcsr_dcoo (
    const dCSRmat * A,
    dCOOmat * B )
```

Transform a REAL matrix from its CSR format to its IJ format.

Parameters

<i>A</i>	Pointer to dCSRmat matrix
<i>B</i>	Pointer to dCOOmat matrix

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xuehai Huang

Date

08/10/2009

Modified by Chunsheng Feng, Zheng Li on 10/12/2012

Definition at line [83](#) of file [BlaFormat.c](#).

9.49.2.7 fasp_format_dcsr_dcsr()

```
dCSRmat * fasp_format_dcsr_dcsr (
    const dCSRmat * A )
```

Convert a [dCSRmat](#) into a [dCSRmat](#).

Parameters

<i>A</i>	Pointer to dCSRmat matrix
----------	---

Returns

Pointer to [dCSRmat](#) matrix

Author

Zhiyang Zhou

Date

2011/01/07

Definition at line [363](#) of file [BlaFormat.c](#).

9.49.2.8 fasp_format_dstr_dbsr()

```
dBSRmat fasp_format_dstr_dbsr (
    const dSTRmat * B )
```

Transfer a '[dSTRmat](#)' type matrix to a '[dBSRmat](#)' type matrix.

Parameters

<i>B</i>	Pointer to dSTRmat matrix
----------	---

Returns

[dBSRmat](#) matrix

Author

Zhiyang Zhou

Date

2010/10/26

Definition at line [844](#) of file [BlaFormat.c](#).

9.49.2.9 fasp_format_dstr_dcsr()

```
SHORT fasp_format_dstr_dcsr (
    const dSTRmat * A,
    dCSRmat * B )
```

Transfer a '[dSTRmat](#)' type matrix into a '[dCSRmat](#)' type matrix.

Parameters

<i>A</i>	Pointer to dSTRmat matrix
<i>B</i>	Pointer to dCSRmat matrix

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Zhiyang Zhou

Date

2010/04/29

Definition at line 119 of file [BlaFormat.c](#).

9.50 BlaFormat.c

[Go to the documentation of this file.](#)

```

00001
00015 #include "fasp.h"
00016 #include "fasp_block.h"
00017 #include "fasp_funcs.h"
00018
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00036 SHORT fasp_format_dcoo_dcsr (const dCOOmat *A,
00037                             dCSRmat *B)
00038 {
00039     const INT m=A->row, n=A->col, nnz=A->nnz;
00040     INT iind, jind, i;
00041
00042     fasp_dcsr_alloc(m,n,nnz,B);
00043     INT *ia = B->IA;
00044
00045     INT *ind = (INT *) fasp_mem_calloc(m+1,sizeof(INT));
00046     memset(ind, 0, sizeof(INT)*(m+1)); // initialize ind
00047     for ( i=0; i<nnz; ++i ) ind[A->rowind[i]+1]++; // count nnz in each row
00048
00049     ia[0] = 0; // first index starting from zero
00050     for ( i=1; i<=m; ++i ) {
00051         ia[i] = ia[i-1]+ind[i]; // set row_idx
00052         ind[i] = ia[i];
00053     }
00054
00055     // loop over nnz and set col_idx and val
00056     for ( i=0; i<nnz; ++i ) {
00057         iind = A->rowind[i]; jind = ind[iind];
00058         B->JA [jind] = A->colind[i];
00059         B->val[jind] = A->val[i];
00060         ind[iind]   = ++jind;
00061     }
00062
00063     fasp_mem_free(ind); ind = NULL;
00064
00065     return FASP_SUCCESS;
00066 }
00067
00083 SHORT fasp_format_dcsr_dcoo (const dCSRmat *A,
00084                             dCOOmat *B)
00085 {
00086     const INT m=A->row, nnz=A->nnz;
00087     INT i, j;
00088
00089     B->rowind = (INT *) fasp_mem_calloc(nnz,sizeof(INT));

```



```

00090     B->colind = (INT *)fasp_mem_calloc(nnz,sizeof(INT));
00091     B->val     = (REAL *)fasp_mem_calloc(nnz,sizeof(REAL));
00092
00093 #ifdef _OPENMP
00094 #pragma omp parallel for if(m>OPENMP_HOLDS) private(i, j)
00095 #endif
00096     for (i=0;i<m;++i) {
00097         for (j=A->IA[i];j<A->IA[i+1];++j) B->rowind[j]=i;
00098     }
00099
00100     memcpy(B->colind, A->JA, nnz*sizeof(INT));
00101     memcpy(B->val, A->val, nnz*sizeof(REAL));
00102
00103     return FASP_SUCCESS;
00104 }
00105
00119 SHORT fasp_format_dstr_dcsr (const dSTRmat *A,
00120                             dCSRmat *B)
00121 {
00122     // some members of A
00123     const INT nc = A->nc;
00124     const INT ngrid = A->ngrid;
00125     const INT nband = A->nband;
00126     const INT *offsets = A->offsets;
00127
00128     REAL *diag = A->diag;
00129     REAL **offdiag = A->offdiag;
00130
00131     // some members of B
00132     const INT glo_row = nc*ngrid;
00133     INT glo_nnz;
00134     INT *ia = NULL;
00135     INT *ja = NULL;
00136     REAL *a = NULL;
00137
00138     dCSRmat B_tmp;
00139
00140     // local variables
00141     INT width;
00142     INT nc2 = nc*nc;
00143     INT BAND,ROW,COL;
00144     INT ncb,nci;
00145     INT row_start,col_start;
00146     INT block; // how many blocks in the current ROW
00147     INT i,j;
00148     INT pos;
00149     INT start;
00150     INT val_L_start,val_R_start;
00151     INT row;
00152     INT tmp_col;
00153     REAL tmp_val;
00154
00155     // allocate for 'ia' array
00156     ia = (INT *)fasp_mem_calloc(glo_row+1,sizeof(INT));
00157
00158     // Generate the 'ia' array
00159     ia[0] = 0;
00160     for (ROW = 0; ROW < ngrid; ++ROW) {
00161         block = 1; // diagonal block
00162         for (BAND = 0; BAND < nband; ++BAND) {
00163             width = offsets[BAND];
00164             COL = ROW + width;
00165             if (width < 0) {
00166                 if (COL >= 0) ++block;
00167             }
00168             else {
00169                 if (COL < ngrid) ++block;
00170             }
00171         } // end for BAND
00172
00173         ncb = nc*block;
00174         row_start = ROW*nc;
00175
00176         for (i = 0; i < nc; i++) {
00177             row = row_start + i;
00178             ia[row+1] = ia[row] + ncb;
00179         }
00180     } // end for ROW
00181
00182     // allocate for 'ja' and 'a' arrays
00183     glo_nnz = ia[glo_row];

```

```

00184     ja = (INT *)fasp_mem_calloc(glo_nnz, sizeof(INT));
00185     a = (REAL *)fasp_mem_calloc(glo_nnz, sizeof(REAL));
00186
00187     // Generate the 'ja' and 'a' arrays at the same time
00188     for (ROW = 0; ROW < ngrid; ++ROW) {
00189         row_start = ROW*nc;
00190         val_L_start = ROW*nc2;
00191
00192         // deal with the diagonal band
00193         for (i = 0; i < nc; i++) {
00194             nci = nc*i;
00195             row = row_start + i;
00196             start = ia[row];
00197             for (j = 0; j < nc; j++) {
00198                 pos = start + j;
00199                 ja[pos] = row_start + j;
00200                 a[pos] = diag[val_L_start+nci+j];
00201             }
00202         }
00203         block = 1;
00204
00205         // deal with the off-diagonal bands
00206         for (BAND = 0; BAND < nband; ++BAND) {
00207             width = offsets[BAND];
00208             COL = ROW + width;
00209             ncb = nc*block;
00210             col_start = COL*nc;
00211
00212             if (width < 0) {
00213                 if (COL >= 0) {
00214                     val_R_start = COL*nc2;
00215                     for (i = 0; i < nc; i++) {
00216                         nci = nc*i;
00217                         row = row_start + i;
00218                         start = ia[row];
00219                         for (j = 0; j < nc; j++) {
00220                             pos = start + ncb + j;
00221                             ja[pos] = col_start + j;
00222                             a[pos] = offdiag[BAND][val_R_start+nci+j];
00223                         }
00224                     }
00225                     ++block;
00226                 }
00227             }
00228             else {
00229                 if (COL < ngrid) {
00230                     for (i = 0; i < nc; i++) {
00231                         nci = nc*i;
00232                         row = row_start + i;
00233                         start = ia[row];
00234                         for (j = 0; j < nc; j++) {
00235                             pos = start + ncb + j;
00236                             ja[pos] = col_start + j;
00237                             a[pos] = offdiag[BAND][val_L_start+nci+j];
00238                         }
00239                     }
00240                     ++block;
00241                 }
00242             }
00243         }
00244     }
00245
00246     // Reordering in such manner that every diagonal element
00247     // is firstly stored in the corresponding row
00248     if (nc > 1) {
00249         for (ROW = 0; ROW < ngrid; ++ROW) {
00250             row_start = ROW*nc;
00251             for (j = 1; j < nc; j++) {
00252                 row = row_start + j;
00253                 start = ia[row];
00254                 pos = start + j;
00255
00256                 // swap in 'ja'
00257                 tmp_col = ja[start];
00258                 ja[start] = ja[pos];
00259                 ja[pos] = tmp_col;
00260
00261                 // swap in 'a'
00262                 tmp_val = a[start];
00263                 a[start] = a[pos];
00264                 a[pos] = tmp_val;

```

```

00265     }
00266     }
00267 }
00268
00269 /* fill all the members of B_tmp */
00270 B_tmp.row = glo_row;
00271 B_tmp.col = glo_row;
00272 B_tmp.nnz = glo_nnz;
00273 B_tmp.IA = ia;
00274 B_tmp.JA = ja;
00275 B_tmp.val = a;
00276
00277 *B = B_tmp;
00278
00279 return FASP_SUCCESS;
00280 }
00281
00294 dCSRmat fasp_format_dblc_dcsr (const dBLCmat *Ab)
00295 {
00296     const INT mb=Ab->brow, nb=Ab->bcol, nbl=mb*nb;
00297     dCSRmat **blockptr=Ab->blocks, *blockptrij, A;
00298
00299     INT i,j,ij,ir,il,length,ilength,start,irmrow,irmrowl;
00300     INT *row, *col;
00301     INT m=0,n=0,nnz=0;
00302
00303     row = (INT *)fasp_mem_calloc(mb+1,sizeof(INT));
00304     col = (INT *)fasp_mem_calloc(nb+1,sizeof(INT));
00305
00306     // count the size of A
00307     row[0]=0; col[0]=0;
00308     for (i=0;i<mb;++i) { m+=blockptr[i*nb]->row; row[i+1]=m; }
00309     for (i=0;i<nb;++i) { n+=blockptr[i]->col; col[i+1]=n; }
00310
00311 #ifdef _OPENMP
00312 #pragma omp parallel for reduction(+:nnz) if (nbl>OPENMP_HOLDS) private(i)
00313 #endif
00314     for (i=0;i<nbl;++i) { nnz+=blockptr[i]->nnz; }
00315
00316     // memory space allocation
00317     A = fasp_dcsr_create(m,n,nnz);
00318
00319     // set dCSRmat for A
00320     A.IA[0]=0;
00321     for (i=0;i<mb;++i) {
00322         for (ir=row[i];ir<row[i+1];ir++) {
00323             for (length=j=0;j<nb;++j) {
00324                 ij=i*nb+j; blockptrij=blockptr[ij];
00325                 if (blockptrij->nnz>0) {
00326                     start=A.IA[ir]+length;
00327                     irmrow=ir-row[i]; irmrowl=irmrow+1;
00328                     ilength=blockptrij->IA[irmrowl]-blockptrij->IA[irmrow];
00329                     if (ilength>0) {
00330                         memcpy(&(A.val[start]), &(blockptrij->val[blockptrij->IA[irmrow]]), ilength*sizeof(REAL));
00331                         memcpy(&(A.JA[start]), &(blockptrij->JA[blockptrij->IA[irmrow]]),
00332                             ilength*sizeof(INT));
00333                     }
00334                     for (il=0;il<ilength;il++) A.JA[start+il]+=col[j];
00335                     length+=ilength;
00336                 }
00337             } // end for j
00338             A.IA[ir+1]=A.IA[ir]+length;
00339         } // end for ir
00340     } // end for i
00341
00342     fasp_mem_free(row); row = NULL;
00343     fasp_mem_free(col); col = NULL;
00344
00345     return(A);
00346 }
00347
00363 dCSRlmat * fasp_format_dcsrl_dcsr (const dCSRmat *A)
00364 {
00365     REAL *DATA = A -> val;
00366     INT *IA = A -> IA;
00367     INT *JA = A -> JA;

```

```

00368     INT      num_rows      = A -> row;
00369     INT      num_cols      = A -> col;
00370     INT      num_nonzeros  = A -> nnz;
00371
00372     dCSRLmat *B            = NULL;
00373     INT      dif;
00374     INT      *nzdifnum     = NULL;
00375     INT      *rowstart     = NULL;
00376     INT      *rowindex     = (INT *)fasp_mem_calloc(num_rows, sizeof(INT));
00377     INT      *ja           = (INT *)fasp_mem_calloc(num_nonzeros, sizeof(INT));
00378     REAL     *data         = (REAL *)fasp_mem_calloc(num_nonzeros, sizeof(REAL));
00379
00380     /* auxiliary arrays */
00381     INT *nzrow      = (INT *)fasp_mem_calloc(num_rows, sizeof(INT));
00382     INT *counter    = NULL;
00383     INT *invnzdif   = NULL;
00384
00385     INT i,j,k,cnt,maxnzrow;
00386
00387     //-----
00388     // Generate 'nzrow' and 'maxnzrow'
00389     //-----
00390
00391     maxnzrow = 0;
00392     for (i = 0; i < num_rows; i++) {
00393         nzrow[i] = IA[i+1] - IA[i];
00394         if (nzrow[i] > maxnzrow) {
00395             maxnzrow = nzrow[i];
00396         }
00397     }
00398     /* generate 'counter' */
00399     counter = (INT *)fasp_mem_calloc(maxnzrow + 1, sizeof(INT));
00400
00401     for (i = 0; i < num_rows; i++) {
00402         counter[nzrow[i]] ++;
00403     }
00404
00405     //-----
00406     // Determine 'dif'
00407     //-----
00408
00409     for (dif = 0, i = 0; i < maxnzrow + 1; i++) {
00410         if (counter[i] > 0) dif ++;
00411     }
00412
00413     //-----
00414     // Generate the 'nzdifnum' and 'rowstart'
00415     //-----
00416
00417     nzdifnum = (INT *)fasp_mem_calloc(dif, sizeof(INT));
00418     invnzdif = (INT *)fasp_mem_calloc(maxnzrow + 1, sizeof(INT));
00419     rowstart = (INT *)fasp_mem_calloc(dif + 1, sizeof(INT));
00420     rowstart[0] = 0;
00421     for (cnt = 0, i = 0; i < maxnzrow + 1; i++) {
00422         if (counter[i] > 0) {
00423             nzdifnum[cnt] = i;
00424             invnzdif[i] = cnt;
00425             rowstart[cnt+1] = rowstart[cnt] + counter[i];
00426             cnt ++;
00427         }
00428     }
00429
00430     //-----
00431     // Generate the 'rowindex'
00432     //-----
00433
00434     for (i = 0; i < num_rows; i++) {
00435         j = invnzdif[nzrow[i]];
00436         rowindex[rowstart[j]] = i;
00437         rowstart[j] ++;
00438     }
00439     /* recover 'rowstart' */
00440     for (i = dif; i > 0; i--) {
00441         rowstart[i] = rowstart[i-1];
00442     }
00443     rowstart[0] = 0;
00444
00445     //-----
00446     // Generate the 'data' and 'ja'
00447     //-----
00448

```

```

00449     for (cnt = 0, i = 0; i < num_rows; i++) {
00450         k = rowindex[i];
00451         for (j = IA[k]; j < IA[k+1]; j++) {
00452             data[cnt] = DATA[j];
00453             ja[cnt] = JA[j];
00454             cnt++;
00455         }
00456     }
00457
00458     //-----
00459     // Create and fill a dCSRLmat B
00460     //-----
00461
00462     B = fasp_dcsrl_create(num_rows, num_cols, num_nonzeros);
00463     B->dif = dif;
00464     B->nz_diff = nzdifnum;
00465     B->index = rowindex;
00466     B->start = rowstart;
00467     B->ja = ja;
00468     B->val = data;
00469
00470     //-----
00471     // Free the auxiliary arrays
00472     //-----
00473
00474     free(nzrow);
00475     free(counter);
00476     free(invnzdif);
00477
00478     return B;
00479 }
00480
00497 dCSRmat fasp_format_dbsr_dcsr (const dBSRmat *B)
00498 {
00499     dCSRmat A;
00500
00501     /* members of B */
00502     INT ROW = B->ROW;
00503     INT COL = B->COL;
00504     INT NNZ = B->NNZ;
00505     INT nb = B->nb;
00506     INT *IA = B->IA;
00507     INT *JA = B->JA;
00508     REAL *val = B->val;
00509
00510     INT storage_manner = B->storage_manner;
00511
00512     INT jump = nb*nb;
00513     INT rowA = ROW*nb;
00514     INT colA = COL*nb;
00515     INT nzA = NNZ*jump;
00516
00517     INT *ia = NULL;
00518     INT *ja = NULL;
00519     REAL *a = NULL;
00520
00521     INT i,j,k;
00522     INT mr,mc;
00523     INT rowstart0,rowstart,colstart0,colstart;
00524     INT colblock,nzperrow;
00525
00526     REAL *vp = NULL;
00527     REAL *ap = NULL;
00528     INT *jap = NULL;
00529
00530     SHORT use_omp = FALSE;
00531
00532     #ifdef _OPENMP
00533     INT stride_i,mybegin,myend,myid,nthreads;
00534     if ( ROW > OPENMP_HOLDS ) {
00535         use_omp = TRUE;
00536         nthreads = fasp_get_num_threads();
00537     }
00538     #endif
00539
00540     //-----
00541     // Create a CSR Matrix
00542     //-----
00543     A = fasp_dcsr_create(rowA, colA, nzA);
00544     ia = A.IA;
00545     ja = A.JA;

```

```

00546     a  = A.val;
00547
00548     //-----
00549     // Compute the number of nonzeros per row, and after this loop,
00550     // ia[i], i=1:rowA, will be the number of nonzeros of the (i-1)-th row.
00551     //-----
00552
00553     if (use_omp) {
00554 #ifdef _OPENMP
00555         stride_i = ROW/nthreads;
00556 #pragma omp parallel private(myid, mybegin, myend, i, rowstart, colblock, nzperrow, j)
00557         {
00558             myid = omp_get_thread_num();
00559             mybegin = myid*stride_i;
00560             if(myid < nthreads-1) myend = mybegin+stride_i;
00561             else myend = ROW;
00562             for (i=mybegin; i<myend; ++i)
00563             {
00564                 rowstart = i*nb + 1;
00565                 colblock = IA[i+1] - IA[i];
00566                 nzperrow = colblock*nb;
00567                 for (j = 0; j < nb; ++j)
00568                 {
00569                     ia[rowstart+j] = nzperrow;
00570                 }
00571             }
00572         }
00573 #endif
00574     }
00575     else {
00576         for (i = 0; i < ROW; ++i)
00577         {
00578             rowstart = i*nb + 1;
00579             colblock = IA[i+1] - IA[i];
00580             nzperrow = colblock*nb;
00581             for (j = 0; j < nb; ++j)
00582             {
00583                 ia[rowstart+j] = nzperrow;
00584             }
00585         }
00586     }
00587
00588     //-----
00589     // Generate the real 'ia' for CSR of A
00590     //-----
00591
00592     ia[0] = 0;
00593     for (i = 1; i <= rowA; ++i)
00594     {
00595         ia[i] += ia[i-1];
00596     }
00597
00598     //-----
00599     // Generate 'ja' and 'a' for CSR of A
00600     //-----
00601
00602     switch (storage_manner)
00603     {
00604         case 0: // each non-zero block elements are stored in row-major order
00605         {
00606             if (use_omp) {
00607 #ifdef _OPENMP
00608 #pragma omp parallel private(myid, mybegin, myend, i, k, j, rowstart, colstart, vp, mr, ap, jap, mc)
00609                 {
00610                     myid = omp_get_thread_num();
00611                     mybegin = myid*stride_i;
00612                     if(myid < nthreads-1) myend = mybegin+stride_i;
00613                     else myend = ROW;
00614                     for (i=mybegin; i<myend; ++i)
00615                     {
00616                         for (k = IA[i]; k < IA[i+1]; ++k)
00617                         {
00618                             j = JA[k];
00619                             rowstart = i*nb;
00620                             colstart = j*nb;
00621                             vp = &val[k*jump];
00622                             for (mr = 0; mr < nb; mr++)
00623                             {
00624                                 ap = &a[ia[rowstart]];
00625                                 jap = &a[ia[rowstart]];
00626                                 for (mc = 0; mc < nb; mc++)

```

```

00627             {
00628                 *ap = *vp;
00629                 *jap = colstart + mc;
00630                 vp ++; ap ++; jap ++;
00631             }
00632             ia[rowstart] += nb;
00633             rowstart ++;
00634         }
00635     }
00636 }
00637 }
00638 #endif
00639 }
00640 else {
00641     for (i = 0; i < ROW; ++i)
00642     {
00643         for (k = IA[i]; k < IA[i+1]; ++k)
00644         {
00645             j = JA[k];
00646             rowstart = i*nb;
00647             colstart = j*nb;
00648             vp = &val[k*jump];
00649             for (mr = 0; mr < nb; mr ++)
00650             {
00651                 ap = &a[ia[rowstart]];
00652                 jap = &ja[ia[rowstart]];
00653                 for (mc = 0; mc < nb; mc ++)
00654                 {
00655                     *ap = *vp;
00656                     *jap = colstart + mc;
00657                     vp ++; ap ++; jap ++;
00658                 }
00659                 ia[rowstart] += nb;
00660                 rowstart ++;
00661             }
00662         }
00663     }
00664 }
00665 }
00666 break;
00667
00668 case 1: // each non-zero block elements are stored in column-major order
00669 {
00670     for (i = 0; i < ROW; ++i)
00671     {
00672         for (k = IA[i]; k < IA[i+1]; ++k)
00673         {
00674             j = JA[k];
00675             rowstart0 = i*nb;
00676             colstart0 = j*nb;
00677             vp = &val[k*jump];
00678             for (mc = 0; mc < nb; mc ++)
00679             {
00680                 rowstart = rowstart0;
00681                 colstart = colstart0 + mc;
00682                 for (mr = 0; mr < nb; mr ++)
00683                 {
00684                     a[ia[rowstart]] = *vp;
00685                     ja[ia[rowstart]] = colstart;
00686                     vp ++; ia[rowstart]++; rowstart++;
00687                 }
00688             }
00689         }
00690     }
00691 }
00692 break;
00693 }
00694
00695 //-----
00696 // Map back the real 'ia' for CSR of A
00697 //-----
00698
00699 for (i = rowA; i > 0; i --) {
00700     ia[i] = ia[i-1];
00701 }
00702 ia[0] = 0;
00703
00704 return (A);
00705 }
00706
00723 dBSRmat fasp_format_dcsr_dbsr (const dCSRmat *A,

```

```

00724                                     const INT      nb)
00725 {
00726     INT i, j, k, ii, jj, kk, l, mod, nnz;
00727     INT row  = A->row/nb;
00728     INT col  = A->col/nb;
00729     INT nb2  = nb*nb;
00730     INT *IA  = A->IA;
00731     INT *JA  = A->JA;
00732     REAL *val = A->val;
00733
00734     dBSRmat B; // Safe-guard check
00735     INT *col_flag, *ia, *ja;
00736     REAL *bval;
00737
00738     if ((A->row)%nb!=0) {
00739         printf("### ERROR: A.row=%d is not a multiplication of nb=%d!\n",
00740             A->row, nb);
00741         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
00742     }
00743
00744     if ((A->col)%nb!=0) {
00745         printf("### ERROR: A.col=%d is not a multiplication of nb=%d!\n",
00746             A->col, nb);
00747         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
00748     }
00749
00750     B.ROW = row;
00751     B.COL = col;
00752     B.nb  = nb;
00753     B.storage_manner = 0;
00754
00755     // allocate memory for B
00756     col_flag = (INT *)fasp_mem_calloc(col, sizeof(INT));
00757     ia = (INT *) fasp_mem_calloc(row+1, sizeof(INT));
00758
00759     fasp_iarray_set(col, col_flag, -1);
00760
00761     // Get ia for BSR format
00762     nnz = 0;
00763     for (i=0; i<row; ++i) {
00764         ii = nb*i;
00765         for (j=0; j<nb; ++j) {
00766             jj = ii+j;
00767             for (k=IA[jj]; k<IA[jj+1]; ++k) {
00768                 kk = JA[k]/nb;
00769                 if (col_flag[kk]!=0) {
00770                     col_flag[kk] = 0;
00771                     //ja[nnz] = kk;
00772                     nnz ++;
00773                 }
00774             }
00775         }
00776         ia[i+1] = nnz;
00777         fasp_iarray_set(col, col_flag, -1);
00778     }
00779
00780     // set NNZ
00781     B.NNZ = nnz;
00782
00783     // allocate ja and bval
00784     ja = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00785     bval = (REAL*)fasp_mem_calloc(nnz*nb2, sizeof(REAL));
00786
00787     // Get ja for BSR format
00788     nnz = 0;
00789     for (i=0; i<row; ++i) {
00790         ii = nb*i;
00791         for(j=0; j<nb; ++j) {
00792             jj = ii+j;
00793             for(k=IA[jj]; k<IA[jj+1]; ++k) {
00794                 kk = JA[k]/nb;
00795                 if (col_flag[kk]!=0) {
00796                     col_flag[kk] = 0;
00797                     ja[nnz] = kk;
00798                     nnz ++;
00799                 }
00800             }
00801         }
00802         ia[i+1] = nnz;
00803         fasp_iarray_set(col, col_flag, -1);
00804     }

```



```

00805
00806 // Get non-zeros of BSR
00807 for (i=0; i<row; ++i) {
00808     ii = nb*i;
00809     for (j=0; j<nb; ++j) {
00810         jj = ii+j;
00811         for (k=IA[jj]; k<IA[jj+1]; ++k) {
00812             for (l=ia[i]; l<ia[i+1]; ++l) {
00813                 if (JA[k]/nb == ja[l]) {
00814                     mod = JA[k]%nb;
00815                     bval[l*nb2+j*nb+mod] = val[k];
00816                     break;
00817                 }
00818             }
00819         }
00820     }
00821 }
00822
00823 B.IA = ia;
00824 B.JA = ja;
00825 B.val = bval;
00826
00827 fasp_mem_free(col_flag); col_flag = NULL;
00828
00829 return B;
00830 }
00831
00844 dBSRmat fasp_format_dstr_dbsr (const dSTRmat *B)
00845 {
00846     // members of 'B'
00847     INT      nc      = B->nc;
00848     INT      ngrid   = B->ngrid;
00849     REAL     *diag   = B->diag;
00850     INT      nband   = B->nband;
00851     INT      *offsets = B->offsets;
00852     REAL     **offdiag = B->offdiag;
00853
00854     // members of 'A'
00855     dBSRmat A;
00856     INT      NNZ;
00857     INT      *IA = NULL;
00858     INT      *JA = NULL;
00859     REAL     *val = NULL;
00860
00861     // local variables
00862     INT i,j,k,m;
00863     INT nc2 = nc*nc;
00864     INT ngridplus1 = ngrid + 1;
00865
00866     // compute NNZ
00867     NNZ = ngrid;
00868     for (i = 0; i < nband; ++i) {
00869         NNZ += (ngrid - abs(offsets[i]));
00870     }
00871
00872     // Create and Initialize a dBSRmat 'A'
00873     A = fasp_dbsr_create(ngrid, ngrid, NNZ, nc, 0);
00874     IA = A.IA;
00875     JA = A.JA;
00876     val = A.val;
00877
00878     // Generate 'IA'
00879     for (i = 1; i < ngridplus1; ++i) IA[i] = 1; // take the diagonal blocks into account
00880     for (i = 0; i < nband; ++i) {
00881         k = offsets[i];
00882         if (k < 0) {
00883             for (j = -k+1; j < ngridplus1; ++j) {
00884                 IA[j] ++;
00885             }
00886         }
00887         else {
00888             m = ngridplus1 - k;
00889             for (j = 1; j < m; ++j)
00890             {
00891                 IA[j] ++;
00892             }
00893         }
00894     }
00895     IA[0] = 0;
00896     for (i = 1; i < ngridplus1; ++i) {
00897         IA[i] += IA[i-1];

```

```

00898     }
00899
00900     // Generate 'JA' and 'val' at the same time
00901     for (i = 0 ; i < ngrid; ++i) {
00902         memcpy(val + IA[i]*nc2, diag + i*nc2, nc2*sizeof(REAL));
00903         JA[IA[i]] = i;
00904         IA[i] ++;
00905     }
00906
00907     for (i = 0; i < nband; ++i) {
00908         k = offsets[i];
00909         if (k < 0) {
00910             for (j = -k; j < ngrid; ++j) {
00911                 m = j + k;
00912                 memcpy(val+IA[j]*nc2, offdiag[i]+m*nc2, nc2*sizeof(REAL));
00913                 JA[IA[j]] = m;
00914                 IA[j] ++;
00915             }
00916         }
00917         else {
00918             m = ngrid - k;
00919             for (j = 0; j < m; ++j) {
00920                 memcpy(val + IA[j]*nc2, offdiag[i] + j*nc2, nc2*sizeof(REAL));
00921                 JA[IA[j]] = k + j;
00922                 IA[j] ++;
00923             }
00924         }
00925     }
00926
00927     // Map back the real 'IA' for BSR of A
00928     for (i = ngrid; i > 0; i --) {
00929         IA[i] = IA[i-1];
00930     }
00931     IA[0] = 0;
00932
00933     return (A);
00934 }
00935
00948 dCOOmat * fasp_format_dbsr_dcoo (const dBSRmat *B)
00949 {
00950     /* members of B */
00951     INT    ROW = B->ROW;
00952     INT    COL = B->COL;
00953     INT    NNZ = B->NNZ;
00954     INT    nb  = B->nb;
00955     INT    *IA = B->IA;
00956     INT    *JA = B->JA;
00957     REAL    *val = B->val;
00958
00959     dCOOmat *A = NULL;
00960     INT    nb2 = nb*nb;
00961     INT    num_nonzeros = NNZ*nb2;
00962     INT    *rowA = NULL;
00963     INT    *colA = NULL;
00964     REAL    *valA = NULL;
00965
00966     INT    i,j,k,inb;
00967     INT    row_start, col_start;
00968     INT    cnt,mr,mc;
00969     REAL    *pt = NULL;
00970
00971     // Create and Initialize a dCOOmat 'A'
00972     A = (dCOOmat *)fasp_mem_malloc(1, sizeof(dCOOmat));
00973     A->row = ROW*nb;
00974     A->col = COL*nb;
00975     A->nnz = num_nonzeros;
00976     rowA = (INT *)fasp_mem_malloc(num_nonzeros, sizeof(INT));
00977     colA = (INT *)fasp_mem_malloc(num_nonzeros, sizeof(INT));
00978     valA = (REAL *)fasp_mem_malloc(num_nonzeros, sizeof(REAL));
00979     A->rowind = rowA;
00980     A->colind = colA;
00981     A->val = valA;
00982
00983     cnt = 0;
00984     for (i = 0; i < ROW; ++i) {
00985         inb = i*nb;
00986         for (k = IA[i]; k < IA[i+1]; ++k) {
00987             j = JA[k];
00988             pt = &val[k*nb2];
00989             row_start = inb;
00990             col_start = j*nb;

```

```

00991         for (mr = 0; mr < nb; mr++) {
00992             for (mc = 0; mc < nb; mc++) {
00993                 rowA[cnt] = row_start;
00994                 colA[cnt] = col_start + mc;
00995                 valA[cnt] = (*pt);
00996                 pt++;
00997                 cnt++;
00998             }
00999             row_start++;
01000         }
01001     }
01002 }
01003
01004     return (A);
01005 }
01006
01007 /*-----*/
01008 /*--      End of File      --*/
01009 /*-----*/

```

9.51 BlalLU.c File Reference

Incomplete LU decomposition: ILUk, ILUt, ILUtp.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_iluk](#) (INT n, REAL *a, INT *ja, INT *ia, INT lfil, REAL *alu, INT *jlu, INT iwk, INT *ierr, INT *nzlu)
Get ILU factorization with level of fill-in k (ilu(k)) for a CSR matrix A.
- void [fasp_ilut](#) (INT n, REAL *a, INT *ja, INT *ia, INT lfil, REAL droptol, REAL *alu, INT *jlu, INT iwk, INT *ierr, INT *nz)
Get incomplete LU factorization with dual truncations of a CSR matrix A.
- void [fasp_ilutp](#) (INT n, REAL *a, INT *ja, INT *ia, INT lfil, REAL droptol, REAL permtol, INT mbloc, REAL *alu, INT *jlu, INT *iperm, INT iwk, INT *ierr, INT *nz)
Get incomplete LU factorization with pivoting dual truncations of a CSR matrix A.
- void [fasp_symbfactor](#) (INT n, INT *colind, INT *rwptr, INT levfill, INT nzmax, INT *nzlu, INT *ijlu, INT *uptr, INT *ierr)
Symbolic factorization of a CSR matrix A in compressed sparse row format, with resulting factors stored in a single MSR data structure.

9.51.1 Detailed Description

Incomplete LU decomposition: ILUk, ILUt, ILUtp.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#)

Translated from SparseKit (Fortran code) by Chunsheng Feng, 09/03/2016
Copyright (C) 2016–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlalLU.c](#).

9.51.2 Function Documentation

9.51.2.1 fasp_iluk()

```
void fasp_iluk (
    INT n,
    REAL * a,
    INT * ja,
    INT * ia,
    INT lfil,
    REAL * alu,
    INT * jlu,
    INT iwk,
    INT * ierr,
    INT * nzlu )
```

Get ILU factorization with level of fill-in k (ilu(k)) for a CSR matrix A.

Parameters

<i>n</i>	row number of A
<i>a</i>	nonzero entries of A
<i>ja</i>	integer array of column for A
<i>ia</i>	integer array of row pointers for A
<i>lfil</i>	integer. The fill-in parameter. Each row of L and each row of U will have a maximum of lfil elements (excluding the diagonal element). lfil must be .ge. 0.
<i>alu</i>	matrix stored in Modified Sparse Row (MSR) format containing the L and U factors together. The diagonal (stored in alu(1:n)) is inverted. Each i-th row of the alu,jlu matrix contains the i-th row of L (excluding the diagonal entry=1) followed by the i-th row of U.
<i>jlu</i>	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
<i>iwk</i>	integer. The minimum length of arrays alu, jlu, and levs.
<i>ierr</i>	integer pointer. Return error message with the following meaning. 0 --> successful return. >0 --> zero pivot encountered at step number ierr. -1 --> Error. input matrix may be wrong. (The elimination process has generated a row in L or U whose length is .gt. n.) -2 --> The matrix L overflows the array al. -3 --> The matrix U overflows the array alu. -4 --> Illegal value for lfil. -5 --> zero row encountered.
<i>nzlu</i>	integer pointer. Return number of nonzero entries for alu and jlu

Note

: All the diagonal elements of the input matrix must be nonzero.

Author

Chunsheng Feng

Date

09/06/2016

Definition at line 72 of file BlalLU.c.

9.51.2.2 fasp_ilut()

```
void fasp_ilut (
    INT n,
    REAL * a,
    INT * ja,
    INT * ia,
    INT lfil,
    REAL droptol,
    REAL * alu,
    INT * jlu,
    INT iwk,
    INT * ierr,
    INT * nz )
```

Get incomplete LU factorization with dual truncations of a CSR matrix A.

Parameters

<i>n</i>	row number of A
<i>a</i>	nonzero entries of A
<i>ja</i>	integer array of column for A
<i>ia</i>	integer array of row pointers for A
<i>lfil</i>	integer. The fill-in parameter. Each row of L and each row of U will have a maximum of lfil elements (excluding the diagonal element). lfil must be .ge. 0.
<i>droptol</i>	real*8. Sets the threshold for dropping small terms in the factorization. See below for details on dropping strategy.
<i>alu</i>	matrix stored in Modified Sparse Row (MSR) format containing the L and U factors together. The diagonal (stored in alu(1:n)) is inverted. Each i-th row of the alu,jlu matrix contains the i-th row of L (excluding the diagonal entry=1) followed by the i-th row of U.
<i>jlu</i>	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
<i>iwk</i>	integer. The lengths of arrays alu and jlu. If the arrays are not big enough to store the ILU factorizations, ilut will stop with an error message.
<i>ierr</i>	integer pointer. Return error message with the following meaning. 0 --> successful return. >0 --> zero pivot encountered at step number ierr. -1 --> Error. input matrix may be wrong. (The elimination process has generated a row in L or U whose length is .gt. n.) -2 --> The matrix L overflows the array al. -3 --> The matrix U overflows the array alu. -4 --> Illegal value for lfil. -5 --> zero row encountered.
<i>nz</i>	integer pointer. Return number of nonzero entries for alu and jlu

Note

All the diagonal elements of the input matrix must be nonzero.

Author

Chunsheng Feng

Date

09/06/2016

Definition at line 467 of file BlalLU.c.

9.51.2.3 fasp_ilutp()

```
void fasp_ilutp (
    INT n,
    REAL * a,
    INT * ja,
    INT * ia,
    INT lfil,
    REAL droptol,
    REAL permtol,
    INT mbloc,
    REAL * alu,
    INT * jlu,
    INT * iperm,
    INT iwk,
    INT * ierr,
    INT * nz )
```

Get incomplete LU factorization with pivoting dual truncations of a CSR matrix A.

Parameters

<i>n</i>	row number of A
<i>a</i>	nonzero entries of A
<i>ja</i>	integer array of column for A
<i>ia</i>	integer array of row pointers for A
<i>lfil</i>	integer. The fill-in parameter. Each row of L and each row of U will have a maximum of lfil elements (excluding the diagonal element). lfil must be .ge. 0.
<i>droptol</i>	real*8. Sets the threshold for dropping small terms in the factorization. See below for details on dropping strategy.
<i>permtol</i>	tolerance ratio used to determine whether or not to permute two columns. At step i columns i and j are permuted when $\text{abs}(a(i,j)) * \text{permtol} > \text{abs}(a(i,i))$ [0 --> never permute; good values 0.1 to 0.01]
<i>mbloc</i>	integer. If desired, permuting can be done only within the diagonal blocks of size mbloc. Useful for PDE problems with several degrees of freedom.. If feature not wanted take mbloc=n.
<i>alu</i>	matrix stored in Modified Sparse Row (MSR) format containing the L and U factors together. The diagonal (stored in alu(1:n)) is inverted. Each i-th row of the alu,jlu matrix contains the i-th row of L (excluding the diagonal entry=1) followed by the i-th row of U.
<i>jlu</i>	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
<i>iperm</i>	permutation arrays
<i>iwk</i>	integer. The lengths of arrays alu and jlu. If the arrays are not big enough to store the ILU factorizations, ilut will stop with an error message.
<i>ierr</i>	integer pointer. Return error message with the following meaning. 0 --> successful return. >0 --> zero pivot encountered at step number ierr. -1 --> Error. input matrix may be wrong. (The elimination process has generated a row in L or U whose length is .gt. n.) -2 --> The matrix L overflows the array al. -3 --> The matrix U overflows the array alu. -4 --> Illegal value for lfil. -5 --> zero row encountered.
<i>nz</i>	integer pointer. Return number of nonzero entries for alu and jlu

Note

: All the diagonal elements of the input matrix must be nonzero.

Author

Chunsheng Feng

Date

09/06/2016

Definition at line 906 of file [BlalLU.c](#).**9.51.2.4 fasp_symbfactor()**

```
void fasp_symbfactor (
    INT n,
    INT * colind,
    INT * rwptr,
    INT levfill,
    INT nzmax,
    INT * nzlu,
    INT * ijlu,
    INT * uptr,
    INT * ierr )
```

Symbolic factorization of a CSR matrix A in compressed sparse row format, with resulting factors stored in a single MSR data structure.

Parameters

<i>n</i>	row number of A
<i>colind</i>	integer array of column for A
<i>rwptr</i>	integer array of row pointers for A
<i>levfill</i>	integer. Level of fill-in allowed
<i>nzmax</i>	integer. The maximum number of nonzero entries in the approximate factorization of a. This is the amount of storage allocated for ijlu.
<i>nzlu</i>	integer pointer. Return number of nonzero entries for alu and jlu
<i>ijlu</i>	integer array of length nzlu containing pointers to delimit rows and specify column number for stored elements of the approximate factors of A. the L and U factors are stored as one matrix.
<i>uptr</i>	integer array of length n containing the pointers to upper trig matrix
<i>ierr</i>	integer pointer. Return error message with the following meaning. 0 --> successful return. 1 --> not enough storage; check mneed.

Author

Chunsheng Feng

Date

09/06/2016

Symbolic factorization of a matrix in compressed sparse row format, * with resulting factors stored in a single MSR data structure. *

This routine uses the CSR data structure of A in two integer vectors * colind, rwptr to set up the data structure for the ILU(levfill) * factorization of A in the integer vectors ijlu and uptr. Both L * and U are stored in the same structure, and uptr(i) is the pointer * to the beginning of the i-th row of U in ijlu. *

Method Used * ===== *

The implementation assumes that the diagonal entries are * nonzero, and remain nonzero throughout the elimination * process. The algorithm proceeds row by row. When computing * the sparsity pattern of the i-th row, the effect of row * operations from previous rows is considered. Only those * preceding rows j for which (i,j) is nonzero need be considered, * since otherwise we would not have formed a linear combination * of rows i and j. *

The method used has some variations possible. The definition * of ILU(s) is not well specified enough to get a factorization * that is uniquely defined, even in the sparsity pattern that * results. For s = 0 or 1, there is not much variation, but for * higher levels of fill the problem is as follows: Suppose * during the decomposition while computing the nonzero pattern * for row i the following principal submatrix is obtained: *

Furthermore, suppose that entry (i,j) resulted from an earlier * fill-in and has level s1, and (j,k) resulted from an earlier * fill-in and has level s2: *

When using A(j,j) to annihilate A(i,j), fill-in will be incurred * in A(i,k). How should its level be defined? It would not be * operated on if A(i,j) or A(j,m) had not been filled in. The * version used here is to define its level as s1 + s2 + 1. However, * other reasonable choices would have been min(s1,s2) or max(s1,s2). * Using the sum gives a more conservative strategy in terms of the * growth of the number of nonzeros as s increases. *

levels(n+2:nzlu) stores the levels from previous rows, * that is, the s2's above. levels(1:n) stores the fill-levels * of the current row (row i), which are the s1's above. * levels(n+1) is not used, so levels is conformant with MSR format. *

Vectors used: * ===== *

lastcol(n): * The integer lastcol(k) is the row index of the last row * to have a nonzero in column k, including the current * row, and fill-in up to this point. So for the matrix *

```
-----
* | 11 12 15 | * | 21 22 26 | * | 32 33 34 | * | 41 43 44 | * | 52 54 55 56 | * | 62 66 | * -----
----- *
```

after step 1, lastcol() = [1 0 0 0 1 0] * after step 2, lastcol() = [2 2 0 0 2 2] * after step 3, lastcol() = [2 3 3 3 2 3] * after step 4, lastcol() = [4 3 4 4 4 3] * after step 5, lastcol() = [4 5 4 5 5 5] * after step 6, lastcol() = [4 6 4 5 5 6] *

Note that on step 2, lastcol(5) = 2 because there is a * fillin position (2,5) in the matrix. lastcol() is used * to determine if a nonzero occurs in column j because * it is a nonzero in the original matrix, or was a fill. *

rowll(n): * The integer vector rowll is used to keep a linked list of * the nonzeros in the current row, allowing fill-in to be * introduced sensibly. rowll is initialized with the * original nonzeros of the current row, and then sorted * using a shell sort. A pointer called head * (what ingenuity) is initialized. Note that at any * point rowll may contain garbage left over from previous * rows, which the linked list structure skips over. * For row 4 of the matrix above, first rowll is set to * rowll() = [3 1 2 5 -], where - indicates any integer. * Then the vector is sorted, which yields * rowll() = [1 2 3 5 -]. The vector is then expanded * to linked list form by setting head = 1 and * rowll() = [2 3 5 - 7 -], where 7 indicates termination. *

ijlu(nzlu): * The returned nonzero structure for the LU factors. * This is built up row by row in MSR format, with both L * and U stored in the data structure. Another vector, uptr(n), * is used to give pointers to the beginning of the upper * triangular part of the LU factors in ijlu. *

levels(n+2:nzlu): * This vector stores the fill level for each entry from * all the previous rows, used to compute if the current entry * will exceed the allowed levels of fill. The value in * levels(m) is added to the level of fill for the element in * the current row that is being reduced, to figure if * a column entry is to be accepted as fill, or rejected. * See the method explanation above. *

levels(1:n): * This vector stores the fill level number for the current * row's entries. If they were created as fill elements * themselves, this number is added to the corresponding * entry in levels(n+2:nzlu) to see if a particular column * entry will * be created as new fill or not. NOTE: in practice, the * value in levels(1:n) is one larger than the "fill" level of * the corresponding row entry, except for the diagonal * entry. That is why the accept/reject test in the code * is "if (levels(j) + levels(m) .le. levfill + 1)". *

on entry:

n = The order of the matrix A. ija = Integer array. Matrix A stored in modified sparse row format. levfill = Integer. Level of fill-in allowed. nzmax = Integer. The maximum number of nonzero entries in the approximate factorization of a. This is the amount of storage allocated for ijlu.

on return:

nzlu = The actual number of entries in the approximate factors, plus one. ijlu = Integer array of length nzlu containing pointers to delimit rows and specify column number for stored elements of the approximate factors of a. the l and u factors are stored as one matrix. uptr = Integer array of length n containing the pointers to upper trig matrix
ierr is an error flag: ierr = -i --> near zero pivot in step i ierr = 0 --> all's OK ierr = 1 --> not enough storage; check mneed. ierr = 2 --> illegal parameter
mneed = contains the actual number of elements in ldu, or the amount of additional storage needed for ldu

work arrays:

lastcol = integer array of length n containing last update of the corresponding column. levels = integer array of length n containing the level of fill-in in current row in its first n entries, and level of fill of previous rows of U in remaining part.
rowl = integer array of length n containing pointers to implement a linked list for the fill-in elements.

external functions:

ifix, float, min0, srr
Definition at line 1372 of file BlalLU.c.

9.52 BlalLU.c

[Go to the documentation of this file.](#)

```

00001
00016 #include <math.h>
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_funcs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 static void fasp_qsplit (REAL *a, INT *ind, INT n, INT ncut);
00027 static void fasp_sortrow (INT num,INT *q);
00028 static void fasp_check_col_index (INT row, INT num, INT *q);
00029
00030 /*-----*/
00031 /*--      Public Functions      --*/
00032 /*-----*/
00033
00072 void fasp_iluk (INT      n,
00073                REAL    *a,
00074                INT      *ja,
00075                INT      *ia,
00076                INT      lfil,
00077                REAL    *alu,
00078                INT      *jlu,
00079                INT      iwk,
00080                INT      *ierr,
00081                INT      *nzlu)
00082 {
00083     #if DEBUG_MODE > 0
00084         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00085     #endif
00086
00087     /*-----*/
00088     SPARSKIT ROUTINE ILUK -- ILU WITH LEVEL OF FILL-IN OF K (ILU(k))
00089     -----
00090
00091     on entry:
00092     =====
00093     n          = integer.  The row dimension of the matrix A. The matrix
00094
00095     a,ja,ia = matrix stored in Compressed Sparse Row format.
00096
00097     lfil      = integer.  The fill-in parameter.  Each element whose
00098     leve-of-fill exceeds lfil during the ILU process is dropped.
00099     lfil must be .ge.  0

```

```

00100
00101 iwk      = integer.  The minimum length of arrays alu, jlu, and levs.
00102
00103 On return:
00104 =====
00105
00106 alu,jlu = matrix stored in Modified Sparse Row (MSR) format containing
00107 the L and U factors together.  The diagonal (stored in
00108 alu(1:n) ) is inverted.  Each i-th row of the alu,jlu matrix
00109 contains the i-th row of L (excluding the diagonal entry=1)
00110 followed by the i-th row of U.
00111
00112 jlu      = integer array of length n containing the pointers to
00113 the beginning of each row of U in the matrix alu,jlu.
00114
00115 levs     = integer (work) array of size iwk -- which contains the
00116 levels of each element in alu, jlu.
00117
00118 ierr     = integer.  Error message with the following meaning.
00119 ierr = 0    --> successful return.
00120 ierr .gt. 0 --> zero pivot encountered at step number ierr.
00121 ierr = -1   --> Error.  input matrix may be wrong.
00122 (The elimination process has generated a
00123 row in L or U whose length is .gt.  n.)
00124 ierr = -2   --> The matrix L overflows the array al.
00125 ierr = -3   --> The matrix U overflows the array alu.
00126 ierr = -4   --> Illegal value for lfil.
00127 ierr = -5   --> zero row encountered in A or U.
00128
00129 work arrays:
00130 =====
00131 jw       = integer work array of length 3*n.
00132 w        = real work array of length n
00133
00134 -----
00135 w, ju (1:n) store the working array [1:ii-1 = L-part, ii:n = U-part]
00136 jw(n+1:2n) stores the nonzero indicator.
00137
00138 Notes:
00139 -----
00140 All the diagonal elements of the input matrix must be nonzero.
00141 ----- */
00142
00143 // locals
00144 INT ju0, k, j1, j2, j, ii, i, lenl, lenu, jj, jrow, jpos, n2, jlev, NE;
00145 REAL t, s, fact;
00146 SHORT cinindex=0;
00147 REAL *w;
00148 INT *ju, *jw, *levs;
00149
00150 if (lfil < 0) goto F998;
00151
00152 w = (REAL *)fasp_mem_calloc(n, sizeof(REAL));
00153 ju = (INT *)fasp_mem_calloc(n, sizeof(INT));
00154 jw = (INT *)fasp_mem_calloc(3*n, sizeof(INT));
00155 levs = (INT *)fasp_mem_calloc(iwk, sizeof(INT));
00156
00157 --jw;
00158 --w;
00159 --ju;
00160 --jlu;
00161 --alu;
00162 --ia;
00163 --ja;
00164 --a;
00165 --levs;
00166
00167 /*-----
00168 shift index for C routines
00169 -----*/
00170 if (ia[1] == 0) cinindex=1 ;
00171 if (cinindex)
00172 {
00173     NE = n + 1; //modify by chunsheng 2012, Sep, 1;
00174     for (i=1; i<=NE; ++i) ++ia[i];
00175     NE = ia[n+1] - 1;
00176     for (i=1; i<=NE; ++i) ++ja[i];
00177 }
00178
00179 /*-----
00180 initialize ju0 (points to next element to be added to alu,jlu)

```

```

00181 and pointer array.
00182 -----*/
00183     n2 = n + n;
00184     ju0 = n + 2;
00185     jlu[1] = ju0;
00186
00187     // initialize nonzero indicator array + levs array --
00188     for(j = 1; j <= 2*n; ++j) jw[j] = 0;
00189
00190     /*-----
00191 beginning of main loop.
00192 -----*/
00193     for(ii = 1; ii <= n; ++ii) { //500
00194         j1 = ia[ii];
00195         j2 = ia[ii + 1] - 1;
00196
00197         // unpack L-part and U-part of row of A in arrays w
00198         lenu = 1;
00199         lenl = 0;
00200         jw[ii] = ii;
00201         w[ii] = 0.0;
00202         jw[n + ii] = ii;
00203
00204         //
00205         for(j = j1; j <= j2; ++j) { //170
00206             k = ja[j];
00207             t = a[j];
00208             if (t == 0.0) continue; //goto g170;
00209             if (k < ii) {
00210                 ++lenl;
00211                 jw[lenl] = k;
00212                 w[lenl] = t;
00213                 jw[n2 + lenl] = 0;
00214                 jw[n + k] = lenl;
00215             } else if (k == ii) {
00216                 w[ii] = t;
00217                 jw[n2 + ii] = 0;
00218             } else {
00219                 ++lenu;
00220                 jpos = ii + lenu - 1;
00221                 jw[jpos] = k;
00222                 w[jpos] = t;
00223                 jw[n2 + jpos] = 0;
00224                 jw[n + k] = jpos;
00225             }
00226         } //170
00227
00228         jj = 0;
00229         // eliminate previous rows
00230
00231         F150:
00232         ++jj;
00233         if (jj > lenl) goto F160;
00234
00235         /*-----
00236 in order to do the elimination in the correct order we must select
00237 the smallest column index among jw(k), k=jj+1, ..., lenl.
00238 -----*/
00239
00240         jrow = jw[jj];
00241         k = jj;
00242
00243         // determine smallest column index
00244         for(j = jj + 1; j <= lenl; ++j) { //151
00245             if (jw[j] < jrow) {
00246                 jrow = jw[j];
00247                 k = j;
00248             }
00249         } //151
00250
00251         if (k != jj) {
00252             // exchange in jw
00253             j = jw[jj];
00254             jw[jj] = jw[k];
00255             jw[k] = j;
00256             // exchange in jw(n+ (pointers/ nonzero indicator).
00257             jw[n + jrow] = jj;
00258             jw[n + j] = k;
00259             // exchange in jw(n2+ (levels)
00260             j = jw[n2 + jj];
00261

```

```

00262         jw[n2 + jj] = jw[n2 + k];
00263         jw[n2 + k] = j;
00264         //      exchange in w
00265         s = w[jj];
00266         w[jj] = w[k];
00267         w[k] = s;
00268     }
00269
00270     //      zero out element in row by resetting jw(n+jrow) to zero.
00271     jw[n + jrow] = 0;
00272
00273     //      get the multiplier for row to be eliminated (jrow) + its level
00274     fact = w[jj]*alu[jrow];
00275     jlev = jw[n2 + jj];
00276     if (jlev > lfil) goto F150;
00277
00278     //      combine current row and row jrow
00279     for(k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k ) { // 203
00280         s = fact*alu[k];
00281         j = jlu[k];
00282         jpos = jw[n + j];
00283         if (j >= ii) {
00284             //      dealing with upper part.
00285             if (jpos == 0) {
00286                 //      this is a fill-in element
00287                 ++lenu;
00288                 if (lenu > n) goto F995;
00289                 i = ii + lenu - 1;
00290                 jw[i] = j;
00291                 jw[n + j] = i;
00292                 w[i] = -s;
00293                 jw[n2 + i] = jlev + levs[k] + 1;
00294             } else {
00295                 //      this is not a fill-in element
00296                 w[jpos] = w[jpos] - s;
00297                 jw[n2 + jpos] = MIN(jw[n2 + jpos], jlev + levs[k] + 1);
00298             }
00299         } else {
00300             //      dealing with lower part.
00301             if (jpos == 0) {
00302                 //      this is a fill-in element
00303                 ++lenl;
00304                 if (lenl > n) goto F995;
00305                 jw[lenl] = j;
00306                 jw[n + j] = lenl;
00307                 w[lenl] = -s;
00308                 jw[n2 + lenl] = jlev + levs[k] + 1;
00309             } else {
00310                 //      this is not a fill-in element
00311                 w[jpos] = w[jpos] - s;
00312                 jw[n2 + jpos] = MIN(jw[n2 + jpos], jlev + levs[k] + 1);
00313             }
00314         }
00315     }
00316     //203
00317     w[jj] = fact;
00318     jw[jj] = jrow;
00319     goto F150;
00320
00321 F160:
00322     //      reset double-pointer to zero (U-part)
00323     for(k = 1; k <= lenu; ++k) jw[n + jw[ii + k - 1]] = 0;
00324
00325     //      update l-matrix
00326     for(k = 1; k <= lenl; ++k ) { //204
00327         if (ju0 > iwk) goto F996;
00328         if (jw[n2 + k] <= lfil) {
00329             alu[ju0] = w[k];
00330             jlu[ju0] = jw[k];
00331             ++ju0;
00332         }
00333     } //204
00334
00335     //      save pointer to beginning of row ii of U
00336     ju[ii] = ju0;
00337
00338     //      update u-matrix
00339     for(k = ii + 1; k <= ii + lenu - 1; ++k ) { //302
00340         if (ju0 > iwk) goto F997;
00341
00342         if (jw[n2 + k] <= lfil) {

```

```

00343         jlu[ju0] = jw[k];
00344         alu[ju0] = w[k];
00345         levs[ju0] = jw[n2 + k];
00346         ++ju0;
00347     }
00348
00349     } //302
00350
00351     if (w[ii] == 0.0) goto F999;
00352     //
00353     alu[ii] = 1.0/w[ii];
00354
00355     // update pointer to beginning of next row of U.
00356     jlu[ii + 1] = ju0;
00357     /*-----
00358 end main loop
00359 -----*/
00360 } //500
00361
00362 *nzlu = ju[n] - 1;
00363
00364 if (cinindex) {
00365     for ( i = 1; i <= *nzlu; ++i ) --jlu[i];
00366 }
00367
00368 *ierr = 0;
00369
00370 F100:
00371 ++jw;
00372 ++w;
00373 ++ju;
00374 ++jlu;
00375 ++alu;
00376 ++ia;
00377 ++ja;
00378 ++a;
00379 ++levs;
00380
00381 fasp_mem_free(w);    w = NULL;
00382 fasp_mem_free(ju);   ju = NULL;
00383 fasp_mem_free(jw);   jw = NULL;
00384 fasp_mem_free(levs); levs = NULL;
00385
00386 #if DEBUG_MODE > 0
00387 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00388 #endif
00389
00390 return;
00391
00392 // incomprehensible error. Matrix must be wrong.
00393 F995:
00394 printf("### ERROR: Incomprehensible error. [%s]\n", __FUNCTION__);
00395 *ierr = -1;
00396 goto F100;
00397
00398 // insufficient storage in L.
00399 F996:
00400 printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
00401 *ierr = -2;
00402 goto F100;
00403
00404 // insufficient storage in U.
00405 F997:
00406 printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
00407 *ierr = -3;
00408 goto F100;
00409
00410 // illegal lfil entered.
00411 F998:
00412 printf("### ERROR: Illegal lfil entered. [%s]\n", __FUNCTION__);
00413 *ierr = -4;
00414 return;
00415
00416 // zero row encountered in A or U.
00417 F999:
00418 printf("### ERROR: Zero row encountered in A or U. [%s]\n", __FUNCTION__);
00419 *ierr = -5;
00420 goto F100;
00421 /*-----end-of-iluk-----
00422 -----*/
00423 }

```

```

00424
00467 void fasp_ilut (INT      n,
00468                 REAL    *a,
00469                 INT      *ja,
00470                 INT      *ia,
00471                 INT      lfil,
00472                 REAL     droptol,
00473                 REAL     *alu,
00474                 INT      *jlu,
00475                 INT      iwk,
00476                 INT      *ierr,
00477                 INT      *nz)
00478 {
00479     #if DEBUG_MODE > 0
00480         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00481     #endif
00482
00483     /*-----*
00484     *** ILUT preconditioner ***
00485     incomplete LU factorization with dual truncation mechanism
00486     -----*
00487     Author: Yousef Saad *May, 5, 1990, Latest revision, August 1996 *
00488     -----*
00489     PARAMETERS
00490     -----
00491
00492     on entry:
00493     =====
00494     n          = integer.  The row dimension of the matrix A. The matrix
00495
00496     a,ja,ia = matrix stored in Compressed Sparse Row format.
00497
00498     lfil       = integer.  The fill-in parameter.  Each row of L and each row
00499     of U will have a maximum of lfil elements (excluding the diagonal
00500     element).  lfil must be .ge.  0.
00501
00502     droptol = real*8.  Sets the threshold for dropping small terms in the
00503     factorization.  See below for details on dropping strategy.
00504
00505     iwk       = integer.  The lengths of arrays alu and jlu.  If the arrays
00506     are not big enough to store the ILU factorizations, ilut
00507     will stop with an error message.
00508
00509     On return:
00510     =====
00511
00512     alu,jlu = matrix stored in Modified Sparse Row (MSR) format containing
00513     the L and U factors together.  The diagonal (stored in
00514     alu(1:n) ) is inverted.  Each i-th row of the alu,jlu matrix
00515     contains the i-th row of L (excluding the diagonal entry=1)
00516     followed by the i-th row of U.
00517
00518     ju       = integer array of length n containing the pointers to
00519     the beginning of each row of U in the matrix alu,jlu.
00520
00521     ierr      = integer.  Error message with the following meaning.
00522     ierr = 0    --> successful return.
00523     ierr .gt. 0 --> zero pivot encountered at step number ierr.
00524     ierr = -1   --> Error.  input matrix may be wrong.
00525     (The elimination process has generated a
00526     row in L or U whose length is .gt.  n.)
00527     ierr = -2   --> The matrix L overflows the array al.
00528     ierr = -3   --> The matrix U overflows the array alu.
00529     ierr = -4   --> Illegal value for lfil.
00530     ierr = -5   --> zero row encountered.
00531
00532     work arrays:
00533     =====
00534     jw       = integer work array of length 2*n.
00535     w        = real work array of length n+1.
00536
00537     -----
00538     w, ju (1:n) store the working array [1:ii-1 = L-part, ii:n = u]
00539     jw(n+1:2n) stores nonzero indicators
00540
00541     Notes:
00542     -----
00543     The diagonal elements of the input matrix must be nonzero (at least
00544     'structurally').
00545
00546     -----*

```

```

00547 ---- Dual drop strategy works as follows.
00548 *
00549 1) Thresholding in L and U as set by droptol. Any element whose *
00550 magnitude is less than some tolerance (relative to the abs *
00551 value of diagonal element in u) is dropped.
00552 *
00553 2) Keeping only the largest lfil elements in the i-th row of L *
00554 and the largest lfil elements in the i-th row of U (excluding *
00555 diagonal elements).
00556 *
00557 Flexibility: one can use droptol=0 to get a strategy based on *
00558 keeping the largest elements in each row of L and U. Taking *
00559 droptol .ne. 0 but lfil=n will give the usual threshold strategy *
00560 (however, fill-in is then unpredictable).
00561 ----- */
00562
00563 // locals
00564 INT ju0, k, j1, j2, j, ii, i, lenl, lenu, jj, jrow, jpos, NE, len;
00565 REAL t, s, fact, tmp;
00566 SHORT cinindex=0;
00567 REAL *w, *tnorm;
00568 INT *ju, *jw;
00569
00570 if (lfil < 0) goto F998;
00571
00572 ju = (INT *)fasp_mem_calloc(n, sizeof(INT));
00573 jw = (INT *)fasp_mem_calloc(2*n, sizeof(INT));
00574 w = (REAL *)fasp_mem_calloc(n+1, sizeof(REAL));
00575 tnorm = (REAL *)fasp_mem_calloc(n, sizeof(REAL));
00576
00577 --jw;
00578 --ju;
00579 --w;
00580 --tnorm;
00581 --jlu;
00582 --alu;
00583 --ia;
00584 --ja;
00585 --a;
00586
00587 if (ia[1] == 0) cinindex=1;
00588
00589 if (cinindex)
00590 {
00591     NE = n + 1; //modify by chunsheng 2012, Sep, 1;
00592     for (i=1; i<=NE; ++i) ++ia[i];
00593     NE = ia[n+1] - 1;
00594     for (i=1; i<=NE; ++i) ++ja[i];
00595 }
00596
00597 /*-----
00598 initialize ju0 (points to next element to be added to alu,jlu)
00599 and pointer array.
00600 -----*/
00601 ju0 = n + 2;
00602 jlu[1] = ju0;
00603
00604 // initialize nonzero indicator array.
00605 for (j = 1; j<=n; ++j) jw[n + j] = 0;
00606
00607 /*-----
00608 beginning of main loop.
00609 -----*/
00610 for (ii = 1; ii <= n; ++ii) {
00611     j1 = ia[ii];
00612     j2 = ia[ii + 1] - 1;
00613     tmp = 0.0;
00614     for (k = j1; k<= j2; ++k) tmp = tmp + ABS(a[k]);
00615     tmp = tmp/(REAL)(j2 - j1 + 1);
00616     tnorm[ii] = tmp*droptol;
00617 }
00618
00619 for (ii = 1; ii<=n; ++ii) {
00620     j1 = ia[ii];
00621     j2 = ia[ii + 1] - 1;
00622
00623     // unpack L-part and U-part of row of A in arrays w
00624     lenu = 1;
00625     lenl = 0;
00626     jw[ii] = ii;

```

```

00627     w[ii] = 0.0;
00628     jw[n + ii] = ii;
00629
00630     for(j = j1; j<=j2; ++j) {
00631         k = ja[j];
00632         t = a[j];
00633         if (k < ii) {
00634             ++len1;
00635             jw[len1] = k;
00636             w[len1] = t;
00637             jw[n + k] = len1;
00638         } else if (k == ii) {
00639             w[ii] = t;
00640         } else {
00641             ++lenu ;
00642             jpos = ii + lenu - 1;
00643             jw[jpos] = k;
00644             w[jpos] = t;
00645             jw[n + k] = jpos;
00646         }
00647     }
00648     jj = 0;
00649     len = 0;
00650
00651     //      eliminate previous rows
00652 F150:
00653     ++jj;
00654     if (jj > len1) goto F160;
00655
00656     /*-----
00657 in order to do the elimination in the correct order we must select
00658 the smallest column index among jw(k), k=jj+1, ..., len1.
00659 -----*/
00660     jrow = jw[jj];
00661     k = jj;
00662
00663     /*
00664 determine smallest column index
00665 */
00666     for(j = jj + 1; j<=len1; ++j) { //151
00667         if (jw[j] < jrow) {
00668             jrow = jw[j];
00669             k = j;
00670         }
00671     } //151
00672
00673     if (k != jj) {
00674         // exchange in jw
00675         j = jw[jj];
00676         jw[jj] = jw[k];
00677         jw[k] = j;
00678         // exchange in jr
00679         jw[n + jrow] = jj;
00680         jw[n + j] = k;
00681         // exchange in w
00682         s = w[jj];
00683         w[jj] = w[k];
00684         w[k] = s;
00685     }
00686
00687     // zero out element in row by setting jw(n+jrow) to zero.
00688     jw[n + jrow] = 0;
00689
00690     // get the multiplier for row to be eliminated (jrow).
00691     fact = w[jj]*alu[jrow];
00692
00693     if (ABS(fact) <= droptol) goto F150;
00694
00695     // combine current row and row jrow
00696     for ( k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k) { //203
00697         s = fact*alu[k];
00698         j = jlu[k];
00699         jpos = jw[n + j];
00700         if (j >= ii) {
00701             //      dealing with upper part.
00702             if (jpos == 0)
00703             {
00704                 //      this is a fill-in element
00705                 ++lenu;
00706                 if (lenu > n) goto F995;
00707                 i = ii + lenu - 1;

```



```

00708         jw[i] = j;
00709         jw[n + j] = i;
00710         w[i] = -s;
00711     } else {
00712         // this is not a fill-in element
00713         w[jpos] = w[jpos] - s;
00714     }
00715 } else {
00716     // dealing with lower part.
00717     if (jpos == 0) {
00718         // this is a fill-in element
00719         ++lenl;
00720         if (lenl > n) goto F995;
00721         jw[lenl] = j;
00722         jw[n + j] = lenl;
00723         w[lenl] = -s;
00724     } else {
00725         // this is not a fill-in element
00726         w[jpos] = w[jpos] - s;
00727     }
00728 }
00729 } //203
00730
00731 /*
00732 store this pivot element -- (from left to right -- no danger of
00733 overlap with the working elements in L (pivots).
00734 */
00735 ++len;
00736 w[len] = fact;
00737 jw[len] = jrow;
00738 goto F150;
00739
00740 F160:
00741 // reset double-pointer to zero (U-part)
00742 for (k = 1; k <= lenu; ++k) jw[n + jw[ii + k - 1]] = 0; //308
00743
00744 // update L-matrix
00745 lenl = len;
00746 len = MIN(lenl, lfil);
00747
00748 // sort by quick-split
00749 fasp_qspllit(&w[1], &jw[1], lenl, len);
00750
00751 // store L-part
00752 for (k = 1; k <= len; ++k) { //204
00753     if (ju0 > iwk) goto F996;
00754     alu[ju0] = w[k];
00755     jlu[ju0] = jw[k];
00756     ++ju0;
00757 }
00758
00759 // save pointer to beginning of row ii of U
00760 ju[ii] = ju0;
00761
00762 // update U-matrix -- first apply dropping strategy
00763 len = 0;
00764 for (k = 1; k <= lenu - 1; ++k) {
00765     // if (ABS(w[ii + k]) > droptol*tnorm)
00766     if (ABS(w[ii + k]) > tnorm[ii]) {
00767         ++len;
00768         w[ii + len] = w[ii + k];
00769         jw[ii + len] = jw[ii + k];
00770     }
00771 }
00772
00773 lenu = len + 1;
00774 len = MIN(lenu, lfil);
00775
00776 fasp_qspllit(&w[ii + 1], &jw[ii + 1], lenu - 1, len);
00777
00778 // copy
00779 t = ABS(w[ii]);
00780 if (len + ju0 > iwk) goto F997;
00781 for (k = ii + 1; k <= ii + len - 1; ++k) { //302
00782     jlu[ju0] = jw[k];
00783     alu[ju0] = w[k];
00784     t = t + ABS(w[k]);
00785     ++ju0;
00786 }
00787
00788 // store inverse of diagonal element of u

```

```

00789      // if (w(ii) .eq. 0.0) w(ii) = (0.0001 + droptol)*tnorm
00790      if (w[ii] == 0.0) w[ii] = tnorm[ii];
00791
00792      alu[ii] = 1.0/w[ii];
00793
00794      // update pointer to beginning of next row of U.
00795      jlu[ii + 1] = ju0;
00796      /*-----
00797 end main loop
00798 ----- */
00799   }
00800
00801   *nz = ju[n] - 1;
00802
00803   if (cinindex) {
00804     for(i = 1; i <= *nz; ++i) --jlu[i];
00805   }
00806
00807   *ierr = 0;
00808
00809 F100:
00810   ++jw;
00811   ++ju;
00812   ++w;
00813   ++tnorm;
00814   ++jlu;
00815   ++alu;
00816   ++ia;
00817   ++ja;
00818   ++a;
00819
00820   fasp_mem_free(ju);      ju = NULL;
00821   fasp_mem_free(jw);      jw = NULL;
00822   fasp_mem_free(w);       w = NULL;
00823   fasp_mem_free(tnorm);   tnorm = NULL;
00824
00825   #if DEBUG_MODE > 0
00826   printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00827   #endif
00828
00829   return;
00830
00831 F995:      // incomprehensible error. Matrix must be wrong.
00832   printf("### ERROR: Input matrix may be wrong. [%s]\n", __FUNCTION__);
00833   *ierr = -1;
00834   goto F100;
00835
00836 F996:      // insufficient storage in L.
00837   printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
00838   *ierr = -2;
00839   goto F100;
00840
00841 F997:      // insufficient storage in U.
00842   printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
00843   *ierr = -3;
00844   goto F100;
00845
00846 F998:      // illegal lfil entered.
00847   *ierr = -4;
00848   printf("### ERROR: Illegal lfil entered. [%s]\n", __FUNCTION__);
00849   return;
00850   /*-----end-of-ilut-----
00851 ----- */
00852 }
00853
00906 void fasp_ilutp (INT      n,
00907                  REAL     *a,
00908                  INT      *ja,
00909                  INT      *ia,
00910                  INT      lfil,
00911                  REAL     droptol,
00912                  REAL     permtol,
00913                  INT      mbloc,
00914                  REAL     *alu,
00915                  INT      *jlu,
00916                  INT      *iperm,
00917                  INT      iwkw,
00918                  INT      *ierr,
00919                  INT      *nz)
00920 {
00921   #if DEBUG_MODE > 0

```

```

00922     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00923 #endif
00924
00925     /*-----*
00926     *** ILUTP preconditioner -- ILUT with pivoting ***
00927     incomplete LU factorization with dual truncation mechanism
00928     -----*
00929     author Yousef Saad *Sep 8, 1993 -- Latest revision, August 1996. *
00930     -----*
00931     on entry:
00932     =====
00933     n           = integer. The dimension of the matrix A.
00934
00935     a,ja,ia = matrix stored in Compressed Sparse Row format.
00936     ON RETURN THE COLUMNS OF A ARE PERMUTED. SEE BELOW FOR
00937     DETAILS.
00938
00939     lfil       = integer. The fill-in parameter. Each row of L and each row
00940     of U will have a maximum of lfil elements (excluding the
00941     diagonal element). lfil must be .ge. 0.
00942     ** WARNING: THE MEANING OF LFIL HAS CHANGED WITH RESPECT TO
00943     EARLIER VERSIONS.
00944
00945     droptol = real*8. Sets the threshold for dropping small terms in the
00946     factorization. See below for details on dropping strategy.
00947
00948     lfil       = integer. The fill-in parameter. Each row of L and
00949     each row of U will have a maximum of lfil elements.
00950
00951     permtol = tolerance ratio used to determne whether or not to permute
00952     two columns. At step i columns i and j are permuted when
00953     abs(a(i,j))*permtol .gt. abs(a(i,i))
00954     [0 --> never permute; good values 0.1 to 0.01]
00955
00956     mbloc      = if desired, permuting can be done only within the diagonal
00957     blocks of size mbloc. Useful for PDE problems with several
00958     degrees of freedom.. If feature not wanted take mbloc=n.
00959
00960     iwk        = integer. The lengths of arrays alu and jlu. If the arrays
00961     are not big enough to store the ILU factorizations, ilut
00962     will stop with an error message.
00963
00964     On return:
00965     =====
00966
00967     alu,jlu = matrix stored in Modified Sparse Row (MSR) format containing
00968     the L and U factors together. The diagonal (stored in
00969     alu(1:n) ) is inverted. Each i-th row of the alu,jlu matrix
00970     contains the i-th row of L (excluding the diagonal entry=1)
00971     followed by the i-th row of U.
00972
00973     ju        = integer array of length n containing the pointers to
00974     the beginning of each row of U in the matrix alu,jlu.
00975
00976     iperm      = contains the permutation arrays.
00977     iperm(1:n) = old numbers of unknowns
00978     iperm(n+1:2*n) = reverse permutation = new unknowns.
00979
00980     ierr       = integer. Error message with the following meaning.
00981     ierr = 0    --> successful return.
00982     ierr .gt. 0 --> zero pivot encountered at step number ierr.
00983     ierr = -1   --> Error. input matrix may be wrong.
00984     (The elimination process has generated a
00985     row in L or U whose length is .gt. n.)
00986     ierr = -2   --> The matrix L overflows the array al.
00987     ierr = -3   --> The matrix U overflows the array alu.
00988     ierr = -4   --> Illegal value for lfil.
00989     ierr = -5   --> zero row encountered.
00990
00991     work arrays:
00992     =====
00993     jw        = integer work array of length 2*n.
00994     w         = real work array of length n
00995
00996     IMPORTANT NOTE:
00997     -----
00998     TO AVOID PERMUTING THE SOLUTION VECTORS ARRAYS FOR EACH LU-SOLVE,
00999     THE MATRIX A IS PERMUTED ON RETURN. [all column indices are
01000     changed]. SIMILARLY FOR THE U MATRIX.

```

```

01003 To permute the matrix back to its original state use the loop:
01004
01005 do k=ia(1), ia(n+1)-1
01006   ja(k) = iperm(ja(k))
01007 enddo
01008
01009 -----*/
01010
01011 // local variables
01012 INT k, i, j, jrow, ju0, ii, j1, j2, jpos, len, imax, lenu, lenl, jj, icut, NE;
01013 REAL s, tmp, tnorm, xmax, xmax0, fact, t;
01014 SHORT cinindex=0;
01015 REAL *w;
01016 INT *ju, *jw;
01017
01018 if (lfil < 0) goto F998;
01019
01020 ju = (INT *) fasp_mem_calloc(n, sizeof(INT));
01021 jw = (INT *) fasp_mem_calloc(2*n, sizeof(INT));
01022 w = (REAL *) fasp_mem_calloc(n+1, sizeof(REAL));
01023
01024 --ju;
01025 --jw;
01026 --iperm;
01027 --w;
01028 --jlu;
01029 --alu;
01030 --ia;
01031 --ja;
01032 --a;
01033
01034 /*-----
01035 shift index for C routines
01036 -----*/
01037 if (ia[1] == 0) cinindex=1 ;
01038
01039 if (cinindex)
01040 {
01041   NE = n + 1; //modify by chunsheng 2012, Sep, 1;
01042   for (i=1; i<=NE; ++i) ++ia[i];
01043   NE = ia[n+1] - 1;
01044   for (i=1; i<=NE; ++i) ++ja[i];
01045 }
01046
01047 /*-----
01048 initialize ju0 (points to next element to be added to alu,jlu)
01049 and pointer array.
01050 -----*/
01051 ju0 = n + 2;
01052 jlu[1] = ju0;
01053
01054 // integer double pointer array.
01055 for ( j = 1; j <= n; ++j ) { //1
01056   jw[n + j] = 0;
01057   iperm[j] = j;
01058   iperm[n + j] = j;
01059 } //1
01060
01061 /*-----
01062 beginning of main loop.
01063 -----*/
01064 for (ii = 1; ii <= n; ++ii ) { //500
01065   j1 = ia[ii];
01066   j2 = ia[ii + 1] - 1;
01067
01068   tnorm = 0.0;
01069   for (k = j1; k <= j2; ++k ) tnorm = tnorm + ABS( a[k] ); //501
01070   if (tnorm == 0.0) goto F999;
01071   tnorm = tnorm/(REAL)(j2 - j1 + 1);
01072
01073   // unpack L-part and U-part of row of A in arrays w --
01074   lenu = 1;
01075   lenl = 0;
01076   jw[ii] = ii;
01077   w[ii] = 0.0;
01078   jw[n + ii] = ii;
01079   //
01080   for (j = j1; j <= j2; ++j ) { // 170
01081     k = iperm[n + ja[j]];
01082     t = a[j];

```

```

01084         if (k < ii) {
01085             ++lenl;
01086             jw[lenl] = k;
01087             w[lenl] = t;
01088             jw[n + k] = lenl;
01089         } else if (k == ii) {
01090             w[ii] = t;
01091         } else {
01092             ++lenu;
01093             jpos = ii + lenu - 1;
01094             jw[jpos] = k;
01095             w[jpos] = t;
01096             jw[n + k] = jpos;
01097         }
01098     } //170
01099
01100     jj = 0;
01101     len = 0;
01102
01103
01104     // eliminate previous rows
01105 F150:
01106     ++jj;
01107     if (jj > lenl) goto F160;
01108
01109     /*-----
01110 in order to do the elimination in the correct order we must select
01111 the smallest column index among jw(k), k=jj+1, ..., lenl.
01112 -----*/
01113     jrow = jw[jj];
01114     k = jj;
01115
01116     // determine smallest column index
01117     for (j = jj + 1; j <= lenl; ++j) { //151
01118         if (jw[j] < jrow) {
01119             jrow = jw[j];
01120             k = j;
01121         }
01122     }
01123
01124     if (k != jj) {
01125         // exchange in jw
01126         j = jw[jj];
01127         jw[jj] = jw[k];
01128         jw[k] = j;
01129         // exchange in jr
01130         jw[n + jrow] = jj;
01131         jw[n + j] = k;
01132         // exchange in w
01133         s = w[jj];
01134         w[jj] = w[k];
01135         w[k] = s;
01136     }
01137
01138     // zero out element in row by resetting jw(n+jrow) to zero.
01139     jw[n + jrow] = 0;
01140
01141     // get the multiplier for row to be eliminated: jrow
01142     fact = w[jj]*alu[jrow];
01143
01144     // drop term if small
01145     if (ABS(fact) <= droptol) goto F150;
01146
01147     // combine current row and row jrow
01148
01149     for (k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k) { //203
01150         s = fact*alu[k];
01151         // new column number
01152         j = iperm[n + jlu[k]];
01153         jpos = jw[n + j];
01154         if (j >= ii) {
01155             // dealing with upper part.
01156             if (jpos == 0) {
01157                 // this is a fill-in element
01158                 ++lenu;
01159                 i = ii + lenu - 1;
01160                 if (lenu > n) goto F995;
01161                 jw[i] = j;
01162                 jw[n + j] = i;
01163                 w[i] = -s;
01164             } else {

```

```

01165         //      no fill-in element --
01166         w[jpos] = w[jpos] - s;
01167     }
01168
01169     } else {
01170         // dealing with lower part.
01171         if (jpos == 0) {
01172             //      this is a fill-in element
01173             ++lenl;
01174             if (lenl > n) goto F995;
01175             jw[lenl] = j;
01176             jw[n + j] = lenl;
01177             w[lenl] = -s;
01178         } else {
01179             //      this is not a fill-in element
01180             w[jpos] = w[jpos] - s;
01181         }
01182     }
01183 } //203
01184
01185 /*
01186 store this pivot element -- (from left to right -- no danger of
01187 overlap with the working elements in L (pivots).
01188 */
01189
01190 ++len;
01191 w[len] = fact;
01192 jw[len] = jrow;
01193 goto F150;
01194
01195 F160:
01196 // reset double-pointer to zero (U-part)
01197 for ( k = 1; k <= lenu; ++k ) jw[n + jw[ii + k - 1]] = 0; //308
01198
01199 // update L-matrix
01200 lenl = len;
01201 len = MIN(lenl, lfil);
01202
01203 // sort by quick-split
01204 fasp_qsplit(&w[1], &jw[1], lenl, len);
01205
01206 // store L-part -- in original coordinates ..
01207 for ( k = 1; k <= len; ++k ) { // 204
01208     if (ju0 > iw[k]) goto F996;
01209     alu[ju0] = w[k];
01210     jlu[ju0] = iperm[jw[k]];
01211     ++ju0;
01212 } //204
01213
01214 // save pointer to beginning of row ii of U
01215 ju[ii] = ju0;
01216
01217 // update U-matrix -- first apply dropping strategy
01218 len = 0;
01219 for(k = 1; k <= lenu - 1; ++k ) {
01220     if ( ABS(w[ii + k]) > droptol*tnorm) {
01221         ++len;
01222         w[ii + len] = w[ii + k];
01223         jw[ii + len] = jw[ii + k];
01224     }
01225 }
01226
01227 lenu = len + 1;
01228 len = MIN(lenu, lfil);
01229 fasp_qsplit(&w[ii + 1], &jw[ii + 1], lenu-1, len);
01230
01231 // determine next pivot --
01232 imax = ii;
01233 xmax = ABS(w[imax]);
01234 xmax0 = xmax;
01235 icut = ii - 1 + mbloc - (ii - 1)%mbloc;
01236
01237 for ( k = ii + 1; k <= ii + len - 1; ++k ) {
01238     t = ABS(w[k]);
01239     if ((t > xmax) && (t*permtol > xmax0) && (jw[k] <= icut)) {
01240         imax = k;
01241         xmax = t;
01242     }
01243 }
01244
01245 // exchange w's

```

```

01246         tmp = w[ii];
01247         w[ii] = w[imax];
01248         w[imax] = tmp;
01249
01250         // update iperm and reverse iperm
01251         j = jw[imax];
01252         i = iperm[ii];
01253         iperm[ii] = iperm[j];
01254         iperm[j] = i;
01255
01256         // reverse iperm
01257         iperm[n + iperm[ii]] = ii;
01258         iperm[n + iperm[j]] = j;
01259
01260         //-----
01261         if (len + ju0 > iwk) goto F997;
01262
01263
01264         // copy U-part in original coordinates
01265         for ( k = ii + 1; k <= ii + len - 1; ++k ) { //302
01266             jlu[ju0] = iperm[jw[k]];
01267             alu[ju0] = w[k];
01268             ++ju0;
01269         }
01270
01271         // store inverse of diagonal element of u
01272         if (w[ii] == 0.0) w[ii] = (1.0e-4 + droptol)*tnorm;
01273         alu[ii] = 1.0/w[ii];
01274
01275         // update pointer to beginning of next row of U.
01276         jlu[ii + 1] = ju0;
01277
01278         /*-----
01279 end main loop
01280 -----*/
01281     } //500
01282
01283     // permute all column indices of LU ...
01284     for ( k = jlu[1]; k <= jlu[n + 1] - 1; ++k )    jlu[k] = iperm[n + jlu[k]];
01285
01286     // ...and of A
01287     for ( k = ia[1]; k <= ia[n + 1] - 1; ++k )    ja[k] = iperm[n + ja[k]];
01288
01289     *nz = ju[n] - 1;
01290
01291     if (cinindex) {
01292         for ( i = 1; i <= *nz; ++i ) --jlu[i];
01293     }
01294
01295     *ierr = 0;
01296
01297 F100:
01298     ++jw;
01299     ++ju;
01300     ++iperm;
01301     ++w;
01302     ++jlu;
01303     ++alu;
01304     ++ia;
01305     ++ja;
01306     ++a;
01307
01308     fasp_mem_free(ju);    ju = NULL;
01309     fasp_mem_free(jw);    jw = NULL;
01310     fasp_mem_free(w);     w = NULL;
01311
01312 #if DEBUG_MODE > 0
01313     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01314 #endif
01315
01316     return;
01317
01318 F995:        // incomprehensible error. Matrix must be wrong.
01319     printf("### ERROR: Input matrix may be wrong. [%s]\n", __FUNCTION__);
01320     *ierr = -1;
01321     goto F100;
01322
01323 F996:        // insufficient storage in L.
01324     printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
01325     *ierr = -2;
01326     goto F100;

```

```

01327
01328 F997:          // insufficient storage in U.
01329     printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
01330     *ierr = -3;
01331     goto F100;
01332
01333 F998:          // illegal lfil entered.
01334     printf("### ERROR: Illegal lfil entered. [%s]\n", __FUNCTION__);
01335     *ierr = -4;
01336     // goto F100;
01337     return;
01338
01339 F999:          // zero row encountered
01340     printf("### ERROR: Zero row encountered. [%s]\n", __FUNCTION__);
01341     *ierr = -5;
01342     goto F100;
01343     //-----end-of-ilutp-----
01344 }
01345
01372 void fasp_symbfactor (INT    n,
01373                        INT    *colind,
01374                        INT    *rowptr,
01375                        INT    levfill,
01376                        INT    nzmax,
01377                        INT    *nzlu,
01378                        INT    *ijlu,
01379                        INT    *uptr,
01380                        INT    *ierr)
01381 {
01382     #if DEBUG_MODE > 0
01383     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01384     #endif
01385
01579     INT icolindj, ijlu, i, j, k, m, ibegin, iend, Ujbeg, Ujend, NE;
01580     INT head, prev, lm, actlev, lowct, k1, k2, levpl, lmk, nzi, rowct;
01581     SHORT cinindex=0;
01582     INT *rowll, *lastcol, *levels;
01583
01584     rowll = (INT *)fasp_mem_calloc(n, sizeof(INT));
01585     lastcol = (INT *)fasp_mem_calloc(n, sizeof(INT));
01586     levels = (INT *)fasp_mem_calloc(nzmax, sizeof(INT));
01587
01588     //=====
01589     //      Beginning of Executable Statements
01590     //=====
01591
01592     /*-----
01593 shift index for C routines
01594 -----*/
01595     --rowll;
01596     --lastcol;
01597     --levels;
01598     --colind;
01599     --rowptr;
01600     --ijlu;
01601     --uptr;
01602
01603     if (rowptr[1] == 0) cinindex=1 ;
01604     if (cinindex) {
01605         NE = n + 1;
01606         for (i=1; i<=NE; ++i) ++rowptr[i];
01607         NE = rowptr[n+1] - 1;
01608         for (i=1; i<=NE; ++i) ++colind[i];
01609     }
01610
01611     // -----
01612     // Because the first row of the factor contains no strictly lower
01613     // triangular parts (parts of L), uptr(1) = ijlu(1) = n+2:
01614     // -----
01615     ijlu[1] = n + 2;
01616     uptr[1] = n + 2;
01617
01618     // -----
01619     // The storage for the nonzeros of LU must be at least n+1,
01620     // for a diagonal matrix:
01621     // -----
01622     *nzlu = n + 1;
01623
01624     // -----
01625     // Number of allowed levels plus 1; used for the test of accept/reject.
01626     // See the notes about the methodology above.

```



```

01627 // -----
01628 levpl = levfill + 1;
01629
01630 // -----
01631 // Initially, for all columns there were no nonzeros in the rows
01632 // above, because there are no rows above the first one.
01633 // -----
01634 for (i = 1; i<=n; ++i) lastcol[i] = 0;
01635
01636 // -----
01637 // Proceed row by row:
01638 // -----
01639 for (i = 1; i <= n; ++i) { // 100
01640
01641 // -----
01642 // Because the matrix diagonal entry is nonzero, the level of
01643 // fill for that diagonal entry is zero:
01644 // -----
01645 levels[i] = 0;
01646
01647 // -----
01648 // ibegin and iend are the beginning of rows i and i+1, resp.
01649 // -----
01650 ibegin = rwptr[i];
01651 iend = rwptr[i + 1];
01652
01653 // -----
01654 // Number of offdiagonal nonzeros in the original matrix's row i
01655 // -----
01656 nzi = iend - ibegin;
01657
01658 // -----
01659 // If only the diagonal entry in row i is nonzero, skip the
01660 // fancy stuff; nothing need be done:
01661 // -----
01662 if (nzi > 1) {
01663 // -----
01664 // Decrement iend, so that it can be used as the ending index
01665 // in icolind of row i:
01666 // -----
01667 iend = iend - 1;
01668
01669 // -----
01670 // rowct keeps count of the number of nondiagonal entries in
01671 // the current row:
01672 // -----
01673 rowct = 0;
01674
01675 // -----
01676 // For nonzeros in the current row from the original matrix A,
01677 // set lastcol to be the current row number, and the levels of
01678 // the entry to be 1. Note that this is really the true level
01679 // of the element, plus 1. At the same time, load up the work
01680 // array rowll with the column numbers for the original entries
01681 // from row i:
01682 // -----
01683 #if DEBUG_MODE > 0
01684 printf("### DEBUG: %s %d row\n", __FUNCTION__, i);
01685 #endif
01686
01687 for (j = ibegin; j <= iend; ++j) {
01688 icolindj = colind[j];
01689 lastcol[icolindj] = i;
01690 if (icolindj != i) {
01691 levels[icolindj] = 1;
01692 rowct = rowct + 1;
01693 rowll[rowct] = icolindj;
01694 }
01695 #if DEBUG_MODE > 0
01696 printf("### DEBUG: %d\n", icolindj);
01697 #endif
01698 }
01699
01700 // -----
01701 // Sort the entries in rowll, so that the row has its column
01702 // entries in increasing order.
01703 // -----
01704 fasp_sortrow(nzi - 1, &rowll[1]);
01705
01706 //check col index
01707 fasp_check_col_index(i, nzi-1, &rowll[1]);

```

```

01708 // -----
01709 // Now set up rowll as a linked list containing the original
01710 // nonzero column numbers, as described in the methods section:
01711 // -----
01712 head = rowll[1];
01713 k1 = n + 1;
01714 for (j = nzi - 1; j >= 1; --j) {
01715     k2 = rowll[j];
01716     rowll[k2] = k1;
01717     k1 = k2;
01718 }
01719
01720 // -----
01721 // Increment count of nonzeros in the LU factors by the number
01722 // of nonzeros in the original matrix's row i. Further
01723 // incrementing will be necessary if any fill-in actually occurs
01724 // -----
01725 *nzlu = *nzlu + nzi - 1;
01726
01727 // -----
01728 // The integer j will be used as a pointer to track through the
01729 // linked list rowll:
01730 // -----
01731 j = head;
01732
01733 // -----
01734 // The integer lowct is used to keep count of the number of
01735 // nonzeros in the current row's strictly lower triangular part,
01736 // for setting uptr pointers to indicate where in ijlu the upperc
01737 // triangular part starts.
01738 // -----
01739 lowct = 0;
01740
01741 // -----
01742 // Fill-in could only have resulted from rows preceding row i,
01743 // so we only need check those rows with index j < i.
01744 // Furthermore, if the current row has a zero in column j,
01745 // there is no need to check the preceding rows; there clearly
01746 // could not be any fill-in from those rows to this entry.
01747 // -----
01748 while (j < i) { //80
01749     // -----
01750     // Increment lower triangular part count, since in this case
01751     // (j<i) we got another entry in L:
01752     // -----
01753     lowct = lowct + 1;
01754
01755     // -----
01756     // If the fill level is zero, there is no way to get fill in
01757     // occurring.
01758     // -----
01759     if (levfill != 0) {
01760         // -----
01761         // Ujbeg is beginning index of strictly upper triangular
01762         // part of U's j-th row, and Ujend is the ending index
01763         // of it, in ijlu().
01764         // -----
01765         Ujbeg = uptr[j];
01766         Ujend = ijlu[j + 1] - 1;
01767
01768         // -----
01769         // Need to set pointer to previous entry before working
01770         // segment of rowll, because if fill occurs that will be
01771         // a moving segment.
01772         // -----
01773         prev = j;
01774
01775         // -----
01776         // lm is the next nonzero pointer in linked list rowll:
01777         // -----
01778         lm = rowll[j];
01779
01780         // -----
01781         // lmk is the fill level in this row, caused by
01782         // eliminating column entry j. That is, level sl from the
01783         // methodology explanation above.
01784         // -----
01785         lmk = levels[j];
01786
01787         // -----
01788         // -----

```

```

01789 // Now proceed through the j-th row of U, because in the
01790 // elimination we add a multiple of it to row i to zero
01791 // out entry (i,j). If a column entry in row j of U is
01792 // zero, there is no need to worry about fill, because it
01793 // cannot cause a fill in the corresponding entry of row i
01794 // -----
01795 for (m = Ujbeg; m <= Ujend; ++m) { //60
01796 // -----
01797 // ijlum is the column number of the current nonzero in
01798 // row j of U:
01799 // -----
01800 ijlum = ijlum[m];
01801
01802 // -----
01803 // actlev is the actual level (plus 1) of column entry
01804 // j in row i, from summing the level contributions
01805 // s1 and s2 as explained in the methods section.
01806 // Note that the next line could reasonably be
01807 // replaced by, e.g., actlev = max(lmk, levels(m)),
01808 // but this would cause greater fill-in:
01809 // -----
01810 actlev = lmk + levels[m];
01811
01812 // -----
01813 // If lastcol of the current column entry in U is not
01814 // equal to the current row number i, then the current
01815 // row has a zero in column j, and the earlier row j
01816 // in U has a nonzero, so possible fill can occur.
01817 // -----
01818 if (lastcol[ijlum] != i) {
01819
01820 // -----
01821 // If actlev < levfill + 1, then the new entry has an
01822 // acceptable fill level and needs to be added to the
01823 // data structure.
01824 // -----
01825 if (actlev <= levpl) {
01826
01827 // -----
01828 // Since the column entry ijlum in the current
01829 // row i is to be filled, we need to update
01830 // lastcol for that column number. Also, the
01831 // level number of the current entry needs to be
01832 // set to actlev. Note that when we finish
01833 // processing this row, the n-vector levels(1:n)
01834 // will be copied over to the corresponding
01835 // trailing part of levels, so that it can be
01836 // used in subsequent rows:
01837 // -----
01838 lastcol[ijlum] = i;
01839 levels[ijlum] = actlev;
01840
01841 // -----
01842 // Now find location in the linked list rowll
01843 // where the fillin entry should be placed.
01844 // Chase through the linked list until the next
01845 // nonzero column is to the right of the fill
01846 // column number.
01847 // -----
01848 while (lm <= ijlum) { //50
01849     prev = lm;
01850     lm = rowll[lm];
01851 } //50
01852
01853 // -----
01854 // Insert new entry into the linked list for
01855 // row i, and increase the nonzero count for LU
01856 // -----
01857 rowll[prev] = ijlum;
01858 rowll[ijlum] = lm;
01859 prev = ijlum;
01860 *nzlu = *nzlu + 1;
01861 }
01862
01863 // -----
01864 // Else clause is for when lastcol(ijlum) = i. In
01865 // this case, the current column has a nonzero, but
01866 // it resulted from an earlier fill-in or from an
01867 // original matrix entry. In this case, need to
01868 // update the level number for this column to be the
01869 // smaller of the two possible fill contributors,

```

```

01870                                     // the current fill number or the computed one from
01871                                     // updating this entry from a previous row.
01872                                     // -----
01873     } else {
01874         levels[ijlum] = MIN(levels[ijlum], actlev);
01875     }
01876
01877                                     // -----
01878                                     // Now go and pick up the next column entry from row
01879                                     // j of U:
01880                                     // -----
01881
01882     } //60
01883     // -----
01884     // End if clause for levfill not equal to zero
01885     // -----
01886 }
01887
01888     // -----
01889     // Pick up next nonzero column index from the linked
01890     // list, and continue processing the i-th row's nonzeros.
01891     // This ends the first while loop (j < i).
01892     // -----
01893     j = rowll[j];
01894 } //80
01895
01896 // -----
01897 // Check to see if we have exceeded the allowed memory
01898 // storage before storing the results of computing row i's
01899 // sparsity pattern into the ijlu and uptr data structures.
01900 // -----
01901 if (*nzlu > nzmax) {
01902     printf("### ERROR: More storage needed! [%s]\n", __FUNCTION__);
01903     *ierr = 1;
01904     goto F100;
01905 }
01906
01907 // -----
01908 // Storage is adequate, so update ijlu data structure.
01909 // Row i ends at nzlu + 1:
01910 // -----
01911 ijlu[i + 1] = *nzlu + 1;
01912
01913 // -----
01914 // ... and the upper triangular part of LU begins at
01915 // lowct entries to right of where row i begins.
01916 // -----
01917 uptr[i] = ijlu[i] + lowct;
01918
01919 // -----
01920 // Now chase through linked list for row i, recording
01921 // information into ijlu. At same time, put level data
01922 // into the levels array for use on later rows:
01923 // -----
01924 j = head;
01925 k1 = ijlu[i];
01926 for (k = k1; k <= *nzlu; ++k) {
01927     ijlu[k] = j;
01928     levels[k] = levels[j];
01929     j = rowll[j];
01930 }
01931
01932 } else {
01933
01934     // -----
01935     // This else clause ends the (nzi > 1) if. If nzi = 1, then
01936     // the update of ijlu and uptr is trivial:
01937     // -----
01938     ijlu[i + 1] = *nzlu + 1;
01939     uptr[i] = ijlu[i];
01940 }
01941
01942 // -----
01943 // And you thought we would never get through....
01944 // -----
01945 } //100
01946
01947 if (cinindex) {
01948     for (i = 1; i <= *nzlu; ++i) --ijlu[i];
01949     for (i = 1; i <= n; ++i) --uptr[i];
01950     NE = rwptr[n + 1] - 1;

```

```

01951         for ( i = 1; i <= NE; ++i )    --colind[i];
01952         NE = n + 1;
01953         for ( i = 1; i <= NE; ++i )    --rwptr[i];
01954     }
01955
01956     *ierr = 0;
01957
01958 F100:
01959     ++rowll;
01960     ++lastcol;
01961     ++levels;
01962     ++colind;
01963     ++rwptr;
01964     ++ijlu;
01965     ++uptr;
01966
01967     fasp_mem_free(rowll);    rowll = NULL;
01968     fasp_mem_free(lastcol);  lastcol = NULL;
01969     fasp_mem_free(levels);   levels = NULL;
01970
01971 #if DEBUG_MODE > 0
01972     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01973 #endif
01974
01975     return;
01976     //===== End of symbfac =====
01977 }
01978
01979 /*-----*/
01980 /*--      Private Functions      --*/
01981 /*-----*/
01982
01983 static void fasp_qsplit (REAL    *a,
01984                        INT      *ind,
01985                        INT      n,
01986                        INT      ncut)
01987 {
01988     /*-----*/
01989     does a quick-sort split of a real array.
01990     on input a(1:n).  is a real array
01991     on output a(1:n) is permuted such that its elements satisfy:
01992     abs(a(i)) .ge. abs(a(ncut)) for i .lt. ncut and
01993     abs(a(i)) .le. abs(a(ncut)) for i .gt. ncut
01994     ind(1:n) is an integer array which permuted in the same way as a(*).
01995     -----*/
01996     REAL tmp, abskey;
01997     INT  itmp, first, last, mid, j;
01998
01999     /* Parameter adjustments */
02000     --ind;
02001     --a;
02002
02003     first = 1;
02004     last = n;
02005     if ((ncut < first) || (ncut > last)) return;
02006
02007     // outer loop -- while mid .ne. ncut do
02008 F161:
02009     mid = first;
02010     abskey = ABS(a[mid]);
02011     for (j = first + 1; j <= last; ++j) {
02012         if (ABS(a[j]) > abskey) {
02013             ++mid;
02014             // interchange
02015             tmp = a[mid];
02016             itmp = ind[mid];
02017             a[mid] = a[j];
02018             ind[mid] = ind[j];
02019             a[j] = tmp;
02020             ind[j] = itmp;
02021         }
02022     }
02023
02024     // interchange
02025     tmp = a[mid];
02026     a[mid] = a[first];
02027     a[first] = tmp;
02028     //
02029     itmp = ind[mid];
02030     ind[mid] = ind[first];
02031     ind[first] = itmp;
02032 }

```

```

02047
02048 // test for while loop
02049 if (mid == ncut) {
02050     ++ind;
02051     ++a;
02052     return;
02053 }
02054
02055 if (mid > ncut) {
02056     last = mid - 1;
02057 } else {
02058     first = mid + 1;
02059 }
02060
02061 goto F161;
02062 /*-----end-of-qsplrit-----*/
02063 }
02064
02077 static void fasp_sortrow (INT num,
02078                          INT *q)
02079 {
02080 #if DEBUG_MODE > 0
02081     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
02082 #endif
02119     INT key, icn, ih, ii, i, j, jj;
02120     INT iinc[6] = {0,1, 4, 13, 40, 121};
02121     //data iinc/1, 4, 13, 40, 121/;
02122
02123     --q;
02124     if (num == 0)
02125         icn = 0;
02126     else if (num < 14)
02127         icn = 1;
02128     else if (num < 41)
02129         icn = 2;
02130     else if (num < 122)
02131         icn = 3;
02132     else if (num < 365)
02133         icn = 4;
02134     else
02135         icn = 5;
02136
02137     for(ii = 1; ii <= icn; ++ii) { // 40
02138         ih = iinc[icn + 1 - ii];
02139         for(j = ih + 1; j <= num; ++j) { // 30
02140             i = j - ih;
02141             key = q[j];
02142             for(jj = 1; jj <= j - ih; jj += ih) { // 10
02143                 if (key >= q[i]) {
02144                     goto F20;
02145                 } else {
02146                     q[i + ih] = q[i];
02147                     i = i - ih;
02148                 }
02149             } // 10
02150             F20:
02151             q[i + ih] = key;
02152         } // 30
02153     } // 40
02154
02155     ++q;
02156
02157 #if DEBUG_MODE > 0
02158     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
02159 #endif
02160     return;
02161 }
02162
02175 static void fasp_check_col_index (INT row,
02176                                  INT num,
02177                                  INT *q)
02178 {
02179 #if DEBUG_MODE > 0
02180     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
02181 #endif
02182
02183     INT ii;
02184     INT num_1 = num - 1;
02185
02186     for ( ii = 0; ii < num_1; ++ii ) {
02187         if ( q[ii] == q[ii+1] ) {

```

```

02188         printf("### ERROR: Multiple entries with same col indices!\n");
02189         printf("### ERROR: row = %d, col = %d, %d!\n", row, q[ii], q[ii+1]);
02190         fasp_chkerr(ERROR_SOLVER_ILUSETUP, __FUNCTION__);
02191     }
02192 }
02193
02194 #if DEBUG_MODE > 0
02195     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
02196 #endif
02197
02198     return;
02199 }
02200
02201 /*-----*/
02202 /*--          End of File          --*/
02203 /*-----*/

```

9.53 BlalLUSetupBSR.c File Reference

Setup incomplete LU decomposition for [dBSRmat](#) matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_ilu_dbsr_setup](#) ([dBSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam)
Get ILU decoposition of a BSR matrix A.
- [SHORT fasp_ilu_dbsr_setup_step](#) ([dBSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam, [INT](#) step)
Get ILU decoposition of a BSR matrix A.
- [SHORT fasp_ilu_dbsr_setup_omp](#) ([dBSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam)
Multi-thread ILU decoposition of a BSR matrix A based on graph coloring.
- [SHORT fasp_ilu_dbsr_setup_levsch_omp](#) ([dBSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam)
Get ILU decoposition of a BSR matrix A based on level schedule strategy.
- [SHORT fasp_ilu_dbsr_setup_levsch_step](#) ([dBSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam, [INT](#) step)
Get ILU decoposition of a BSR matrix A based on level schedule strategy.
- [SHORT fasp_ilu_dbsr_setup_mc_omp](#) ([dBSRmat](#) *A, [dCSRmat](#) *Ap, [ILU_data](#) *iludata, [ILU_param](#) *iluparam)
Multi-thread ILU decoposition of a BSR matrix A based on graph coloring.
- void [topologic_sort_ILU](#) ([ILU_data](#) *iludata)
Reordering vertices according to level schedule strategy.
- void [mulcol_independ_set](#) ([AMG_data](#) *mgl, [INT](#) gslvl)
Multi-coloring vertices of adjacency graph of A.

9.53.1 Detailed Description

Setup incomplete LU decomposition for [dBSRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxTiming.c](#), [BlaSmallMatInv.c](#), [BlalLU.c](#), [BlaSmallMat.c](#), [BlaSmallMatInv.c](#), [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), and [PreDataInit.c](#)

Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlalLUSetupBSR.c](#).

9.53.2 Function Documentation

9.53.2.1 fasp_ilu_dbsr_setup()

```
SHORT fasp_ilu_dbsr_setup (
    dBSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam )
```

Get ILU decoposition of a BSR matrix A.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Shiquan Zhang, Xiaozhe Hu

Date

11/08/2010

Note

Works for general nb (Xiaozhe)

Change the size of work space by Zheng Li 04/26/2015.

Modified by Chunsheng Feng on 08/11/2017 for iludata->type not initied.

Definition at line 55 of file [BlalLUSetupBSR.c](#).

9.53.2.2 fasp_ilu_dbsr_setup_levsch_omp()

```
SHORT fasp_ilu_dbsr_setup_levsch_omp (
    dBSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam )
```

Get ILU decoposition of a BSR matrix A based on level schedule strategy.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Zheng Li

Date

12/04/2016

Note

Only works for nb = 1, 2, 3 (Zheng)

Modified by Chunsheng Feng on 09/06/2017 for iludata->type not inited

Definition at line 456 of file [BlalLUSetupBSR.c](#).

9.53.2.3 fasp_ilu_dbsr_setup_levsch_step()

```
SHORT fasp_ilu_dbsr_setup_levsch_step (
    dBSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam,
    INT step )
```

Get ILU decoposition of a BSR matrix A based on level schedule strategy.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param
<i>step</i>	Step in ILU factorization

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Zheng Li

Date

12/04/2016

Note

Only works for nb = 1, 2, 3 (Zheng)

Modified by Chunsheng Feng on 09/06/2017 for iludata->type not inited

Modified by Li Zhao on 04/29/2021: ILU factorization divided into two steps: step == 1: symbolic factoration; if step == 2: numerical factoration.

Definition at line 597 of file [BlalLUSetupBSR.c](#).

9.53.2.4 fasp_ilu_dbsr_setup_mc_omp()

```
SHORT fasp_ilu_dbsr_setup_mc_omp (
    dBSRmat * A,
    dCSRmat * Ap,
    ILU_data * iludata,
    ILU_param * iluparam )
```

Multi-thread ILU decoposition of a BSR matrix A based on graph coloring.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>Ap</i>	Pointer to dCSRmat matrix which provides sparsity pattern
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Zheng Li

Date

12/04/2016

Note

Only works for 1, 2, 3 nb (Zheng)

Modified by Chunsheng Feng on 09/06/2017 for iludata->type not inited.

Definition at line [745](#) of file [BlalLUSetupBSR.c](#).

9.53.2.5 fasp_ilu_dbsr_setup_omp()

```
SHORT fasp_ilu_dbsr_setup_omp (
    dBSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam )
```

Multi-thread ILU decoposition of a BSR matrix A based on graph coloring.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Zheng Li

Date

12/04/2016

Note

Only works for 1, 2, 3 nb (Zheng)

Modified by Chunsheng Feng on 09/06/2017 for iludata->type not inited.

Definition at line 320 of file [BlalLUSetupBSR.c](#).

9.53.2.6 fasp_ilu_dbsr_setup_step()

```
SHORT fasp_ilu_dbsr_setup_step (
    dBSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam,
    INT step )
```

Get ILU decoposition of a BSR matrix A.

Parameters

<i>A</i>	Pointer to dBSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param
<i>step</i>	Step in ILU factorization

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Shiquan Zhang, Xiaozhe Hu, Li Zhao

Date

11/08/2010

Note

Works for general nb (Xiaozhe)

Change the size of work space by Zheng Li 04/26/2015.

Modified by Chunsheng Feng on 08/11/2017 for iludata->type not inited.

Modified by Li Zhao on 04/29/2021: ILU factorization divided into two steps: step == 1: symbolic factoration; if step == 2: numerical factoration.

Definition at line 187 of file [BlalLUSetupBSR.c](#).

9.53.2.7 mulcol_independ_set()

```
void mulcol_independ_set (
    AMG_data * mgl,
    INT gslvl )
```

Multi-coloring vertices of adjacency graph of A.

Parameters

<i>mgl</i>	Pointer to input matrix
<i>gslvl</i>	Used to specify levels of AMG using multicolor smoothing

Author

Zheng Li, Chunsheng Feng

Date

12/04/2016

Definition at line 1909 of file [BlalLUSetupBSR.c](#).

9.53.2.8 topologic_sort_ILU()

```
void topologic_sort_ILU (
    ILU_data * iludata )
```

Reordering vertices according to level schedule strategy.

Parameters

<i>iludata</i>	Pointer to iludata
----------------	--------------------

Author

Zheng Li, Chensong Zhang

Date

12/04/2016

Definition at line 1827 of file [BlalLUSetupBSR.c](#).

9.54 BlalLUSetupBSR.c

[Go to the documentation of this file.](#)

```
00001
00016 #include <math.h>
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_funcs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 static INT numfactor (dBSRmat *, REAL *, INT *, INT *);
```

```

00027 static INT numfactor_mulcol (dBSRmat *, REAL *, INT *, INT *, INT, INT *, INT *);
00028 static INT numfactor_levsch (dBSRmat *, REAL *, INT *, INT *, INT, INT *, INT *);
00029 static void generate_S_theta(dCSRmat *, iCSRmat *, REAL);
00030 // static void topologic_sort_ILU (ILU_data *);
00031 // static void mulcol_independ_set (AMG_data *, INT);
00032
00033 /*-----*/
00034 /*--      Public Functions      --*/
00035 /*-----*/
00036
00055 SHORT fasp_ilu_dbsr_setup(dBSRmat *A,
00056                          ILU_data *iludata,
00057                          ILU_param *iluparam)
00058 {
00059
00060     const SHORT prtlvl = iluparam->print_level;
00061     const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;
00062
00063     // local variables
00064     INT    lfil = iluparam->ILU_lfil;
00065     INT    ierr, iwk, nzlu, nwork, *ijlu, *uptr;
00066     SHORT  status = FASP_SUCCESS;
00067     REAL    setup_start, setup_end, setup_duration;
00068
00069     #if DEBUG_MODE > 0
00070         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00071         printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
00072     #endif
00073
00074     fasp_gettime(&setup_start);
00075
00076     // Expected amount of memory for ILU needed and allocate memory
00077     iwk = (lfil+2)*nnz;
00078
00079     #if DEBUG_MODE > 0
00080         if (iluparam->ILU_type == ILUtp) {
00081             printf("### WARNING: iludata->type = %d not supported!\n",
00082                   iluparam->ILU_type);
00083         }
00084     #endif
00085
00086     // setup preconditioner
00087     iludata->type = 0; // Must be initialized
00088     iludata->iper = NULL;
00089     iludata->A = NULL; // No need for BSR matrix
00090     iludata->row = iludata->col = n;
00091     iludata->nb = nb;
00092     iludata->ilevL = iludata->jlevL = NULL;
00093     iludata->ilevU = iludata->jlevU = NULL;
00094
00095     ijlu = (INT*)fasp_mem_calloc(iwk, sizeof(INT));
00096     uptr = (INT*)fasp_mem_calloc(A->ROW, sizeof(INT));
00097
00098     #if DEBUG_MODE > 1
00099         printf("### DEBUG: symbolic factorization ... \n");
00100     #endif
00101
00102     // ILU decomposition
00103     // (1) symbolic factorization
00104     fasp_symbfactor(A->ROW, A->JA, A->IA, lfil, iwk, &nzlu, ijlu, uptr, &ierr);
00105
00106     if ( ierr != 0 ) {
00107         printf("### ERROR: ILU setup failed (ierr=%d)! [%s]\n", ierr, __FUNCTION__);
00108         status = ERROR_SOLVER_ILUSETUP;
00109         goto FINISHED;
00110     }
00111
00112     iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2, sizeof(REAL));
00113
00114     #if DEBUG_MODE > 1
00115         printf("### DEBUG: numerical factorization ... \n");
00116     #endif
00117
00118     // (2) numerical factorization
00119     status = numfactor(A, iludata->luval, ijlu, uptr);
00120
00121     if ( status < 0 ) {
00122         printf("### ERROR: ILU factorization failed! [%s]\n", __FUNCTION__);
00123         status = ERROR_SOLVER_ILUSETUP;
00124         goto FINISHED;
00125     }

```

```

00126
00127     //nwork = 6*nzlu*nb;
00128     nwork = 20*A->ROW*A->nb;
00129     iludata->nzlu = nzlu;
00130     iludata->nwork = nwork;
00131     iludata->ijlu = (INT*)fasp_mem_calloc(nzlu, sizeof(INT));
00132
00133     memcpy(iludata->ijlu, ijlu, nzlu*sizeof(INT));
00134     iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
00135     // Check: Is the work space too large? --Xiaozhe
00136
00137 #if DEBUG_MODE > 1
00138     printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
00139     printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
00140 #endif
00141
00142     if ( iwk < nzlu ) {
00143         printf("### ERROR: ILU needs more RAM %d! [%s]\n", iwk-nzlu, __FUNCTION__);
00144         status = ERROR_SOLVER_ILUSETUP;
00145         goto FINISHED;
00146     }
00147
00148     if ( prtlvl > PRINT_NONE ) {
00149         fasp_gettime(&setup_end);
00150         setup_duration = setup_end - setup_start;
00151         printf("BSR ILU(%d)-seq setup costs %f seconds.\n", lfil, setup_duration);
00152     }
00153
00154 FINISHED:
00155     fasp_mem_free(ijlu); ijlu = NULL;
00156     fasp_mem_free(uptr); uptr = NULL;
00157
00158 #if DEBUG_MODE > 0
00159     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00160 #endif
00161
00162     return status;
00163 }
00164
00187 SHORT fasp_ilu_dbsr_setup_step (dBSRmat      *A,
00188                                ILU_data     *iludata,
00189                                ILU_param    *iluparam,
00190                                INT step)
00191 {
00192
00193     const SHORT prtlvl = iluparam->print_level;
00194     const INT   n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;
00195
00196     // local variables
00197     INT lfil = iluparam->ILU_lfil;
00198     static INT ierr, iwk, nzlu, nwork, *ijlu, *uptr;
00199     SHORT status = FASP_SUCCESS;
00200
00201     REAL setup_start, setup_end, setup_duration;
00202
00203 #if DEBUG_MODE > 0
00204     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00205     printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
00206 #endif
00207
00208     fasp_gettime(&setup_start);
00209
00210     if (step==1) {
00211         // Expected amount of memory for ILU needed and allocate memory
00212         iwk = (lfil+2)*nnz;
00213
00214 #if DEBUG_MODE > 0
00215         if (iluparam->ILU_type == ILUtp) {
00216             printf("### WARNING: iludata->type = %d not supported!\n",
00217                   iluparam->ILU_type);
00218         }
00219 #endif
00220
00221         // setup preconditioner
00222         iludata->type = 0; // Must be initialized
00223         iludata->iperm = NULL;
00224         iludata->A = NULL; // No need for BSR matrix
00225         iludata->row = iludata->col = n;
00226         iludata->nb = nb;
00227         iludata->ilevL = iludata->jlevL = NULL;
00228         iludata->ilevU = iludata->jlevU = NULL;

```

```

00229
00230     ijlu = (INT*) fasp_mem_calloc(iwk, sizeof(INT));
00231
00232     if (uptr != NULL)     fasp_mem_free(uptr);
00233     uptr = (INT*) fasp_mem_calloc(A->ROW, sizeof(INT));
00234
00235 #if DEBUG_MODE > 1
00236     printf("### DEBUG: symbolic factorization ... \n");
00237 #endif
00238
00239     // ILU decomposition
00240     // (1) symbolic factoration
00241     fasp_symbfactor(A->ROW, A->JA, A->IA, lfil, iw, &nzlu, ijlu, uptr, &ierr);
00242
00243     iludata->luval = (REAL*) fasp_mem_calloc(nzlu*nb2, sizeof(REAL));
00244
00245
00246 #if DEBUG_MODE > 1
00247     printf("### DEBUG: numerical factorization ... \n");
00248 #endif
00249
00250     //nwork = 6*nzlu*nb;
00251     nwork = 5*A->ROW*A->nb;
00252     iludata->nwork = nwork;
00253     iludata->nzlu = nzlu;
00254     iludata->ijlu = (INT*) fasp_mem_calloc(nzlu, sizeof(INT));
00255
00256     memcpy(iludata->ijlu, ijlu, nzlu*sizeof(INT));
00257     fasp_mem_free(ijlu);   ijlu = NULL;
00258
00259     iludata->work = (REAL*) fasp_mem_calloc(nwork, sizeof(REAL));
00260     // Check: Is the work space too large? --Xiaozhe
00261
00262 #if DEBUG_MODE > 1
00263     printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
00264     printf("### DEBUG: iw = %d, nzlu = %d\n", iw, nzlu);
00265 #endif
00266
00267     if (ierr != 0) {
00268         printf("### ERROR: ILU setup failed (ierr=%d)! [%s]\n", ierr, __FUNCTION__);
00269         status = ERROR_SOLVER_ILUSETUP;
00270         goto FINISHED;
00271     }
00272
00273     if (iw < nzlu) {
00274         printf("### ERROR: ILU needs more RAM %d! [%s]\n", iw-nzlu, __FUNCTION__);
00275         status = ERROR_SOLVER_ILUSETUP;
00276         goto FINISHED;
00277     }
00278 }
00279 else if (step==2) {
00280     // (2) numerical factoration
00281     numfactor(A, iludata->luval, iludata->ijlu, uptr);
00282 } else {
00283
00284
00285 FINISHED:
00286     fasp_mem_free(uptr);   uptr = NULL;
00287 }
00288
00289 if (prtlvl > PRINT_NONE) {
00290     fasp_gettime(&setup_end);
00291     setup_duration = setup_end - setup_start;
00292     printf("BSR ILU(%d) setup costs %f seconds.\n", lfil, setup_duration);
00293 }
00294
00295 #if DEBUG_MODE > 0
00296     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00297 #endif
00298
00299     return status;
00300 }
00301
00302 SHORT fasp_ilu_dbsr_setup_omp (dBSRmat    *A,
00303                               ILU_data   *iludata,
00304                               ILU_param  *iluparam)
00305 {
00306
00307     const SHORT prtlvl = iluparam->print_level;
00308     const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;
00309
00310

```

```

00328     // local variables
00329     INT    lfil = iluparam->ILU_lfil;
00330     INT    ierr, iwk, nzlu, nwork, *ijlu, *uptr;
00331     SHORT  status = FASP_SUCCESS;
00332
00333     REAL    setup_start, setup_end, setup_duration;
00334     REAL    symbolic_start, symbolic_end, numfac_start, numfac_end;
00335
00336 #if DEBUG_MODE > 0
00337     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00338     printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
00339 #endif
00340
00341     fasp_gettime(&setup_start);
00342
00343     // Expected amount of memory for ILU needed and allocate memory
00344     iwk = (lfil+2)*nnz;
00345
00346 #if DEBUG_MODE > 0
00347     if (iluparam->ILU_type == ILUtp) {
00348         printf("### WARNING: iludata->type = %d not supported any more!\n",
00349             iluparam->ILU_type);
00350     }
00351 #endif
00352
00353     // setup preconditioner
00354     iludata->type = 0; // Must be initialized
00355     iludata->iperm = NULL;
00356     iludata->A = NULL; // No need for BSR matrix
00357     iludata->row = iludata->col = n;
00358     iludata->nb = nb;
00359
00360     ijlu = (INT *) fasp_mem_calloc(iwk, sizeof(INT));
00361     uptr = (INT *) fasp_mem_calloc(A->ROW, sizeof(INT));
00362
00363 #if DEBUG_MODE > 1
00364     printf("### DEBUG: symbolic factorization ... \n");
00365 #endif
00366
00367     // ILU decomposition
00368     // (1) symbolic factorization
00369     fasp_gettime(&symbolic_start);
00370
00371     fasp_symbfactor(A->ROW, A->JA, A->IA, lfil, iwk, &nzlu, ijlu, uptr, &ierr);
00372
00373     fasp_gettime(&symbolic_end);
00374
00375 #if prtlvl > PRINT_MIN
00376     printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
00377 #endif
00378
00379     nwork = 5*A->ROW*A->nb;
00380     iludata->nzlu = nzlu;
00381     iludata->nwork = nwork;
00382     iludata->ijlu = (INT*)fasp_mem_calloc(nzlu, sizeof(INT));
00383     iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2, sizeof(REAL));
00384     iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
00385     memcpy(iludata->ijlu, ijlu, nzlu*sizeof(INT));
00386     fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);
00387
00388 #if DEBUG_MODE > 1
00389     printf("### DEBUG: numerical factorization ... \n");
00390 #endif
00391
00392     // (2) numerical factorization
00393     fasp_gettime(&numfac_start);
00394
00395     numfactor_mulcol(A, iludata->luval, ijlu, uptr, iludata->nlevL,
00396         iludata->ilevL, iludata->jlevL);
00397
00398     fasp_gettime(&numfac_end);
00399
00400 #if prtlvl > PRINT_MIN
00401     printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
00402 #endif
00403
00404 #if DEBUG_MODE > 1
00405     printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
00406     printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
00407 #endif
00408

```



```

00409     if ( ierr != 0 ) {
00410         printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
00411         status = ERROR_SOLVER_ILUSETUP;
00412         goto FINISHED;
00413     }
00414
00415     if ( iwk < nzlu ) {
00416         printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
00417         status = ERROR_SOLVER_ILUSETUP;
00418         goto FINISHED;
00419     }
00420
00421     if ( prtlvl > PRINT_NONE ) {
00422         fasp_gettime(&setup_end);
00423         setup_duration = setup_end - setup_start;
00424         printf("BSR ILU(%d)-mc setup costs %f seconds.\n", lfil, setup_duration);
00425     }
00426
00427 FINISHED:
00428     fasp_mem_free(ijlu);  ijlu = NULL;
00429     fasp_mem_free(uptr);  uptr = NULL;
00430
00431     #if DEBUG_MODE > 0
00432         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00433     #endif
00434
00435     return status;
00436 }
00437
00456 SHORT fasp_ilu_dbsr_setup_levsch_omp (dBSRmat      *A,
00457                                       ILU_data      *iludata,
00458                                       ILU_param      *iluparam)
00459 {
00460     const SHORT prtlvl = iluparam->print_level;
00461     const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;
00462
00463     // local variables
00464     INT lfil = iluparam->ILU_lfil;
00465     INT ierr, iwk, nzlu, nwork, *ijlu, *uptr;
00466     SHORT status = FASP_SUCCESS;
00467
00468     REAL setup_start, setup_end, setup_duration;
00469     REAL symbolic_start, symbolic_end, numfac_start, numfac_end;
00470
00471     #if DEBUG_MODE > 0
00472         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00473         printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->ROW, n, nnz);
00474     #endif
00475
00476     fasp_gettime(&setup_start);
00477
00478     // Expected amount of memory for ILU needed and allocate memory
00479     iwk = (lfil+2)*nnz;
00480
00481     #if DEBUG_MODE > 0
00482         if (iluparam->ILU_type == ILUtp) {
00483             printf("### WARNING: iludata->type = %d not supported!\n",
00484                   iluparam->ILU_type);
00485         }
00486     #endif
00487
00488     // setup preconditioner
00489     iludata->type = 0; // Must be initialized
00490     iludata->iperm = NULL;
00491     iludata->A = NULL; // No need for BSR matrix
00492     iludata->row = iludata->col=n;
00493     iludata->nb = nb;
00494
00495     ijlu = (INT*)fasp_mem_calloc(iwk, sizeof(INT));
00496     uptr = (INT*)fasp_mem_calloc(A->ROW, sizeof(INT));
00497
00498     #if DEBUG_MODE > 1
00499         printf("### DEBUG: symbolic factorization ... \n");
00500     #endif
00501
00502     fasp_gettime(&symbolic_start);
00503
00504     // ILU decomposition
00505     // (1) symbolic factorization
00506     fasp_symbfactor(A->ROW, A->JA, A->IA, lfil, iwk, &nzlu, ijlu, uptr, &ierr);
00507

```

```

00508     fasp_gettime(&symbolic_end);
00509
00510 #if prtlvl > PRINT_MIN
00511     printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
00512 #endif
00513
00514     nwork = 5*A->ROW*A->nb;
00515     iludata->nzlu = nzlu;
00516     iludata->nwork = nwork;
00517     iludata->ijlu = (INT*)fasp_mem_calloc(nzlu,sizeof(INT));
00518     iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));
00519     iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
00520     memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
00521     fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);
00522     iludata->uptr = NULL; iludata->ic = NULL; iludata->icmap = NULL;
00523
00524     topologic_sort_ILU(iludata);
00525
00526 #if DEBUG_MODE > 1
00527     printf("### DEBUG: numerical factorization ... \n");
00528 #endif
00529
00530     fasp_gettime(&numfac_start);
00531
00532     // (2) numerical factoration
00533     numfactor_levsch(A, iludata->luval, ijlu, uptr, iludata->nlevL,
00534                     iludata->ilevL, iludata->jlevL);
00535
00536     fasp_gettime(&numfac_end);
00537
00538 #if prtlvl > PRINT_MIN
00539     printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
00540 #endif
00541
00542 #if DEBUG_MODE > 1
00543     printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
00544     printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
00545 #endif
00546
00547     if ( ierr != 0 ) {
00548         printf("### ERROR: ILU setup failed (ierr=%d)! [%s]\n", ierr, __FUNCTION__);
00549         status = ERROR_SOLVER_ILUSETUP;
00550         goto FINISHED;
00551     }
00552
00553     if ( iwk < nzlu ) {
00554         printf("### ERROR: ILU needs more RAM %d! [%s]\n", iwk-nzlu, __FUNCTION__);
00555         status = ERROR_SOLVER_ILUSETUP;
00556         goto FINISHED;
00557     }
00558
00559     if ( prtlvl > PRINT_NONE ) {
00560         fasp_gettime(&setup_end);
00561         setup_duration = setup_end - setup_start;
00562         printf("BSR ILU(%d)-ls setup costs %f seconds.\n", lfil, setup_duration);
00563     }
00564
00565 FINISHED:
00566     fasp_mem_free(ijlu); ijlu = NULL;
00567     fasp_mem_free(uptr); uptr = NULL;
00568
00569 #if DEBUG_MODE > 0
00570     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00571 #endif
00572
00573     return status;
00574 }
00575
00597 SHORT fasp_ilu_dbsr_setup_levsch_step (dBSRmat *A,
00598                                       ILU_data *iludata,
00599                                       ILU_param *iluparam,
00600                                       INT step)
00601 {
00602     const SHORT prtlvl = iluparam->print_level;
00603     const INT n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;
00604
00605     // local variables
00606     INT lfil = iluparam->ILU_lfil;
00607     static INT ierr, iwk, nzlu, nwork, *ijlu, *uptr;
00608     SHORT status = FASP_SUCCESS;
00609

```

```

00610     REAL    setup_start, setup_end, setup_duration;
00611     REAL    symbolic_start, symbolic_end, numfac_start, numfac_end;
00612
00613     #if DEBUG_MODE > 0
00614         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00615         printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->ROW, n, nnz);
00616         printf("### DEBUG: step=%d(1: symbolic factorization, 2: numerical factorization)\n", step); // zhaoli
00617     #endif
00618
00619     fasp_gettime(&setup_start);
00620     if (step==1) {
00621         // Expected amount of memory for ILU needed and allocate memory
00622         iwk = (lfil+2)*nnz;
00623
00624     #if DEBUG_MODE > 0
00625         if (iluparam->ILU_type == ILUtp) {
00626             printf("### WARNING: iludata->type = %d not supported!\n",
00627                 iluparam->ILU_type);
00628         }
00629     #endif
00630
00631         // setup preconditioner
00632         iludata->type = 0; // Must be initialized
00633         iludata->iperm = NULL;
00634         iludata->A = NULL; // No need for BSR matrix
00635         iludata->row = iludata->col=n;
00636         iludata->nb = nb;
00637
00638         fasp_mem_free(ijlu);
00639         ijlu = (INT*)fasp_mem_calloc(iwk, sizeof(INT));
00640
00641         fasp_mem_free(uptr);
00642         uptr = (INT*)fasp_mem_calloc(A->ROW, sizeof(INT));
00643
00644     #if DEBUG_MODE > 1
00645         printf("### DEBUG: symbolic factorization ... \n");
00646     #endif
00647
00648         fasp_gettime(&symbolic_start);
00649
00650         // ILU decomposition
00651         // (1) symbolic factorization
00652         fasp_symbfactor(A->ROW, A->JA, A->IA, lfil, iwk, &nzlu, ijlu, uptr, &ierr);
00653
00654         fasp_gettime(&symbolic_end);
00655
00656     #if prtlvl > PRINT_MIN
00657         printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
00658     #endif
00659
00660         nwork = 5*A->ROW*A->nb;
00661         iludata->nzlu = nzlu;
00662         iludata->nwork = nwork;
00663         iludata->ijlu = (INT*)fasp_mem_calloc(nzlu, sizeof(INT));
00664         iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2, sizeof(REAL));
00665         iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
00666         memcpy(iludata->ijlu, ijlu, nzlu*sizeof(INT));
00667         fasp_mem_free(ijlu); ijlu = NULL;
00668
00669         fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);
00670         iludata->upt = NULL; iludata->ic = NULL; iludata->icmap = NULL;
00671
00672         topologic_sort_ILU(iludata);
00673     #if DEBUG_MODE > 1
00674         printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
00675         printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
00676     #endif
00677
00678     if (ierr != 0) {
00679         printf("### ERROR: ILU setup failed (ierr=%d)! [%s]\n", ierr, __FUNCTION__);
00680         status = ERROR_SOLVER_ILUSETUP;
00681         goto FINISHED;
00682     }
00683
00684     if (iwk < nzlu) {
00685         printf("### ERROR: ILU needs more RAM %d! [%s]\n", iwk-nzlu, __FUNCTION__);
00686         status = ERROR_SOLVER_ILUSETUP;
00687         goto FINISHED;
00688     }
00689     } else if (step==2) {

```

```

00690
00691 #if DEBUG_MODE > 1
00692     printf("### DEBUG: numerical factorization ... \n");
00693 #endif
00694
00695     fasp_gettime(&numfac_start);
00696
00697     // (2) numerical factoration
00698     numfactor_levsch(A, iludata->luval, iludata->ijlu, uptr, iludata->nlevL,
00699                     iludata->ilevL, iludata->jlevL);
00700     fasp_gettime(&numfac_end);
00701
00702 #if prtlvl > PRINT_MIN
00703     printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
00704 #endif
00705 } else {
00706
00707 FINISHED:
00708 //     fasp_mem_free(ijlu); ijlu = NULL;
00709     fasp_mem_free(uptr); uptr = NULL;
00710 }
00711
00712 if ( prtlvl > PRINT_NONE ) {
00713     fasp_gettime(&setup_end);
00714     setup_duration = setup_end - setup_start;
00715     printf("BSR ILU(%d)-ls setup costs %f seconds.\n", lfil, setup_duration);
00716 }
00717
00718 #if DEBUG_MODE > 0
00719     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00720 #endif
00721
00722
00723     return status;
00724 }
00725
00745 SHORT fasp_ilu_dbsr_setup_mc_omp (dBSRmat      *A,
00746                                  dCSRmat      *Ap,
00747                                  ILU_data      *iludata,
00748                                  ILU_param     *iluparam)
00749 {
00750     INT status;
00751     AMG_data *mgl=fasp_amg_data_create(1);
00752     dCSRmat pp, Apl;
00753     dBSRmat A_LU;
00754
00755     if (iluparam->ILU_lfil==0) { //for ILU0
00756         mgl[0].A = fasp_dcsr_sympart(Ap);
00757     }
00758     else if (iluparam->ILU_lfil==1) { // for ILU1
00759         Apl = fasp_dcsr_create(Ap->row,Ap->col, Ap->nnz);
00760         fasp_dcsr_cp(Ap, &Apl);
00761         fasp_blas_dcsr_mxm (Ap,&Apl,&pp);
00762         mgl[0].A = fasp_dcsr_sympart(&pp);
00763         fasp_dcsr_free(&Apl);
00764         fasp_dcsr_free(&pp);
00765     }
00766
00767     mgl->num_levels = 20;
00768
00769     mulcol_independ_set(mgl, 1);
00770
00771     A_LU = fasp_dbsr_perm(A, mgl[0].icmap);
00772
00773     // hold color info with nlevL, ilevL and jlevL.
00774     iludata->nlevL = mgl[0].colors;
00775     iludata->ilevL = mgl[0].ic;
00776     iludata->jlevL = mgl[0].icmap;
00777     iludata->nlevU = 0;
00778     iludata->ilevU = NULL;
00779     iludata->jlevU = NULL;
00780     iludata->A      = NULL; // No need for BSR matrix
00781
00782 #if DEBUG_MODE > 0
00783     if (iluparam->ILU_type == ILUtp) {
00784         printf("### WARNING: iludata->type = %d not supported!\n",
00785             iluparam->ILU_type);
00786     }
00787 #endif
00788
00789     // setup preconditioner

```

```

00790     iludata->type = 0; // Must be initialized
00791     iludata->iperm = NULL;
00792
00793     status = fasp_ilu_dbsr_setup_omp(&A_LU, iludata, iluparam);
00794
00795     fasp_dcsr_free(&mgl[0].A);
00796     fasp_dbsr_free(&A_LU);
00797
00798     return status;
00799 }
00800
00801 /*-----*/
00802 /*--      Private Functions      --*/
00803 /*-----*/
00804
00819 static INT numfactor (dBSRmat *A,
00820                      REAL *luval,
00821                      INT *jlu,
00822                      INT *uptr)
00823 {
00824     INT n=A->ROW, nb=A->nb, nb2=nb*nb, ib, ibstart, ibstart1;
00825     INT k, indj, inds, indja, jluj, jlus, ija;
00826     REAL *mult, *mult1;
00827     INT *colptrs;
00828     INT status=FASP_SUCCESS;
00829
00830     colptrs=(INT*) fasp_mem_malloc(n, sizeof(INT));
00831     mult=(REAL*) fasp_mem_malloc(nb2, sizeof(REAL));
00832     mult1=(REAL*) fasp_mem_malloc(nb2, sizeof(REAL));
00833
00848     //for (k=0; k<n; k++) colptrs[k]=0;
00849     memset(colptrs, 0, sizeof(INT)*n);
00850
00851     switch (nb) {
00852
00853     case 1:
00854
00855         for (k = 0; k < n; ++k) {
00856
00857             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
00858                 colptrs[jlu[indj]] = indj;
00859                 ibstart=indj*nb2;
00860                 for (ib=0; ib<nb2; ++ib) luval[ibstart+ib] = 0;
00861             }
00862
00863             colptrs[k] = k;
00864
00865             for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
00866                 ija = A->JA[indja];
00867                 ibstart=colptrs[ija]*nb2;
00868                 ibstart1=indja*nb2;
00869                 for (ib=0; ib<nb2; ++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
00870             }
00871
00872             for (indj = jlu[k]; indj < uptr[k]; ++indj) {
00873
00874                 jluj = jlu[indj];
00875
00876                 luval[indj] = luval[indj]*luval[jluj];
00877                 mult[0] = luval[indj];
00878
00879                 for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
00880                     jlus = jlu[inds];
00881                     if (colptrs[jlus] != 0)
00882                         luval[colptrs[jlus]] = luval[colptrs[jlus]] - mult[0]*luval[inds];
00883                 }
00884             }
00885
00886             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
00887
00888             colptrs[k] = 0;
00889             luval[k] = 1.0/luval[k];
00890         }
00891     }
00892
00893     break;
00894
00895     case 3:
00896
00897         for (k = 0; k < n; ++k) {
00898

```

```

00899         for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
00900             colptrs[jlu[indj]] = indj;
00901             ibstart=indj*nb2;
00902             for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
00903         }
00904
00905         colptrs[k] = k;
00906
00907         for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
00908             ija = A->JA[indja];
00909             ibstart=colptrs[ija]*nb2;
00910             ibstart1=indja*nb2;
00911             for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
00912         }
00913
00914         for (indj = jlu[k]; indj < uptr[k]; ++indj) {
00915             jluj = jlu[indj];
00916
00917             ibstart=indj*nb2;
00918             fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
00919             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
00920
00921             for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
00922                 jlus = jlu[inds];
00923                 if (colptrs[jlus] != 0) {
00924                     fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
00925                     ibstart=colptrs[jlus]*nb2;
00926                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
00927                 }
00928             }
00929         }
00930
00931         for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
00932
00933         colptrs[k] = 0;
00934
00935         fasp_smat_inv_nc3(&(luval[k*nb2]));
00936     }
00937
00938     break;
00939
00940 case -5:
00941
00942     for (k = 0; k < n; ++k) {
00943
00944         for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
00945             colptrs[jlu[indj]] = indj;
00946             ibstart=indj*nb2;
00947             for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
00948         }
00949
00950         colptrs[k] = k;
00951
00952         for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
00953             ija = A->JA[indja];
00954             ibstart=colptrs[ija]*nb2;
00955             ibstart1=indja*nb2;
00956             for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
00957         }
00958
00959         for (indj = jlu[k]; indj < uptr[k]; ++indj) {
00960             jluj = jlu[indj];
00961
00962             ibstart=indj*nb2;
00963             fasp_blas_smat_mul_nc5(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
00964             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
00965
00966             for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
00967                 jlus = jlu[inds];
00968                 if (colptrs[jlus] != 0) {
00969                     fasp_blas_smat_mul_nc5(mult,&(luval[inds*nb2]),mult1);
00970                     ibstart=colptrs[jlus]*nb2;
00971                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
00972                 }
00973             }
00974         }
00975
00976     }
00977
00978     for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
00979

```

```

00980         colptrs[k] = 0;
00981
00982         // fasp_smat_inv_nc5(&(luval[k*nb2])); // not numerically stable --zcs 04/26/2021
00983         status = fasp_smat_inv_nc(&(luval[k*nb2]), 5);
00984     }
00985
00986     break;
00987
00988     case -7:
00989
00990         for (k = 0; k < n; ++k) {
00991
00992             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
00993                 colptrs[jlu[indj]] = indj;
00994                 ibstart=indj*nb2;
00995                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
00996             }
00997
00998             colptrs[k] = k;
00999
01000             for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01001                 ija = A->JA[indja];
01002                 ibstart=colptrs[ija]*nb2;
01003                 ibstart1=indja*nb2;
01004                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01005             }
01006
01007             for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01008                 jluj = jlu[indj];
01009
01010                 ibstart=indj*nb2;
01011                 fasp_blas_smat_mul_nc7(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
01012                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01013
01014                 for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01015                     jlus = jlu[inds];
01016                     if (colptrs[jlus] != 0) {
01017                         fasp_blas_smat_mul_nc7(mult,&(luval[inds*nb2]),mult1);
01018                         ibstart=colptrs[jlus]*nb2;
01019                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01020                     }
01021                 }
01022             }
01023
01024             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01025
01026             colptrs[k] = 0;
01027
01028             // fasp_smat_inv(&(luval[k*nb2]),nb); // not numerically stable --zcs 04/26/2021
01029             status = fasp_smat_inv_nc(&(luval[k*nb2]), nb);
01030
01031         }
01032
01033     break;
01034
01035     default:
01036
01037         for (k=0;k<n;k++) {
01038
01039             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01040                 colptrs[jlu[indj]] = indj;
01041                 ibstart=indj*nb2;
01042                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01043             }
01044
01045             colptrs[k] = k;
01046
01047             for (indja = A->IA[k]; indja < A->IA[k+1]; indja++) {
01048                 ija = A->JA[indja];
01049                 ibstart=colptrs[ija]*nb2;
01050                 ibstart1=indja*nb2;
01051                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01052             }
01053
01054             for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01055                 jluj = jlu[indj];
01056
01057                 ibstart=indj*nb2;
01058                 fasp_blas_smat_mul(&(luval[ibstart]),&(luval[jluj*nb2]),mult,nb);
01059                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01060

```

```

01061         for (inds = uptr[jluj]; inds < jlu[jluj+1]; inds++) {
01062             jlus = jlu[inds];
01063             if (colptrs[jlus] != 0) {
01064                 fasp_blas_smat_mul(mult, &(luval[inds*nb2]), mult1, nb);
01065                 ibstart=colptrs[jlus]*nb2;
01066                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01067             }
01068         }
01069     }
01070 }
01071
01072 for (indj = jlu[k]; indj < jlu[k+1]; ++indj)
01073     colptrs[jlu[indj]] = 0;
01074
01075 colptrs[k] = 0;
01076
01077 //fasp_smat_inv(&(luval[k*nb2]),nb); // not numerically stable --zcs 04/26/2021
01078 status = fasp_smat_invp_nc(&(luval[k * nb2]), nb);
01079 }
01080 }
01081
01082 fasp_mem_free(colptrs); colptrs = NULL;
01083 fasp_mem_free(mult); mult = NULL;
01084 fasp_mem_free(mult1); mult1 = NULL;
01085
01086 return status;
01087 }
01088
01107 static INT numfactor_mulcol (dBSRmat *A,
01108                             REAL *luval,
01109                             INT *jlu,
01110                             INT *uptr,
01111                             INT ncolors,
01112                             INT *ic,
01113                             INT *icmap)
01114 {
01115     INT status = FASP_SUCCESS;
01116
01117 #ifdef _OPENMP
01118     INT n = A->ROW, nb = A->nb, nb2 = nb*nb;
01119     INT ib, ibstart, ibstart1;
01120     INT k, i, indj, inds, indja, jluj, jlus, ija, tmp;
01121     REAL *mult, *mult1;
01122     INT *colptrs;
01123
01124     switch (nb) {
01125     case 1:
01126         for (i = 0; i < ncolors; ++i) {
01127             #pragma omp parallel private(k, indj, ibstart, ib, indja, ija, ibstart1, jluj, inds, jlus, colptrs, tmp)
01128             {
01129                 colptrs=(INT*)fasp_mem_malloc(n, sizeof(INT));
01130                 memset(colptrs, 0, sizeof(INT)*n);
01131                 #pragma omp for
01132                 for (k = ic[i]; k < ic[i+1]; ++k) {
01133                     for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01134                         colptrs[jlu[indj]] = indj;
01135                         ibstart=indj*nb2;
01136                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01137                     }
01138                     colptrs[k] = k;
01139                     for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01140                         ija = A->JA[indja];
01141                         ibstart=colptrs[ija]*nb2;
01142                         ibstart1=indja*nb2;
01143                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01144                     }
01145                     for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01146                         jluj = jlu[indj];
01147                         luval[indj] = luval[indj]*luval[jluj];
01148                         tmp = luval[indj];
01149                         for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01150                             jlus = jlu[inds];
01151                             if (colptrs[jlus] != 0)
01152                                 luval[colptrs[jlus]] = luval[colptrs[jlus]] - tmp*luval[inds];
01153                         }
01154                     }
01155                     for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01156                     colptrs[k] = 0;
01157                     luval[k] = 1.0/luval[k];
01158                 }
01159             }
01160         }
01161     }

```



```

01174         }
01175         fasp_mem_free(colptrs); colptrs = NULL;
01176     }
01177 }
01178
01179     break;
01180
01181     case 2:
01182
01183         for (i = 0; i < ncolors; ++i) {
01184 #pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs)
01185         {
01186             colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01187             memset(colptrs, 0, sizeof(INT)*n);
01188             mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01189             mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01190 #pragma omp for
01191             for (k = ic[i]; k < ic[i+1]; ++k) {
01192                 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01193                     colptrs[jlu[indj]] = indj;
01194                     ibstart=indj*nb2;
01195                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01196                 }
01197                 colptrs[k] = k;
01198                 for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01199                     ijaj = A->JA[indja];
01200                     ibstart=colptrs[ijaj]*nb2;
01201                     ibstart1=indja*nb2;
01202                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01203                 }
01204                 for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01205                     jluj = jlu[indj];
01206                     ibstart=indj*nb2;
01207                     fasp_blas_smat_mul_nc2(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
01208                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01209                     for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01210                         jlus = jlu[inds];
01211                         if (colptrs[jlus] != 0) {
01212                             fasp_blas_smat_mul_nc2(mult,&(luval[inds*nb2]),mult1);
01213                             ibstart=colptrs[jlus]*nb2;
01214                             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01215                         }
01216                     }
01217                 }
01218                 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01219                 colptrs[k] = 0;
01220                 fasp_smat_inv_nc2(&(luval[k*nb2]));
01221             }
01222             fasp_mem_free(colptrs); colptrs = NULL;
01223             fasp_mem_free(mult);      mult = NULL;
01224             fasp_mem_free(mult1);    mult1 = NULL;
01225         }
01226     }
01227     break;
01228
01229     case 3:
01230
01231         for (i = 0; i < ncolors; ++i) {
01232 #pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs)
01233         {
01234             colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01235             memset(colptrs, 0, sizeof(INT)*n);
01236             mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01237             mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01238 #pragma omp for
01239             for (k = ic[i]; k < ic[i+1]; ++k) {
01240                 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01241                     colptrs[jlu[indj]] = indj;
01242                     ibstart=indj*nb2;
01243                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01244                 }
01245                 colptrs[k] = k;
01246                 for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01247                     ijaj = A->JA[indja];
01248                     ibstart=colptrs[ijaj]*nb2;
01249                     ibstart1=indja*nb2;
01250                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01251                 }
01252                 for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01253                     jluj = jlu[indj];
01254                     ibstart=indj*nb2;

```

```

01255         fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
01256     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01257     for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01258         jlus = jlu[inds];
01259         if (colptrs[jlus] != 0) {
01260             fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
01261             ibstart=colptrs[jlus]*nb2;
01262             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01263         }
01264     }
01265 }
01266 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01267 colptrs[k] = 0;
01268 fasp_smat_inv_nc3(&(luval[k*nb2]));
01269 }
01270 fasp_mem_free(colptrs); colptrs = NULL;
01271 fasp_mem_free(mult); mult = NULL;
01272 fasp_mem_free(mult1); mult1 = NULL;
01273 }
01274 }
01275 break;
01276
01277 default:
01278 {
01279     if (nb > 3) printf("Multi-thread ILU numerical decomposition for %d\
01280 components has not been implemented!!!", nb);
01281     exit(0);
01282 }
01283 }
01284
01285 #endif
01286
01287 return status;
01288 }
01289
01309 static INT numfactor_levsch (dBSRmat *A,
01310                             REAL *luval,
01311                             INT *jlu,
01312                             INT *uptr,
01313                             INT ncolors,
01314                             INT *ic,
01315                             INT *icmap)
01316 {
01317     INT status = FASP_SUCCESS;
01318
01319 #ifdef _OPENMP
01320     INT n = A->ROW, nb = A->nb, nb2 = nb*nb;
01321     INT ib, ibstart,ibstart1;
01322     INT k, i, indj, inds, indja, jluj, jlus, ija, tmp, ii;
01323     REAL *mult, *mult1;
01324     INT *colptrs;
01325
01340     switch (nb) {
01341
01342     case 1:
01343         for (i = 0; i < ncolors; ++i) {
01344 #pragma omp parallel private(k,indj,ibstart,ib,indja,ija,ibstart1,jluj,inds,jlus,colptrs,tmp)
01345             {
01346                 colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01347                 memset(colptrs, 0, sizeof(INT)*n);
01348 #pragma omp for
01349                 for (k = ic[i]; k < ic[i+1]; ++k) {
01350                     for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01351                         colptrs[jlu[indj]] = indj;
01352                         ibstart=indj*nb2;
01353                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01354                     }
01355                     colptrs[k] = k;
01356                     for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01357                         ija = A->JA[indja];
01358                         ibstart=colptrs[ija]*nb2;
01359                         ibstart1=indja*nb2;
01360                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01361                     }
01362                     for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01363                         jluj = jlu[indj];
01364                         luval[indj] = luval[indj]*luval[jluj];
01365                         tmp = luval[indj];
01366                         for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01367                             jlus = jlu[inds];
01368                             if (colptrs[jlus] != 0)

```

```

01369             luval[colptrs[jlus]] = luval[colptrs[jlus]] - tmp*luval[inds];
01370         }
01371     }
01372     }
01373     for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01374     colptrs[k] = 0;
01375     luval[k] = 1.0/luval[k];
01376 }
01377 fasp_mem_free(colptrs); colptrs = NULL;
01378 }
01379 }
01380
01381 break;
01382 case 2:
01383
01384     for (i = 0; i < ncolors; ++i) {
01385 #pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
01386     {
01387         colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01388         memset(colptrs, 0, sizeof(INT)*n);
01389         mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01390         mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01391 #pragma omp for
01392         for (ii = ic[i]; ii < ic[i+1]; ++ii) {
01393             k = icmap[ii];
01394             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01395                 colptrs[jlu[indj]] = indj;
01396                 ibstart=indj*nb2;
01397                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01398             }
01399             colptrs[k] = k;
01400             for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01401                 ijaj = A->JA[indja];
01402                 ibstart=colptrs[ijaj]*nb2;
01403                 ibstart1=indja*nb2;
01404                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01405             }
01406             for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01407                 jluj = jlu[indj];
01408                 ibstart=indj*nb2;
01409                 fasp_blas_smat_mul_nc2(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
01410                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01411                 for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01412                     jlus = jlu[inds];
01413                     if (colptrs[jlus] != 0) {
01414                         fasp_blas_smat_mul_nc2(mult,&(luval[inds*nb2]),mult1);
01415                         ibstart=colptrs[jlus]*nb2;
01416                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01417                     }
01418                 }
01419             }
01420             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01421             colptrs[k] = 0;
01422             fasp_smat_inv_nc2(&(luval[k*nb2]));
01423         }
01424         fasp_mem_free(colptrs); colptrs = NULL;
01425         fasp_mem_free(mult); mult = NULL;
01426         fasp_mem_free(mult1); mult1 = NULL;
01427     }
01428 }
01429 break;
01430
01431 case 3:
01432
01433     for (i = 0; i < ncolors; ++i) {
01434 #pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
01435     {
01436         colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01437         memset(colptrs, 0, sizeof(INT)*n);
01438         mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01439         mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01440 #pragma omp for
01441         for (ii = ic[i]; ii < ic[i+1]; ++ii) {
01442             k = icmap[ii];
01443             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01444                 colptrs[jlu[indj]] = indj;
01445                 ibstart=indj*nb2;
01446                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01447             }
01448             colptrs[k] = k;
01449             for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {

```

```

01450             ija[j] = A->JA[indja];
01451             ibstart=colptrs[ija]*nb2;
01452             ibstart1=indja*nb2;
01453             for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01454         }
01455         for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01456             jlu[j] = jlu[indj];
01457             ibstart=indj*nb2;
01458             fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jlu*nb2]),mult);
01459             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01460             for (inds = uptr[jlu]; inds < jlu[jlu+1]; ++inds) {
01461                 jlus = jlu[inds];
01462                 if (colptrs[jlus] != 0) {
01463                     fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
01464                     ibstart=colptrs[jlus]*nb2;
01465                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01466                 }
01467             }
01468         }
01469         for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01470         colptrs[k] = 0;
01471         fasp_smat_inv_nc3(&(luval[k*nb2]));
01472     }
01473     fasp_mem_free(colptrs); colptrs = NULL;
01474     fasp_mem_free(mult); mult = NULL;
01475     fasp_mem_free(mult1); mult1 = NULL;
01476 }
01477 }
01478 break;
01479
01480 case 4:
01481     for (i = 0; i < ncolors; ++i) {
01482 #pragma omp parallel private(k,indj,ibstart,ib,indja,ija,j,ibstart1,jlu,inds,jlus,mult,mult1,colptrs,ii)
01483     {
01484         colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
01485         memset(colptrs, 0, sizeof(INT)*n);
01486         mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01487         mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
01488 #pragma omp for
01489         for (ii = ic[i]; ii < ic[i+1]; ++ii) {
01490             k = icmap[ii];
01491             for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01492                 colptrs[jlu[indj]] = indj;
01493                 ibstart=indj*nb2;
01494                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01495             }
01496             colptrs[k] = k;
01497             for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01498                 ija[j] = A->JA[indja];
01499                 ibstart=colptrs[ija]*nb2;
01500                 ibstart1=indja*nb2;
01501                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01502             }
01503             for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01504                 jlu[j] = jlu[indj];
01505                 ibstart=indj*nb2;
01506                 fasp_blas_smat_mul_nc4(&(luval[ibstart]),&(luval[jlu*nb2]),mult);
01507                 for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01508                 for (inds = uptr[jlu]; inds < jlu[jlu+1]; ++inds) {
01509                     jlus = jlu[inds];
01510                     if (colptrs[jlus] != 0) {
01511                         fasp_blas_smat_mul_nc4(mult,&(luval[inds*nb2]),mult1);
01512                         ibstart=colptrs[jlus]*nb2;
01513                         for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01514                     }
01515                 }
01516             }
01517         }
01518         for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01519         colptrs[k] = 0;
01520         fasp_smat_inv_nc4(&(luval[k*nb2]));
01521     }
01522     fasp_mem_free(colptrs); colptrs = NULL;
01523     fasp_mem_free(mult); mult = NULL;
01524     fasp_mem_free(mult1); mult1 = NULL;
01525 }
01526 }
01527 break;
01528
01529 case 5:
01530

```

```

01531         for (i = 0; i < ncolors; ++i) {
01532 #pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
01533         {
01534             colptrs=(INT*) fasp_mem_calloc(n,sizeof(INT));
01535             memset(colptrs, 0, sizeof(INT)*n);
01536             mult=(REAL*) fasp_mem_calloc(nb2,sizeof(REAL));
01537             mult1=(REAL*) fasp_mem_calloc(nb2,sizeof(REAL));
01538 #pragma omp for
01539             for (ii = ic[i]; ii < ic[i+1]; ++ii) {
01540                 k = icmap[ii];
01541                 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
01542                     colptrs[jlu[indj]] = indj;
01543                     ibstart=indj*nb2;
01544                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
01545                 }
01546                 colptrs[k] = k;
01547                 for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
01548                     ijaj = A->JA[indja];
01549                     ibstart=colptrs[ijaj]*nb2;
01550                     ibstart1=indja*nb2;
01551                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
01552                 }
01553                 for (indj = jlu[k]; indj < uptr[k]; ++indj) {
01554                     jluj = jlu[indj];
01555                     ibstart=indj*nb2;
01556                     fasp_blas_smat_mul_nc5(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
01557                     for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
01558                     for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
01559                         jlus = jlu[inds];
01560                         if (colptrs[jlus] != 0) {
01561                             fasp_blas_smat_mul_nc5(mult,&(luval[inds*nb2]),mult1);
01562                             ibstart=colptrs[jlus]*nb2;
01563                             for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
01564                         }
01565                     }
01566                 }
01567                 for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
01568                 colptrs[k] = 0;
01569                 fasp_smat_inv_nc5(&(luval[k*nb2]));
01570             }
01571             fasp_mem_free(colptrs); colptrs = NULL;
01572             fasp_mem_free(mult); mult = NULL;
01573             fasp_mem_free(mult1); mult1 = NULL;
01574         }
01575     }
01576     break;
01577
01578     default:
01579     {
01580         if (nb > 5) printf("Multi-thread ILU numerical decomposition for %d components has not been
01581 implemented!!\n", nb);
01582         exit(0);
01583         break;
01584     }
01585 }
01586 #endif
01587
01588     return status;
01589 }
01590
01603 static void generate_S_theta (dCSRmat *A,
01604                               iCSRmat *S,
01605                               REAL theta)
01606 {
01607     const INT row=A->row, col=A->col;
01608     const INT row_plus_one = row+1;
01609     const INT nnz=A->IA[row]-A->IA[0];
01610
01611     INT index, i, j, begin_row, end_row;
01612     INT *ia=A->IA, *ja=A->JA;
01613     REAL *aj=A->val;
01614
01615     // get the diagonal entry of A
01616     //dvector diag; fasp_dcsr_getdiag(0, A, &diag);
01617
01618     /* generate S */
01619     REAL row_abs_sum;
01620
01621     // copy the structure of A to S
01622     S->row=row; S->col=col; S->nnz=nnz; S->val=NULL;

```

```

01623
01624     S->IA=(INT*)fasp_mem_calloc(row_plus_one, sizeof(INT));
01625
01626     S->JA=(INT*)fasp_mem_calloc(nnz, sizeof(INT));
01627
01628     fasp_iarray_cp(row_plus_one, ia, S->IA);
01629     fasp_iarray_cp(nnz, ja, S->JA);
01630
01631     for (i=0;i<row;++i) {
01632         /* compute scaling factor and row sum */
01633         row_abs_sum=0;
01634
01635         begin_row=ia[i]; end_row=ia[i+1];
01636
01637         for (j=begin_row;j<end_row;j++) row_abs_sum+=ABS(a[j]);
01638
01639         row_abs_sum = row_abs_sum*theta;
01640
01641         /* deal with the diagonal element of S */
01642         // for (j=begin_row;j<end_row;j++) {
01643         //     if (ja[j]==i) {S->JA[j]=-1; break;}
01644         // }
01645
01646         /* deal with the element of S */
01647         for (j=begin_row;j<end_row;j++){
01648             /* if $sum_{j=1}^n |a_{ij}|*theta>= |a_{ij}|$ */
01649             if ( (row_abs_sum >= ABS(a[j])) && (ja[j] !=i) ) S->JA[j]=-1;
01650         }
01651     } // end for i
01652
01653     /* Compress the strength matrix */
01654     index=0;
01655     for (i=0;i<row;++i) {
01656         S->IA[i]=index;
01657         begin_row=ia[i]; end_row=ia[i+1]-1;
01658         for (j=begin_row;j<=end_row;j++) {
01659             if (S->JA[j]>-1) {
01660                 S->JA[index]=S->JA[j];
01661                 index++;
01662             }
01663         }
01664     }
01665
01666     if (index > 0) {
01667         S->IA[row]=index;
01668         S->nnz=index;
01669         S->JA=(INT*)fasp_mem_realloc(S->JA, index*sizeof(INT));
01670     }
01671     else {
01672         S->nnz = 0;
01673         S->JA = NULL;
01674     }
01675 }
01676
01691 static void multicoloring (AMG_data *mgl,
01692                             REAL      theta,
01693                             INT       *rowmax,
01694                             INT       *groups)
01695 {
01696     INT k, i, j, pre, group, iend;
01697     INT icount;
01698     INT front, rear;
01699     INT *IA, *JA;
01700     INT *cq, *newr;
01701
01702     const INT n = mgl->A.row;
01703     dCSRmat A = mgl->A;
01704     iCSRmat S;
01705
01706     S.IA = S.JA = NULL; S.val = NULL;
01707
01708     theta = MAX(0.0, MIN(1.0, theta));
01709
01710     if (theta > 0.0 && theta < 1.0) {
01711         generate_S_theta(&A, &S, theta);
01712         IA = S.IA;
01713         JA = S.JA;
01714     }
01715     else if (theta == 1.0) {
01716         mgl->ic = (INT*)malloc(sizeof(INT)*2);
01717

```

```

01718     mgl->icmap = (INT *)malloc(sizeof(INT)*(n+1));
01719     mgl->ic[0] = 0;
01720     mgl->ic[1] = n;
01721     for(k=0; k<n; k++) mgl->icmap[k]= k;
01722
01723     mgl->color = 1;
01724     *groups = 1;
01725     *rowmax = 1;
01726
01727     printf("### WARNING: Theta = %lf! [%s]\n", theta, __FUNCTION__);
01728
01729     return;
01730 }
01731 else {
01732     IA = A.IA;
01733     JA = A.JA;
01734 }
01735
01736 cq = (INT *)malloc(sizeof(INT)*(n+1));
01737 newr = (INT *)malloc(sizeof(INT)*(n+1));
01738
01739 #ifdef _OPENMP
01740 #pragma omp parallel for private(k)
01741 #endif
01742     for ( k=0; k<n; k++ ) cq[k]= k;
01743
01744     group = 0;
01745     for ( k=0; k<n; k++ ) {
01746         if ((A.IA[k+1] - A.IA[k]) > group ) group = A.IA[k+1] - A.IA[k];
01747     }
01748     *rowmax = group;
01749
01750     mgl->ic = (INT *)malloc(sizeof(INT)*(group+2));
01751     mgl->icmap = (INT *)malloc(sizeof(INT)*(n+1));
01752
01753     front = n-1;
01754     rear = n-1;
01755
01756     memset(newr, -1, sizeof(INT)*(n+1));
01757     memset(mgl->icmap, 0, sizeof(INT)*n);
01758
01759     group=0;
01760     icount = 0;
01761     mgl->ic[0] = 0;
01762     pre=0;
01763
01764     do {
01765         //front = (front+1)%n;
01766         front ++;
01767         if (front == n ) front =0; // front = front < n ? front : 0 ;
01768         i = cq[front];
01769
01770         if(i <= pre) {
01771             mgl->ic[group] = icount;
01772             mgl->icmap[icount] = i;
01773             group++;
01774             icount++;
01775         #if 0
01776             if ((IA[i+1]-IA[i]) > igold)
01777                 iend = MIN(IA[i+1], (IA[i] + igold));
01778             else
01779                 iend = IA[i+1];
01780             for (j= IA[i]; j< iend; j++) newr[JA[j]] = group;
01781         }
01782         else if (newr[i] == group) {
01783             //rear = (rear +1)%n;
01784             rear ++;
01785             if (rear == n) rear = 0;
01786             cq[rear] = i;
01787         }
01788         else {
01789             mgl->icmap[icount] = i;
01790             icount++;
01791         #if 0
01792             if ((IA[i+1] - IA[i]) > igold) iend =MIN(IA[i+1], (IA[i] + igold));
01793             else
01794                 iend = IA[i+1];
01795             for (j = IA[i]; j< iend; j++) newr[JA[j]] = group;
01796         }
01797     }
01798

```

```

01799     }
01800     pre=i;
01801
01802     } while(rear != front);
01803
01804     mgl->ic[group] = icount;
01805     mgl->colors = group;
01806     *groups = group;
01807
01808     free(cq);
01809     free(newr);
01810
01811     fasp_mem_free(S.IA); S.IA = NULL;
01812     fasp_mem_free(S.JA); S.JA = NULL;
01813
01814     return;
01815 }
01816
01827 void topologic_sort_ILU (ILU_data *iludata)
01828 {
01829     INT i, j, k, l;
01830     INT nlevL, nlevU;
01831
01832     INT n = iludata->row;
01833     INT *ijlu = iludata->ijlu;
01834
01835     INT *level = (INT *)fasp_mem_calloc(n, sizeof(INT));
01836     INT *jlevL = (INT *)fasp_mem_calloc(n, sizeof(INT));
01837     INT *ilevL = (INT *)fasp_mem_calloc(n+1, sizeof(INT));
01838
01839     INT *jlevU = (INT *)fasp_mem_calloc(n, sizeof(INT));
01840     INT *ilevU = (INT *)fasp_mem_calloc(n+1, sizeof(INT));
01841
01842     nlevL = 0;
01843     ilevL[0] = 0;
01844
01845     // form level for each row of lower triangular matrix.
01846     for (i=0; i<n; i++) {
01847         l = 0;
01848         for(j=ijlu[i]; j<ijlu[i+1]; j++) if (ijlu[j]<=i) l = MAX(l, level[ijlu[j]]);
01849         level[i] = l+1;
01850         ilevL[l+1] ++;
01851         nlevL = MAX(nlevL, l+1);
01852     }
01853
01854     for (i=1; i<=nlevL; i++) ilevL[i] += ilevL[i-1];
01855
01856     for (i=0; i<n; i++) {
01857         k = ilevL[level[i]-1];
01858         jlevL[k] = i;
01859         ilevL[level[i]-1]++;
01860     }
01861
01862     for (i=nlevL-1; i>0; i--) ilevL[i] = ilevL[i-1];
01863
01864     // form level for each row of upper triangular matrix.
01865     nlevU = 0;
01866     ilevL[0] = 0;
01867
01868     for (i=0; i<n; i++) level[i] = 0;
01869
01870     ilevU[0] = 0;
01871
01872     for (i=n-1; i>=0; i--) {
01873         l = 0;
01874         for (j=ijlu[i]; j<ijlu[i+1]; j++) if (ijlu[j]>=i) l = MAX(l, level[ijlu[j]]);
01875         level[i] = l+1;
01876         ilevU[l+1] ++;
01877         nlevU = MAX(nlevU, l+1);
01878     }
01879
01880     for (i=1; i<=nlevU; i++) ilevU[i] += ilevU[i-1];
01881
01882     for (i=n-1; i>=0; i--) {
01883         k = ilevU[level[i]-1];
01884         jlevU[k] = i;
01885         ilevU[level[i]-1]++;
01886     }
01887
01888     for (i=nlevU-1; i>0; i--) ilevU[i] = ilevU[i-1];
01889

```



```

01890     ilevU[0] = 0;
01891
01892     iludata->nlevL = nlevL+1; iludata->ilevL = ilevL; iludata->jlevL = jlevL;
01893     iludata->nlevU = nlevU+1; iludata->ilevU = ilevU; iludata->jlevU = jlevU;
01894
01895     fasp_mem_free(level); level = NULL;
01896 }
01897
01909 void mulcol_independ_set (AMG_data *mgl,
01910                          INT      gslvl)
01911 {
01912
01913     INT Colors, rowmax, level, prtlvl = 0;
01914
01915     REAL theta = 0.00;
01916
01917     INT maxlvl = MIN(gslvl, mgl->num_levels-1);
01918
01919 #ifdef _OPENMP
01920 #pragma omp parallel for private(level,rowmax,Colors) schedule(static, 1)
01921 #endif
01922     for ( level=0; level<maxlvl; level++ ) {
01923
01924         multicoloring(&mgl[level], theta, &rowmax, &Colors);
01925
01926         // print
01927         if ( prtlvl > PRINT_MIN )
01928             printf("mgl[%3d].A.row = %12d rowmax = %5d rowavg = %7.2lf colors = %5d theta = %1e\n",
01929                   level, mgl[level].A.row, rowmax, (double)mgl[level].A.nnz/mgl[level].A.row,
01930                   mgl[level].colors, theta);
01931     }
01932 }
01933
01934 /*-----*/
01935 /*--      End of File      --*/
01936 /*-----*/

```

9.55 BlalLUSetupCSR.c File Reference

Setup incomplete LU decomposition for [dCSRmat](#) matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_ilu_dcsr_setup](#) ([dCSRmat](#) *A, [ILU_data](#) *iludata, [ILU_param](#) *iluparam)
Get ILU decomposition of a CSR matrix A.

9.55.1 Detailed Description

Setup incomplete LU decomposition for [dCSRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxTiming.c](#), [BlalLU.c](#), [BlaSparseCSR.c](#), and [PreDataInit.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlalLUSetupCSR.c](#).

9.55.2 Function Documentation

9.55.2.1 fasp_ilu_dcsr_setup()

```
SHORT fasp_ilu_dcsr_setup (
    dCSRmat * A,
    ILU_data * iludata,
    ILU_param * iluparam )
```

Get ILU decomposition of a CSR matrix A.

Parameters

<i>A</i>	Pointer to dCSRmat matrix
<i>iludata</i>	Pointer to ILU_data
<i>iluparam</i>	Pointer to ILU_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Shiquan Zhang Xiaozhe Hu

Date

12/27/2009

Modified by Chunsheng Feng on 02/12/2017: add iperm array for ILUTp
Definition at line 40 of file [BlalLUSetupCSR.c](#).

9.56 BlalLUSetupCSR.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <math.h>
00015 #include <time.h>
00016
00017 #include "fasp.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/
00021 /*--      Public Functions      --*/
00022 /*-----*/
00023
00040 SHORT fasp_ilu_dcsr_setup (dCSRmat      *A,
00041                          ILU_data      *iludata,
00042                          ILU_param      *iluparam)
00043 {
00044     const INT    type = iluparam->ILU_type, print_level = iluparam->print_level;
00045     const INT    n = A->col, nnz = A->nnz, mbloc = n;
00046     const REAL   ILU_droptol = iluparam->ILU_droptol;
00047     const REAL   permtol = iluparam->ILU_permtol;
00048
00049     // local variable
00050     INT    lfil = iluparam->ILU_lfil, lfilt = iluparam->ILU_lfil;
00051     INT    ierr, iwk, nzlu, nwork, *ijlu, *iperm;
00052     REAL   *luval;
00053
00054     REAL   setup_start, setup_end, setup_duration;
00055     SHORT  status = FASP_SUCCESS;
```

```

00056
00057 #if DEBUG_MODE > 0
00058     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00059     printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->row, n, nnz);
00060 #endif
00061
00062     fasp_gettime(&setup_start);
00063
00064     // Expected amount of memory for ILU needed and allocate memory
00065     switch (type) {
00066     case ILUt:
00067         iwk=100*nnz;      // iwk is the maxim possible nnz for ILU
00068         lfilt=(int)floor(n*0.5)+1;
00069         break;
00070     case ILUtp:
00071         iwk=100*nnz;      // iwk is the maxim possible nnz for ILU
00072         lfilt=(int)floor(n*0.5)+1;
00073         break;
00074     default: // ILUk
00075         if (lfil == 0) iwk=nnz+500;
00076         else iwk=(lfil+5)*nnz;
00077     }
00078
00079     nwork = 4*n;
00080
00081 #if DEBUG_MODE > 1
00082     printf("### DEBUG: fill-in = %d, iwk = %d, nwork = %d\n", lfil, iwk, nwork);
00083 #endif
00084
00085     // setup ILU preconditioner
00086     iludata->A = A; // save a pointer to the coeff matrix for ILUtp
00087     iludata->row = iludata->col = n;
00088     iludata->ilevL = iludata->jlevL = NULL;
00089     iludata->ilevU = iludata->jlevU = NULL;
00090     iludata->iperm = NULL;
00091     iludata->type = type;
00092
00093     fasp_ilu_data_create(iwk, nwork, iludata);
00094
00095 #if DEBUG_MODE > 1
00096     printf("### DEBUG: memory usage after %s: \n", __FUNCTION__);
00097     fasp_mem_usage();
00098 #endif
00099
00100     // ILU decomposition
00101     ijlu = iludata->ijlu;
00102     luval = iludata->luval;
00103
00104     switch (type) {
00105
00106     case ILUt:
00107         fasp_ilut (n, A->val, A->JA, A->IA, lfilt, ILU_droptol, luval, ijlu,
00108                 iwk, &ierr, &nzlu);
00109         break;
00110
00111     case ILUtp:
00112         iperm = iludata->iperm;
00113         fasp_ilutp (n, A->val, A->JA, A->IA, lfilt, ILU_droptol, permtol,
00114                 mbloc, luval, ijlu, iperm, iwk, &ierr, &nzlu);
00115         break;
00116
00117     default: // ILUk
00118         fasp_iluk (n, A->val, A->JA, A->IA, lfilt, luval, ijlu, iwk,
00119                 &ierr, &nzlu);
00120         break;
00121     }
00122
00123     if (ierr != -4)
00124         fasp_dcsr_shift(A, -1);
00125
00126 #if DEBUG_MODE > 1
00127     printf("### DEBUG: memory usage after ILU setup: \n");
00128     fasp_mem_usage();
00129 #endif
00130
00131     iludata->nzlu = nzlu;
00132     iludata->nwork = nwork;
00133
00134 #if DEBUG_MODE > 1
00135     printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
00136 #endif

```

```

00137
00138     if (ierr!=0) {
00139         printf("### ERROR: ILU setup failed (ierr=%d)! [%s]\n", ierr, __FUNCTION__);
00140         status = ERROR_SOLVER_ILUSETUP;
00141         goto FINISHED;
00142     }
00143
00144     if (iwk<nzlu) {
00145         printf("### ERROR: ILU needs more RAM %d! [%s]\n", iwknzlu, __FUNCTION__);
00146         status = ERROR_SOLVER_ILUSETUP;
00147         goto FINISHED;
00148     }
00149
00150     if (print_level>PRINT_NONE) {
00151         fasp_gettime(&setup_end);
00152         setup_duration = setup_end - setup_start;
00153
00154         switch (type) {
00155             case ILUt:
00156                 printf("ILUt setup costs %f seconds.\n", setup_duration);
00157                 break;
00158             case ILUtp:
00159                 printf("ILUtp setup costs %f seconds.\n", setup_duration);
00160                 break;
00161             default: // ILUk
00162                 printf("ILUk setup costs %f seconds.\n", setup_duration);
00163                 break;
00164         }
00165     }
00166
00167 FINISHED:
00168
00169 #if DEBUG_MODE > 0
00170     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00171 #endif
00172
00173     return status;
00174 }
00175
00176 /*-----*/
00177 /*--      End of File      --*/
00178 /*-----*/

```

9.57 BlalLUSetupSTR.c File Reference

Setup incomplete LU decomposition for [dSTRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_ilu_dstr_setup0](#) ([dSTRmat](#) *A, [dSTRmat](#) *LU)
Get ILU(0) decomposition of a structured matrix A.
- void [fasp_ilu_dstr_setup1](#) ([dSTRmat](#) *A, [dSTRmat](#) *LU)
Get ILU(1) decomposition of a structured matrix A.

9.57.1 Detailed Description

Setup incomplete LU decomposition for [dSTRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#), [BlaSmallMat.c](#), [BlaSmallMatInv.c](#), [BlaSparseSTR.c](#), and [BlaArray.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlalLUSetupSTR.c](#).

9.57.2 Function Documentation

9.57.2.1 fasp_ilu_dstr_setup0()

```
void fasp_ilu_dstr_setup0 (  
    dSTRmat * A,  
    dSTRmat * LU )
```

Get ILU(0) decomposition of a structured matrix A.

Parameters

<i>A</i>	Pointer to dSTRmat
<i>LU</i>	Pointer to ILU structured matrix of REAL type

Author

Shiquan Zhang, Xiaozhe Hu

Date

11/08/2010

Note

Only works for 5 bands 2D and 7 bands 3D matrix with default offsets (order can be arbitrary)!

Definition at line 38 of file [BlalLUSetupSTR.c](#).

9.57.2.2 fasp_ilu_dstr_setup1()

```
void fasp_ilu_dstr_setup1 (  
    dSTRmat * A,  
    dSTRmat * LU )
```

Get ILU(1) decomposition of a structured matrix A.

Parameters

<i>A</i>	Pointer to original structured matrix of REAL type
<i>LU</i>	Pointer to ILU structured matrix of REAL type

Author

Shiquan Zhang, Xiaozhe Hu

Date

11/08/2010

Note

Put L and U in a STR matrix and it has the following structure: the diag is d, the offdiag of L are alpha1 to alpha6, the offdiag of U are beta1 to beta6

Only works for 5 bands 2D and 7 bands 3D matrix with default offsets

Definition at line 333 of file [BlalLUSetupSTR.c](#).

9.58 BlalLUSetupSTR.c

[Go to the documentation of this file.](#)

```

00001
00015 #include <math.h>
00016
00017 #include "fasp.h"
00018 #include "fasp_functs.h"
00019
00020 /*-----*/
00021 /*--      Public Functions      --*/
00022 /*-----*/
00023
00038 void fasp_ilu_dstr_setup0 (dSTRmat  *A,
00039                          dSTRmat  *LU)
00040 {
00041     // local variables
00042     INT i, il, ix, ixy, ii;
00043     INT *LUoffsets;
00044     INT nline, nplane;
00045
00046     // information of A
00047     INT nc = A->nc;
00048     INT nc2 = nc*nc;
00049     INT nx = A->nx;
00050     INT ny = A->ny;
00051     INT nz = A->nz;
00052     INT nxy = A->nxy;
00053     INT ngrid = A->ngrid;
00054     INT nband = A->nband;
00055
00056     INT *offsets = A->offsets;
00057     REAL *smat=(REAL *)fasp_mem_calloc(nc2, sizeof(REAL));
00058     REAL *diag = A->diag;
00059     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL;
00060     REAL *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;
00061
00062     // initialize
00063     if (nx == 1) {
00064         nline = ny;
00065         nplane = ngrid;
00066     }
00067     else if (ny == 1) {
00068         nline = nx;
00069         nplane = ngrid;
00070     }
00071     else if (nz == 1) {
00072         nline = nx;
00073         nplane = ngrid;
00074     }
00075     else {
00076         nline = nx;
00077         nplane = nxy;
00078     }
00079
00080     // check number of bands
00081     if (nband == 4) {
00082         LUoffsets=(INT *)fasp_mem_calloc(4, sizeof(INT));
00083         LUoffsets[0]=-1; LUoffsets[1]=1; LUoffsets[2]=-nline; LUoffsets[3]=nline;
00084     }
00085     else if (nband == 6) {
00086         LUoffsets=(INT *)fasp_mem_calloc(6, sizeof(INT));
00087         LUoffsets[0]=-1; LUoffsets[1]=1; LUoffsets[2]=-nline;
00088         LUoffsets[3]=nline; LUoffsets[4]=-nplane; LUoffsets[5]=nplane;
00089     }
00090     else {
00091         printf("%s:  number of bands for structured ILU is illegal!\n", __FUNCTION__);
00092         return;

```

```

00093     }
00094
00095     // allocate memory to store LU decomposition
00096     fasp_dstr_alloc(nx, ny, nz, nxy, ngrid, nband, nc, offsets, LU);
00097
00098     // copy diagonal
00099     memcpy(LU->diag, diag, (ngrid*nc2)*sizeof(REAL));
00100
00101     // check offsets and copy off-diagonals
00102     for (i=0; i<nband; ++i) {
00103         if (offsets[i] == -1) {
00104             offdiag0 = A->offdiag[i];
00105             memcpy(LU->offdiag[0], offdiag0, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00106         }
00107         else if (offsets[i] == 1) {
00108             offdiag1 = A->offdiag[i];
00109             memcpy(LU->offdiag[1], offdiag1, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00110         }
00111         else if (offsets[i] == -nline) {
00112             offdiag2 = A->offdiag[i];
00113             memcpy(LU->offdiag[2], offdiag2, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00114         }
00115         else if (offsets[i] == nline) {
00116             offdiag3 = A->offdiag[i];
00117             memcpy(LU->offdiag[3], offdiag3, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00118         }
00119         else if (offsets[i] == -nplane) {
00120             offdiag4 = A->offdiag[i];
00121             memcpy(LU->offdiag[4], offdiag4, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00122         }
00123         else if (offsets[i] == nplane) {
00124             offdiag5 = A->offdiag[i];
00125             memcpy(LU->offdiag[5], offdiag5, ((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
00126         }
00127         else {
00128             printf("### ERROR: Illegal offset for ILU! [%s]\n", __FUNCTION__);
00129             return;
00130         }
00131     }
00132
00133     // Setup
00134     if (nc == 1) {
00135
00136         LU->diag[0]=1.0/(LU->diag[0]);
00137
00138         for (i=1; i<ngrid; ++i) {
00139
00140             LU->offdiag[0][i-1]=(offdiag0[i-1])*(LU->diag[i-1]);
00141             if (i>=nline)
00142                 LU->offdiag[2][i-nline]=(offdiag2[i-nline])*(LU->diag[i-nline]);
00143             if (i>=nplane)
00144                 LU->offdiag[4][i-nplane]=(offdiag0[i-nplane])*(LU->diag[i-nplane]);
00145
00146             LU->diag[i]=diag[i]-(LU->offdiag[0][i-1])*(LU->offdiag[1][i-1]);
00147
00148             if (i>=nline)
00149                 LU->diag[i]=LU->diag[i]-(LU->offdiag[2][i-nline])*(LU->offdiag[3][i-nline]);
00150             if (i>=nplane)
00151                 LU->diag[i]=LU->diag[i]-(LU->offdiag[4][i-nplane])*(LU->offdiag[5][i-nplane]);
00152
00153             LU->diag[i]=1.0/(LU->diag[i]);
00154
00155         } // end for (i=1; i<ngrid; ++i)
00156
00157     } // end if (nc == 1)
00158
00159     else if (nc == 3) {
00160
00161         fasp_smat_inv_nc3(LU->diag);
00162
00163         for (i=1; i<ngrid; ++i) {
00164
00165             i1=(i-1)*9;
00166             ix=(i-nline)*9;
00167             ixy=(i-nplane)*9;
00168             ii=i*9;
00169
00170             fasp_blas_smat_mul_nc3(&(offdiag0[i1]), &(LU->diag[i1]), &(LU->offdiag[0][i1]));
00171
00172             if (i>=nline)
00173                 fasp_blas_smat_mul_nc3(&(offdiag2[ix]), &(LU->diag[ix]), &(LU->offdiag[2][ix]));

```

```

00174         if (i>=nplane)
00175             fasp_blas_smat_mul_nc3(&(offdiag4[ixy]), &(LU->diag[ixy]), &(LU->offdiag[4][ixy]));
00176
00177         fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1]), &(LU->offdiag[1][i1]), smat);
00178
00179         fasp_blas_darray_axpyz_nc3(-1, smat, &(diag[ii]), &(LU->diag[ii]));
00180
00181         if (i>=nline) {
00182             fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix]), &(LU->offdiag[3][ix]), smat);
00183             fasp_blas_darray_axpy_nc3(-1.0, smat, &(LU->diag[ii]));
00184         } //end if (i>=nline)
00185
00186         if (i>=nplane) {
00187             fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixy]), &(LU->offdiag[5][ixy]), smat);
00188             fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[ii]));
00189         } // end if (i>=nplane)
00190
00191         fasp_smat_inv_nc3(&(LU->diag[ii]));
00192     } // end for(i=1;i<A->ngrid;++i)
00193 } // end if (nc == 3)
00194
00195 else if (nc == 5) {
00196
00197     fasp_smat_inv_nc5(LU->diag);
00198
00199     for (i=1;i<ngrid;++i) {
00200
00201         i1=(i-1)*25;
00202         ix=(i-nline)*25;
00203         ixy=(i-nplane)*25;
00204         ii=i*25;
00205
00206         fasp_blas_smat_mul_nc5(&(offdiag0[i1]), &(LU->diag[i1]), &(LU->offdiag[0][i1]));
00207
00208         if (i>=nline)
00209             fasp_blas_smat_mul_nc5(&(offdiag2[ix]), &(LU->diag[ix]), &(LU->offdiag[2][ix]));
00210         if (i>=nplane)
00211             fasp_blas_smat_mul_nc5(&(offdiag4[ixy]), &(LU->diag[ixy]), &(LU->offdiag[4][ixy]));
00212
00213         fasp_blas_smat_mul_nc5(&(LU->offdiag[0][i1]), &(LU->offdiag[1][i1]), smat);
00214
00215         fasp_blas_darray_axpyz_nc5(-1.0, smat, &(diag[ii]), &(LU->diag[ii]));
00216
00217         if (i>=nline) {
00218             fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix]), &(LU->offdiag[3][ix]), smat);
00219             fasp_blas_darray_axpy_nc5(-1.0, smat, &(LU->diag[ii]));
00220         } //end if (i>=nline)
00221
00222         if (i>=nplane) {
00223             fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixy]), &(LU->offdiag[5][ixy]), smat);
00224             fasp_blas_darray_axpy_nc5(-1.0, smat, &(LU->diag[ii]));
00225         } // end if (i>=nplane)
00226
00227         fasp_smat_inv_nc5(&(LU->diag[ii]));
00228     } // end for(i=1;i<A->ngrid;++i)
00229 } // end if (nc == 5)
00230
00231 else if (nc == 7) {
00232
00233     fasp_smat_inv_nc7(LU->diag);
00234
00235     for (i=1;i<ngrid;++i) {
00236
00237         i1=(i-1)*49;
00238         ix=(i-nline)*49;
00239         ixy=(i-nplane)*49;
00240         ii=i*49;
00241
00242         fasp_blas_smat_mul_nc7(&(offdiag0[i1]), &(LU->diag[i1]), &(LU->offdiag[0][i1]));
00243
00244         if (i>=nline)
00245             fasp_blas_smat_mul_nc7(&(offdiag2[ix]), &(LU->diag[ix]), &(LU->offdiag[2][ix]));
00246         if (i>=nplane)
00247             fasp_blas_smat_mul_nc7(&(offdiag4[ixy]), &(LU->diag[ixy]), &(LU->offdiag[4][ixy]));
00248
00249         fasp_blas_smat_mul_nc7(&(LU->offdiag[0][i1]), &(LU->offdiag[1][i1]), smat);
00250     }
00251 }
00252
00253
00254

```



```

00255         fasp_blas_darray_axpyz_nc7(-1.0, smat, &(diag[ii]), &(LU->diag[ii]));
00256
00257     if (i>=nline) {
00258         fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ix]), &(LU->offdiag[3][ix]), smat);
00259         fasp_blas_darray_axpy_nc7(-1.0, smat, &(LU->diag[ii]));
00260     } //end if (i>=nline)
00261
00262     if (i>=nplane) {
00263         fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixy]), &(LU->offdiag[5][ixy]), smat);
00264         fasp_blas_darray_axpy_nc7(-1.0, smat, &(LU->diag[ii]));
00265     } // end if (i>=nplane)
00266
00267     fasp_smat_inv_nc7(&(LU->diag[ii]));
00268
00269 } // end for(i=1;i<A->ngrid;++i)
00270
00271 } // end if (nc == 7)
00272
00273 else {
00274
00275     fasp_smat_inv(LU->diag, nc);
00276
00277     for (i=1; i<ngrid; ++i) {
00278
00279         il=(i-1)*nc2;
00280         ix=(i-nline)*nc2;
00281         ixy=(i-nplane)*nc2;
00282         ii=i*nc2;
00283
00284         fasp_blas_smat_mul(&(offdiag0[il]), &(LU->diag[il]), &(LU->offdiag[0][il]), nc);
00285
00286         if (i>=nline)
00287             fasp_blas_smat_mul(&(offdiag2[ix]), &(LU->diag[ix]), &(LU->offdiag[2][ix]), nc);
00288         if (i>=nplane)
00289             fasp_blas_smat_mul(&(offdiag4[ixy]), &(LU->diag[ixy]), &(LU->offdiag[4][ixy]), nc);
00290
00291         fasp_blas_smat_mul(&(LU->offdiag[0][il]), &(LU->offdiag[1][il]), smat, nc);
00292
00293         fasp_blas_darray_axpyz(nc2, -1, smat, &(diag[ii]), &(LU->diag[ii]));
00294
00295         if (i>=nline) {
00296             fasp_blas_smat_mul(&(LU->offdiag[2][ix]), &(LU->offdiag[3][ix]), smat, nc);
00297             fasp_blas_darray_axpy(nc2, -1, smat, &(LU->diag[ii]));
00298         } //end if (i>=nline)
00299
00300         if (i>=nplane) {
00301             fasp_blas_smat_mul(&(LU->offdiag[4][ixy]), &(LU->offdiag[5][ixy]), smat, nc);
00302             fasp_blas_darray_axpy(nc2, -1, smat, &(LU->diag[ii]));
00303         } // end if (i>=nplane)
00304
00305         fasp_smat_inv(&(LU->diag[ii]), nc);
00306
00307     } // end for(i=1; i<A->ngrid; ++i)
00308
00309 }
00310
00311 fasp_mem_free(smat); smat = NULL;
00312
00313 return;
00314 }
00315
00333 void fasp_ilu_dstr_setup1 (dSTRmat *A,
00334                          dSTRmat *LU)
00335 {
00336     const INT LUnband = 12;
00337     INT LUoffsets[12];
00338
00339     const INT nc=A->nc, nc2=nc*nc;
00340     const INT nx=A->nx;
00341     const INT ny=A->ny;
00342     const INT nz=A->nz;
00343     const INT nxy=A->nxy;
00344     const INT nband=A->nband;
00345     const INT ngrid=A->ngrid;
00346     INT nline, nplane;
00347
00348     INT i, j, il, ix, ixy, ix1, ixyl, ic, ilc, ixc, ixlc, ixc, ixylc, ixyc, ixylc, ixyc;
00349     register REAL *smat, t, *tc;
00350
00351     if (nx == 1) {
00352         nline = ny;

```

```

00353     nplane = ngrid;
00354 }
00355 else if (ny == 1) {
00356     nline = nx;
00357     nplane = ngrid;
00358 }
00359 else if (nz == 1) {
00360     nline = nx;
00361     nplane = ngrid;
00362 }
00363 else {
00364     nline = nx;
00365     nplane = nxy;
00366 }
00367
00368 smat=(REAL *)fasp_mem_calloc(nc2,sizeof(REAL));
00369
00370 tc=(REAL *)fasp_mem_calloc(nc2,sizeof(REAL));
00371
00372 LUoffsets[0] = -1;
00373 LUoffsets[1] = 1;
00374 LUoffsets[2] = 1-nline;
00375 LUoffsets[3] = nline-1;
00376 LUoffsets[4] = -nline;
00377 LUoffsets[5] = nline;
00378 LUoffsets[6] = nline-nplane;
00379 LUoffsets[7] = nplane-nline;
00380 LUoffsets[8] = 1-nplane;
00381 LUoffsets[9] = nplane-1;
00382 LUoffsets[10] = -nplane;
00383 LUoffsets[11] = nplane;
00384
00385 fasp_dstr_alloc(nx,A->ny,A->nz,nxy,ngrid,LUnband,nc,LUoffsets,LU);
00386
00387 if (nband == 6) memcpy(LU->offdiag[11],A->offdiag[5],((ngrid-nxy)*nc2)*sizeof(REAL));
00388 memcpy(LU->diag,A->diag,nc2*sizeof(REAL));
00389
00390 if (nc == 1) {
00391     // compute the first row
00392     LU->diag[0]=1.0/(LU->diag[0]);
00393     LU->offdiag[1][0]=A->offdiag[1][0];
00394     LU->offdiag[5][0]=A->offdiag[3][0];
00395     LU->offdiag[3][0]=0;
00396     LU->offdiag[7][0]=0;
00397     LU->offdiag[9][0]=0;
00398
00399     for (i=1;i<ngrid;++i) {
00400
00401         il=i-1;ix=i-nline;ixy=i-nplane;ixl=ix+1;ixyx=ixy+nline;ixyl=ixy+1;
00402
00403         // comput alpha6[i-nxy]
00404         if (ixy>=0)
00405             LU->offdiag[10][ixy]=A->offdiag[4][ixy]*LU->diag[ixy];
00406
00407         // comput alpha5[ixyl]
00408         if (ixyl>=0) {
00409             t=0;
00410
00411             if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[1][ixy];
00412
00413             LU->offdiag[8][ixyl]=t*(LU->diag[ixyl]);
00414         }
00415
00416         // comput alpha4[ixyx]
00417         if (ixyx>=0) {
00418             t=0;
00419
00420             if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[5][ixy];
00421             if (ixyl>=0) t-=LU->offdiag[8][ixyl]*LU->offdiag[3][ixyl];
00422
00423             LU->offdiag[6][ixyx]=t*(LU->diag[ixyx]);
00424         }
00425
00426         // comput alpha3[ix]
00427         if (ix>=0) {
00428             t=A->offdiag[2][ix];
00429
00430             if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[7][ixy];
00431
00432             LU->offdiag[4][ix]=t*(LU->diag[ix]);
00433         }

```

```

00434
00435 // comput alpha2[i-nx+1]
00436 if (ix1>=0) {
00437     t=0;
00438
00439     if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[1][ix];
00440     if (ixy1>=0) t-=LU->offdiag[8][ixy1]*LU->offdiag[7][ixy1];
00441
00442     LU->offdiag[2][ix1]=t*(LU->diag[ix1]);
00443 }
00444
00445 // comput alpha1[i-1]
00446 t=A->offdiag[0][i1];
00447
00448 if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[3][ix];
00449 if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[9][ixy];
00450
00451 LU->offdiag[0][i1]=t*(LU->diag[i1]);
00452
00453 // comput beta1[i]
00454 if (i+1<ngrid) {
00455     t=A->offdiag[1][i];
00456
00457     if (ix1>=0) t-=LU->offdiag[2][ix1]*LU->offdiag[5][ix1];
00458     if (ixy1>=0) t-=LU->offdiag[8][ixy1]*LU->offdiag[11][ixy1];
00459
00460     LU->offdiag[1][i]=t;
00461 }
00462
00463 // comput beta2[i]
00464 if (i+nline-1<ngrid) {
00465     t=-LU->offdiag[0][i1]*LU->offdiag[5][i1];
00466
00467     if (ixyx>=0) t-=LU->offdiag[6][ixyx]*LU->offdiag[9][ixyx];
00468
00469     LU->offdiag[3][i]=t;
00470 }
00471
00472 // comput beta3[i]
00473 if (i+nline<ngrid) {
00474     t=A->offdiag[3][i];
00475
00476     if (ixyx>=0) t-=LU->offdiag[6][ixyx]*LU->offdiag[11][ixyx];
00477
00478     LU->offdiag[5][i]=t;
00479 }
00480
00481 // comput beta4[i]
00482 if (i+nplane-nline<ngrid) {
00483     t=0;
00484
00485     if (ix1>=0) t-=LU->offdiag[2][ix1]*LU->offdiag[9][ix1];
00486     if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[11][ix];
00487
00488     LU->offdiag[7][i]=t;
00489 }
00490
00491 // comput beta5[i]
00492 if (i+nplane-1<ngrid) LU->offdiag[9][i]=-LU->offdiag[0][i1]*LU->offdiag[11][i1];
00493
00494 // comput d[i]
00495 LU->diag[i]=A->diag[i]-(LU->offdiag[0][i1])*(LU->offdiag[1][i1]);
00496
00497 if (ix1>=0) LU->diag[i]-(LU->offdiag[2][ix1])*(LU->offdiag[3][ix1]);
00498 if (ix>=0) LU->diag[i]-(LU->offdiag[4][ix])*(LU->offdiag[5][ix]);
00499 if (ixyx>=0) LU->diag[i]-(LU->offdiag[6][ixyx])*(LU->offdiag[7][ixyx]);
00500 if (ixy1>=0) LU->diag[i]-(LU->offdiag[8][ixy1])*(LU->offdiag[9][ixy1]);
00501 if (ixy>=0) LU->diag[i]-(LU->offdiag[10][ixy])*(LU->offdiag[11][ixy]);
00502
00503 LU->diag[i]=1.0/(LU->diag[i]);
00504
00505 } // end for (i=1; i<ngrid; ++i)
00506
00507 } // end if (nc == 1)
00508
00509 else if (nc == 3) {
00510
00511     // comput the first row
00512     fasp_smat_inv_nc3(LU->diag);
00513     memcpy(LU->offdiag[1],A->offdiag[1],9*sizeof(REAL));
00514     memcpy(LU->offdiag[5],A->offdiag[3],9*sizeof(REAL));

```

```

00515
00516     for (i=1;i<ngrid;++i) {
00517         il=i-1;ix=i-nline;ixy=i-nplane;ixl=ix+1;ixyx=ixy+nline;ixyl=ixy+1;
00518         ic=i*nc2;ilc=il*nc2;ixc=ix*nc2;ixlc=ixl*nc2;ixyc=ixy*nc2;
00519         ixylc=ixyl*nc2;ixycx=ixyx*nc2;
00520
00521         // comput alpha6[i-nxy]
00522         if (ixy>=0)
00523             fasp_blas_smat_mul_nc3(&(A->offdiag[4][ixyc]),&(LU->diag[ixyc]),&(LU->offdiag[10][ixyc]));
00524
00525         // comput alpha5[ixyl]
00526         if (ixyl>=0) {
00527             for (j=0;j<9;++j) tc[j]=0;
00528
00529             if (ixy>=0) {
00530                 fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[1][ixyc]),smat);
00531                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00532             }
00533
00534             fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixylc]),&(LU->offdiag[8][ixylc]));
00535         }
00536
00537         // comput alpha4[ixyx]
00538         if (ixyx>=0) {
00539             for (j=0;j<9;++j) tc[j]=0;
00540
00541             if (ixy>=0) {
00542                 fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[5][ixyc]),smat);
00543                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00544             }
00545
00546             if (ixyl>=0) {
00547                 fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixylc]),&(LU->offdiag[3][ixylc]),smat);
00548                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00549             }
00550
00551             fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixycx]),&(LU->offdiag[6][ixycx]));
00552         }
00553
00554         // comput alpha3[ix]
00555         if (ix>=0) {
00556             memcpy(tc,&(A->offdiag[2][ixc]),9*sizeof(REAL));
00557
00558             if (ixy>=0) {
00559                 fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[7][ixyc]),smat);
00560                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00561             }
00562
00563             fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixc]),&(LU->offdiag[4][ixc]));
00564         }
00565
00566         // comput alpha2[i-nx+1]
00567         if (ixl>=0) {
00568             for (j=0;j<9;++j) tc[j]=0;
00569
00570             if (ix>=0) {
00571                 fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[1][ixc]),smat);
00572                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00573             }
00574
00575             if (ixyl>=0) {
00576                 fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixylc]),&(LU->offdiag[7][ixylc]),smat);
00577                 fasp_blas_darray_axpy_nc3(-1,smat,tc);
00578             }
00579
00580             fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixlc]),&(LU->offdiag[2][ixlc]));
00581         }
00582     } // end if (ixl >= 0)
00583
00584     // comput alpha1[i-1]
00585
00586     memcpy(tc,&(A->offdiag[0][ilc]),9*sizeof(REAL));
00587
00588     if (ix>=0) {
00589         fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[3][ixc]),smat);
00590         fasp_blas_darray_axpy_nc3(-1,smat,tc);
00591     }
00592
00593     if (ixy>=0) {
00594

```

```

00595         fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]), &(LU->offdiag[9][ixyc]), smat);
00596         fasp_blas_darray_axpy_nc3(-1, smat, tc);
00597     }
00598
00599     fasp_blas_smat_mul_nc3(tc, &(LU->diag[i1c]), &(LU->offdiag[0][i1c]));
00600
00601     // comput beta1[i]
00602     if (i+1<ngrid) {
00603
00604         memcpy(&(LU->offdiag[1][ic]), &(A->offdiag[1][ic]), 9*sizeof(REAL));
00605
00606         if (ix1>=0) {
00607             fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]), &(LU->offdiag[5][ix1c]), smat);
00608             fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[1][ic]));
00609         }
00610
00611         if (ixy1>=0) {
00612             fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixy1c]), &(LU->offdiag[11][ixy1c]), smat);
00613             fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[1][ic]));
00614         }
00615     }
00616
00617     // comput beta2[i]
00618     if (i+nline-1<ngrid) {
00619
00620         {
00621             fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]), &(LU->offdiag[5][i1c]), smat);
00622             fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[3][ic]));
00623         }
00624
00625         if (ixyx>=0) {
00626             fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[9][ixyxc]), smat);
00627             fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[3][ic]));
00628         }
00629     }
00630
00631 }
00632
00633 // comput beta3[i]
00634 if (i+nline<ngrid) {
00635
00636     memcpy(&(LU->offdiag[5][ic]), &(A->offdiag[3][ic]), 9*sizeof(REAL));
00637
00638     if (ixyx>=0) {
00639         fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[11][ixyxc]), smat);
00640         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[5][ic]));
00641     }
00642 }
00643
00644 // comput beta4[i]
00645 if (i+nplane-nline<ngrid) {
00646
00647     if (ix1>=0) {
00648         fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]), &(LU->offdiag[9][ix1c]), smat);
00649         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[7][ic]));
00650     }
00651
00652     if (ix>=0) {
00653         fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]), &(LU->offdiag[11][ixc]), smat);
00654         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[7][ic]));
00655     }
00656 }
00657
00658 // comput beta5[i]
00659 if (i+nplane-1<ngrid) {
00660
00661     fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]), &(LU->offdiag[11][i1c]), smat);
00662     fasp_blas_darray_axpy_nc3(-1, smat, &(LU->offdiag[9][ic]));
00663 }
00664
00665 // comput d[i]
00666 {
00667     fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]), &(LU->offdiag[1][i1c]), smat);
00668     fasp_blas_darray_axpy_nc3(-1, smat, &(A->diag[i1c]), &(LU->diag[i1c]));
00669 }
00670
00671 if (ix1>=0) {
00672     fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]), &(LU->offdiag[3][ix1c]), smat);
00673     fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[i1c]));
00674 }
00675

```

```

00676
00677     if (ix>=0) {
00678         fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]), &(LU->offdiag[5][ixc]), smat);
00679         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[ic]));
00680     }
00681
00682     if (ixyx>=0) {
00683         fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyc]), &(LU->offdiag[7][ixyc]), smat);
00684         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[ic]));
00685     }
00686
00687     if (ixyl>=0) {
00688         fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixylc]), &(LU->offdiag[9][ixylc]), smat);
00689         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[ic]));
00690     }
00691
00692     if (ixy>=0) {
00693         fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]), &(LU->offdiag[11][ixyc]), smat);
00694         fasp_blas_darray_axpy_nc3(-1, smat, &(LU->diag[ic]));
00695     }
00696
00697     fasp_smat_inv_nc3(&(LU->diag[ic]));
00698
00699 } // end for(i=1;i<ngrid;++i)
00700
00701 } // end if (nc == 3)
00702
00703 else if (nc == 5) {
00704     // compute the first row
00705     // fasp_smat_inv_nc5(LU->diag);
00706     fasp_smat_inv(LU->diag, 5);
00707     memcpy(LU->offdiag[1], A->offdiag[1], 25*sizeof(REAL));
00708     memcpy(LU->offdiag[5], A->offdiag[3], 25*sizeof(REAL));
00709
00710     for(i=1;i<ngrid;++i) {
00711         il=i-1; ix=i-nline; ixy=i-nplane; ixl=ix+1; ixyx=ixy+nxline; ixyl=ixy+1;
00712         ic=i*nc2; ilc=il*nc2; ixc=ix*nc2; ixlc=ixl*nc2; ixyz=ixy*nc2; ixylc=ixyl*nc2; ixyxc=ixyx*nc2;
00713
00714         // compute alpha6[i-nxy]
00715         if (ixy>=0)
00716             fasp_blas_smat_mul_nc5(&(A->offdiag[4][ixyc]), &(LU->diag[ixyc]), &(LU->offdiag[10][ixyc]));
00717
00718         // compute alpha5[ixyl]
00719         if (ixyl>=0) {
00720             for (j=0; j<25; ++j) tc[j]=0;
00721
00722             if (ixy>=0) {
00723                 fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]), &(LU->offdiag[1][ixyc]), smat);
00724                 fasp_blas_darray_axpy_nc5(-1.0, smat, tc);
00725             }
00726
00727             fasp_blas_smat_mul_nc5(tc, &(LU->diag[ixylc]), &(LU->offdiag[8][ixylc]));
00728
00729             // compute alpha4[ixyx]
00730             if (ixyx>=0) {
00731                 for (j=0; j<25; ++j) tc[j]=0;
00732
00733                 if (ixy>=0) {
00734                     fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]), &(LU->offdiag[5][ixyc]), smat);
00735                     fasp_blas_darray_axpy_nc5(-1, smat, tc);
00736                 }
00737
00738                 if (ixyl>=0) {
00739                     fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixylc]), &(LU->offdiag[3][ixylc]), smat);
00740                     fasp_blas_darray_axpy_nc5(-1, smat, tc);
00741                 }
00742
00743                 fasp_blas_smat_mul_nc5(tc, &(LU->diag[ixyc]), &(LU->offdiag[6][ixyc]));
00744             }
00745
00746             // compute alpha3[ix]
00747             if (ix>=0) {
00748                 memcpy(tc, &(A->offdiag[2][ixc]), 25*sizeof(REAL));
00749
00750                 if (ixy>=0) {
00751                     fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]), &(LU->offdiag[7][ixyc]), smat);
00752                     fasp_blas_darray_axpy_nc5(-1, smat, tc);
00753                 }
00754             }
00755

```

```

00756         fasp_blas_smat_mul_nc5(tc, &(LU->diag[ixc]), &(LU->offdiag[4][ixc]));
00757     }
00758
00759     // comput alpha2[i-nx+1]
00760     if (ix1>=0) {
00761
00762         for (j=0; j<25; ++j) tc[j]=0;
00763
00764         if (ix>=0) {
00765             fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]), &(LU->offdiag[1][ixc]), smat);
00766             fasp_blas_darray_axpy_nc5(-1, smat, tc);
00767         }
00768
00769         if (ixy1>=0) {
00770             fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]), &(LU->offdiag[7][ixy1c]), smat);
00771             fasp_blas_darray_axpy_nc5(-1, smat, tc);
00772         }
00773
00774         fasp_blas_smat_mul_nc5(tc, &(LU->diag[ix1c]), &(LU->offdiag[2][ix1c]));
00775
00776     } // end if (ix1 >= 0)
00777
00778     // comput alphas[i-1]
00779
00780     memcpy(tc, &(A->offdiag[0][ilc]), 25*sizeof(REAL));
00781
00782     if (ix>=0) {
00783         fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]), &(LU->offdiag[3][ixc]), smat);
00784         fasp_blas_darray_axpy_nc5(-1, smat, tc);
00785     }
00786
00787     if (ixy>=0) {
00788         fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]), &(LU->offdiag[9][ixyc]), smat);
00789         fasp_blas_darray_axpy_nc5(-1, smat, tc);
00790     }
00791
00792     fasp_blas_smat_mul_nc5(tc, &(LU->diag[ilc]), &(LU->offdiag[0][ilc]));
00793
00794     // comput beta1[i]
00795     if (i+1<ngrid) {
00796
00797         memcpy(&(LU->offdiag[1][ic]), &(A->offdiag[1][ic]), 25*sizeof(REAL));
00798
00799         if (ix1>=0) {
00800             fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]), &(LU->offdiag[5][ix1c]), smat);
00801             fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[1][ic]));
00802         }
00803
00804         if (ixy1>=0) {
00805             fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]), &(LU->offdiag[11][ixy1c]), smat);
00806             fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[1][ic]));
00807         }
00808
00809     }
00810
00811     // comput beta2[i]
00812     if (i+nlne-1<ngrid) {
00813
00814         {
00815             fasp_blas_smat_mul_nc5(&(LU->offdiag[0][ilc]), &(LU->offdiag[5][ilc]), smat);
00816             fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[3][ic]));
00817         }
00818
00819         if (ixyx>=0) {
00820             fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[9][ixyxc]), smat);
00821             fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[3][ic]));
00822         }
00823
00824     }
00825
00826     // comput beta3[i]
00827     if (i+nlne<ngrid) {
00828
00829         memcpy(&(LU->offdiag[5][ic]), &(A->offdiag[3][ic]), 25*sizeof(REAL));
00830
00831         if (ixyx>=0) {
00832             fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[11][ixyxc]), smat);
00833             fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[5][ic]));
00834         }
00835
00836     }

```

```

00837
00838 // comput beta4[i]
00839 if (i+nplane-nline<ngrid) {
00840
00841     if (ix1>=0) {
00842         fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]), &(LU->offdiag[9][ix1c]), smat);
00843         fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[7][ic]));
00844     }
00845
00846     if (ix>=0) {
00847         fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]), &(LU->offdiag[11][ixc]), smat);
00848         fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[7][ic]));
00849     }
00850
00851 }
00852
00853 // comput beta5[i]
00854 if (i+nplane-1<ngrid) {
00855     fasp_blas_smat_mul_nc5(&(LU->offdiag[0][ilc]), &(LU->offdiag[11][ilc]), smat);
00856     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->offdiag[9][ic]));
00857 }
00858
00859 // comput d[i]
00860 {
00861     fasp_blas_smat_mul_nc5(&(LU->offdiag[0][ilc]), &(LU->offdiag[1][ilc]), smat);
00862     fasp_blas_darray_axpy_nc5(-1, smat, &(A->diag[ic]), &(LU->diag[ic]));
00863 }
00864
00865 if (ix1>=0) {
00866     fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]), &(LU->offdiag[3][ix1c]), smat);
00867     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->diag[ic]));
00868 }
00869
00870 if (ix>=0) {
00871     fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]), &(LU->offdiag[5][ixc]), smat);
00872     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->diag[ic]));
00873 }
00874
00875 if (ixyx>=0) {
00876     fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyc]), &(LU->offdiag[7][ixyc]), smat);
00877     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->diag[ic]));
00878 }
00879
00880 if (ixy1>=0) {
00881     fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]), &(LU->offdiag[9][ixy1c]), smat);
00882     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->diag[ic]));
00883 }
00884
00885 if (ixy>=0) {
00886     fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]), &(LU->offdiag[11][ixyc]), smat);
00887     fasp_blas_darray_axpy_nc5(-1, smat, &(LU->diag[ic]));
00888 }
00889
00890 //fasp_smat_inv_nc5(&(LU->diag[ic]));
00891 fasp_smat_inv(&(LU->diag[ic]), 5);
00892
00893 } // end for(i=1;i<ngrid;++i)
00894
00895 } // end if (nc == 5)
00896
00897 else if (nc == 7) {
00898     // comput the first row
00899     //fasp_smat_inv_nc5(LU->diag);
00900     fasp_smat_inv(LU->diag, 7);
00901     memcpy(LU->offdiag[1], A->offdiag[1], 49*sizeof(REAL));
00902     memcpy(LU->offdiag[5], A->offdiag[3], 49*sizeof(REAL));
00903
00904     for(i=1;i<ngrid;++i) {
00905         il=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;
00906         ic=i*nc2;ilc=il*nc2;ixc=ix*nc2;ix1c=ix1*nc2;ixyc=ixy*nc2;ixy1c=ixy1*nc2;ixycx=ixyx*nc2;
00907
00908         // comput alpha6[i-nxy]
00909         if (ixy>=0)
00910             fasp_blas_smat_mul_nc7(&(A->offdiag[4][ixyc]), &(LU->diag[ixyc]), &(LU->offdiag[10][ixyc]));
00911
00912         // comput alpha5[ixy1]
00913         if (ixy1>=0) {
00914             for (j=0; j<49; ++j) tc[j]=0;
00915
00916             if (ixy>=0) {
00917                 fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]), &(LU->offdiag[1][ixyc]), smat);

```



```

00917         fasp_blas_darray_axpy_nc7(-1.0, smat, tc);
00918     }
00919
00920     fasp_blas_smat_mul_nc7(tc, &(LU->diag[ixylc]), &(LU->offdiag[8][ixylc]));
00921 }
00922
00923 // comput alpha4[ixyx]
00924 if (ixyx>=0) {
00925     for (j=0; j<49; ++j) tc[j]=0;
00926
00927     if (ixy>=0) {
00928         fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]), &(LU->offdiag[5][ixyc]), smat);
00929         fasp_blas_darray_axpy_nc7(-1, smat, tc);
00930     }
00931
00932     if (ixyl>=0) {
00933         fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixylc]), &(LU->offdiag[3][ixylc]), smat);
00934         fasp_blas_darray_axpy_nc7(-1, smat, tc);
00935     }
00936
00937     fasp_blas_smat_mul_nc7(tc, &(LU->diag[ixyc]), &(LU->offdiag[6][ixyc]));
00938 }
00939
00940 // comput alpha3[ix]
00941 if (ix>=0) {
00942     memcpy(tc, &(A->offdiag[2][ixc]), 49*sizeof(REAL));
00943
00944     if (ixy>=0) {
00945         fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]), &(LU->offdiag[7][ixyc]), smat);
00946         fasp_blas_darray_axpy_nc7(-1, smat, tc);
00947     }
00948
00949     fasp_blas_smat_mul_nc7(tc, &(LU->diag[ixc]), &(LU->offdiag[4][ixc]));
00950 }
00951
00952 // comput alpha2[i-nx+1]
00953 if (ixl>=0) {
00954     for (j=0; j<49; ++j) tc[j]=0;
00955
00956     if (ix>=0) {
00957         fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]), &(LU->offdiag[1][ixc]), smat);
00958         fasp_blas_darray_axpy_nc7(-1, smat, tc);
00959     }
00960
00961     if (ixyl>=0) {
00962         fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixylc]), &(LU->offdiag[7][ixylc]), smat);
00963         fasp_blas_darray_axpy_nc7(-1, smat, tc);
00964     }
00965
00966     fasp_blas_smat_mul_nc7(tc, &(LU->diag[ixlc]), &(LU->offdiag[2][ixlc]));
00967 } // end if (ixl >= 0)
00968
00969 // comput alphas[i-1]
00970
00971 memcpy(tc, &(A->offdiag[0][ilc]), 49*sizeof(REAL));
00972
00973 if (ix>=0) {
00974     fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]), &(LU->offdiag[3][ixc]), smat);
00975     fasp_blas_darray_axpy_nc7(-1, smat, tc);
00976 }
00977
00978 if (ixy>=0) {
00979     fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]), &(LU->offdiag[9][ixyc]), smat);
00980     fasp_blas_darray_axpy_nc7(-1, smat, tc);
00981 }
00982
00983 fasp_blas_smat_mul_nc7(tc, &(LU->diag[ilc]), &(LU->offdiag[0][ilc]));
00984
00985 // comput beta1[i]
00986 if (i+1<ngrid) {
00987     memcpy(&(LU->offdiag[1][ic]), &(A->offdiag[1][ic]), 49*sizeof(REAL));
00988
00989     if (ixl>=0) {
00990         fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ixlc]), &(LU->offdiag[5][ixlc]), smat);
00991         fasp_blas_darray_axpy_nc7(-1, smat, &(LU->offdiag[1][ic]));
00992     }
00993 }
00994

```

```

00998         if (ixyl>=0) {
00999             fasp_blas_smat_mul_nc7 (&(LU->offdiag[8][ixylc]), &(LU->offdiag[11][ixylc]), smat);
01000             fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[1][ic]));
01001         }
01002     }
01003 }
01004
01005 // comput beta2[i]
01006 if (i+nline-1<ngrid) {
01007     {
01008         fasp_blas_smat_mul_nc7 (&(LU->offdiag[0][ilc]), &(LU->offdiag[5][ilc]), smat);
01009         fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[3][ic]));
01010     }
01011     if (ixyx>=0) {
01012         fasp_blas_smat_mul_nc7 (&(LU->offdiag[6][ixyc]), &(LU->offdiag[9][ixyc]), smat);
01013         fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[3][ic]));
01014     }
01015 }
01016
01017 // comput beta3[i]
01018 if (i+nline<ngrid) {
01019     memcpy (&(LU->offdiag[5][ic]), &(A->offdiag[3][ic]), 49*sizeof(REAL));
01020     if (ixyx>=0) {
01021         fasp_blas_smat_mul_nc7 (&(LU->offdiag[6][ixyc]), &(LU->offdiag[11][ixyc]), smat);
01022         fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[5][ic]));
01023     }
01024 }
01025
01026 // comput beta4[i]
01027 if (i+nplane-nline<ngrid) {
01028     if (ixl>=0) {
01029         fasp_blas_smat_mul_nc7 (&(LU->offdiag[2][ixlc]), &(LU->offdiag[9][ixlc]), smat);
01030         fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[7][ic]));
01031     }
01032     if (ix>=0) {
01033         fasp_blas_smat_mul_nc7 (&(LU->offdiag[4][ixc]), &(LU->offdiag[11][ixc]), smat);
01034         fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[7][ic]));
01035     }
01036 }
01037
01038 // comput beta5[i]
01039 if (i+nplane-1<ngrid) {
01040     fasp_blas_smat_mul_nc7 (&(LU->offdiag[0][ilc]), &(LU->offdiag[11][ilc]), smat);
01041     fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->offdiag[9][ic]));
01042 }
01043
01044 // comput d[i]
01045 {
01046     fasp_blas_smat_mul_nc7 (&(LU->offdiag[0][ilc]), &(LU->offdiag[1][ilc]), smat);
01047     fasp_blas_darray_axpy_nc7 (-1, smat, &(A->diag[ic]), &(LU->diag[ic]));
01048 }
01049
01050 if (ixl>=0) {
01051     fasp_blas_smat_mul_nc7 (&(LU->offdiag[2][ixlc]), &(LU->offdiag[3][ixlc]), smat);
01052     fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->diag[ic]));
01053 }
01054
01055 if (ix>=0) {
01056     fasp_blas_smat_mul_nc7 (&(LU->offdiag[4][ixc]), &(LU->offdiag[5][ixc]), smat);
01057     fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->diag[ic]));
01058 }
01059
01060 if (ixyx>=0) {
01061     fasp_blas_smat_mul_nc7 (&(LU->offdiag[6][ixyc]), &(LU->offdiag[7][ixyc]), smat);
01062     fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->diag[ic]));
01063 }
01064
01065 if (ixyl>=0) {
01066     fasp_blas_smat_mul_nc7 (&(LU->offdiag[8][ixylc]), &(LU->offdiag[9][ixylc]), smat);
01067     fasp_blas_darray_axpy_nc7 (-1, smat, &(LU->diag[ic]));
01068 }
01069 }
01070
01071
01072
01073
01074
01075
01076
01077
01078

```

```

01079         if (ixy>=0) {
01080             fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]), &(LU->offdiag[11][ixyc]), smat);
01081             fasp_blas_darray_axpy_nc7(-1, smat, &(LU->diag[ic]));
01082         }
01083
01084         //fasp_smat_inv_nc5(&(LU->diag[ic]));
01085         fasp_smat_inv(&(LU->diag[ic]), 7);
01086
01087     } // end for(i=1;i<ngrid;++i)
01088
01089 } // end if (nc == 7)
01090
01091 else {
01092     // compute the first row
01093     fasp_smat_inv(LU->diag, nc);
01094     memcpy(LU->offdiag[1], A->offdiag[1], nc2*sizeof(REAL));
01095     memcpy(LU->offdiag[5], A->offdiag[3], nc2*sizeof(REAL));
01096
01097     for(i=1;i<ngrid;++i) {
01098
01099         il=i-1; ix=i-nline; ixy=i-nplane; ixl=ix+1; ixyx=ixy+iline; ixyl=ixy+1;
01100         ic=i*nc2; ilc=il*nc2; icx=ix*nc2; ixlc=ixl*nc2; ixc=ixy*nc2; ixylc=ixyl*nc2; ixyxc=ixyx*nc2;
01101         // compute alpha6[i-nxy]
01102         if (ixy>=0)
01103             fasp_blas_smat_mul(&(A->offdiag[4][ixyc]), &(LU->diag[ixyc]), &(LU->offdiag[10][ixyc]), nc);
01104
01105         // compute alpha5[ixyl]
01106         if (ixyl>=0) {
01107             for (j=0; j<nc2; ++j) tc[j]=0;
01108             if (ixy>=0) {
01109                 fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]), &(LU->offdiag[1][ixyc]), smat, nc);
01110                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01111             }
01112
01113             fasp_blas_smat_mul(tc, &(LU->diag[ixylc]), &(LU->offdiag[8][ixylc]), nc);
01114         }
01115
01116         // compute alpha4[ixyx]
01117         if (ixyx>=0) {
01118             for (j=0; j<nc2; ++j) tc[j]=0;
01119             if (ixy>=0) {
01120                 fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]), &(LU->offdiag[5][ixyc]), smat, nc);
01121                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01122             }
01123             if (ixyl>=0) {
01124                 fasp_blas_smat_mul(&(LU->offdiag[8][ixylc]), &(LU->offdiag[3][ixylc]), smat, nc);
01125                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01126             }
01127
01128             fasp_blas_smat_mul(tc, &(LU->diag[ixyxc]), &(LU->offdiag[6][ixyxc]), nc);
01129         }
01130
01131         // compute alpha3[ix]
01132         if (ix>=0) {
01133
01134             memcpy(tc, &(A->offdiag[2][ixc]), nc2*sizeof(REAL));
01135             if (ixy>=0) {
01136                 fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]), &(LU->offdiag[7][ixyc]), smat, nc);
01137                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01138             }
01139
01140             fasp_blas_smat_mul(tc, &(LU->diag[ixc]), &(LU->offdiag[4][ixc]), nc);
01141         }
01142
01143         // compute alpha2[i-nx+1]
01144         if (ixl>=0) {
01145
01146             for (j=0; j<nc2; ++j) tc[j]=0;
01147
01148             if (ix>=0) {
01149                 fasp_blas_smat_mul(&(LU->offdiag[4][ixc]), &(LU->offdiag[1][ixc]), smat, nc);
01150                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01151             }
01152
01153             if (ixyl>=0) {
01154                 fasp_blas_smat_mul(&(LU->offdiag[8][ixylc]), &(LU->offdiag[7][ixylc]), smat, nc);
01155                 fasp_blas_darray_axpy(nc2, -1, smat, tc);
01156             }
01157
01158             fasp_blas_smat_mul(tc, &(LU->diag[ixlc]), &(LU->offdiag[2][ixlc]), nc);
01159         }

```

```

01160
01161 // comput alpha1[i-1]
01162
01163 memcpy(tc, &(A->offdiag[0][ilc]), nc2*sizeof(REAL));
01164 if (ix>=0) {
01165     fasp_blas_smat_mul(&(LU->offdiag[4][ixc]), &(LU->offdiag[3][ixc]), smat, nc);
01166     fasp_blas_darray_axpy(nc2, -1, smat, tc);
01167 }
01168 if (ixy>=0) {
01169     fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]), &(LU->offdiag[9][ixyc]), smat, nc);
01170     fasp_blas_darray_axpy(nc2, -1, smat, tc);
01171 }
01172
01173 fasp_blas_smat_mul(tc, &(LU->diag[ilc]), &(LU->offdiag[0][ilc]), nc);
01174
01175 // comput beta1[i]
01176 if (i+1<ngrid) {
01177
01178     memcpy(&(LU->offdiag[1][ic]), &(A->offdiag[1][ic]), nc2*sizeof(REAL));
01179     if (ix1>=0) {
01180         fasp_blas_smat_mul(&(LU->offdiag[2][ix1c]), &(LU->offdiag[5][ix1c]), smat, nc);
01181         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[1][ic]));
01182     }
01183     if (ixy1>=0) {
01184         fasp_blas_smat_mul(&(LU->offdiag[8][ixy1c]), &(LU->offdiag[11][ixy1c]), smat, nc);
01185         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[1][ic]));
01186     }
01187 }
01188
01189 // comput beta2[i]
01190 if (i+nline-1<ngrid) {
01191
01192     {
01193         fasp_blas_smat_mul(&(LU->offdiag[0][ilc]), &(LU->offdiag[5][ilc]), smat, nc);
01194         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[3][ic]));
01195     }
01196
01197     if (ixyx>=0) {
01198         fasp_blas_smat_mul(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[9][ixyxc]), smat, nc);
01199         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[3][ic]));
01200     }
01201 }
01202
01203 // comput beta3[i]
01204 if (i+nline<ngrid) {
01205
01206     memcpy(&(LU->offdiag[5][ic]), &(A->offdiag[3][ic]), nc2*sizeof(REAL));
01207     if (ixyx>=0) {
01208         fasp_blas_smat_mul(&(LU->offdiag[6][ixyxc]), &(LU->offdiag[11][ixyxc]), smat, nc);
01209         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[5][ic]));
01210     }
01211 }
01212
01213 // comput beta4[i]
01214 if (i+nplane-nline<ngrid) {
01215
01216     if (ix1>=0) {
01217         fasp_blas_smat_mul(&(LU->offdiag[2][ix1c]), &(LU->offdiag[9][ix1c]), smat, nc);
01218         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[7][ic]));
01219     }
01220
01221     if (ix>=0) {
01222         fasp_blas_smat_mul(&(LU->offdiag[4][ixc]), &(LU->offdiag[11][ixc]), smat, nc);
01223         fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[7][ic]));
01224     }
01225 }
01226
01227 // comput beta5[i]
01228 if (i+nplane-1<ngrid) {
01229     fasp_blas_smat_mul(&(LU->offdiag[0][ilc]), &(LU->offdiag[11][ilc]), smat, nc);
01230     fasp_blas_darray_axpy(nc2, -1, smat, &(LU->offdiag[9][ic]));
01231 }
01232
01233 // comput d[i]
01234 {
01235     fasp_blas_smat_mul(&(LU->offdiag[0][ilc]), &(LU->offdiag[1][ilc]), smat, nc);
01236     fasp_blas_darray_axpyz(nc2, -1, smat, &(A->diag[ic]), &(LU->diag[ic]));
01237 }
01238
01239
01240

```

```

01241
01242         if (ix1>=0) {
01243             fasp_blas_smat_mul (&(LU->offdiag[2][ix1c]), &(LU->offdiag[3][ix1c]), smat, nc);
01244             fasp_blas_darray_axpy (nc2, -1, smat, &(LU->diag[ix1]));
01245         }
01246
01247         if (ix>=0) {
01248             fasp_blas_smat_mul (&(LU->offdiag[4][ixc]), &(LU->offdiag[5][ixc]), smat, nc);
01249             fasp_blas_darray_axpy (nc2, -1, smat, &(LU->diag[ix]));
01250         }
01251
01252         if (ixyx>=0) {
01253             fasp_blas_smat_mul (&(LU->offdiag[6][ixyc]), &(LU->offdiag[7][ixyc]), smat, nc);
01254             fasp_blas_darray_axpy (nc2, -1, smat, &(LU->diag[ix]));
01255         }
01256
01257
01258         if (ixy1>=0) {
01259             fasp_blas_smat_mul (&(LU->offdiag[8][ixy1c]), &(LU->offdiag[9][ixy1c]), smat, nc);
01260             fasp_blas_darray_axpy (nc2, -1, smat, &(LU->diag[ix]));
01261         }
01262
01263         if (ixy>=0) {
01264             fasp_blas_smat_mul (&(LU->offdiag[10][ixyc]), &(LU->offdiag[11][ixyc]), smat, nc);
01265             fasp_blas_darray_axpy (nc2, -1, smat, &(LU->diag[ix]));
01266         }
01267
01268         fasp_smat_inv (&(LU->diag[ix]), nc);
01269     }
01270 }
01271
01272 } // end else
01273
01274 fasp_mem_free(smat); smat = NULL;
01275 fasp_mem_free(tc); tc = NULL;
01276
01277 return;
01278 }
01279
01280 /*-----*/
01281 /*--      End of File      --*/
01282 /*-----*/

```

9.59 BlaiO.c File Reference

Matrix/vector input/output subroutines.

```

#include "fasp.h"
#include "fasp_functs.h"
#include "hb_io.h"
#include "BlaiOUtil.inl"

```

Functions

- void [fasp_dcsrvec_read1](#) (const char *filename, [dCSRmat](#) *A, [dvector](#) *b)
Read A and b from a SINGLE disk file.
- void [fasp_dcsrvec_read2](#) (const char *filerhs, const char *filename, [dCSRmat](#) *A, [dvector](#) *b)
Read A and b from two separate disk files.
- void [fasp_dcsr_read](#) (const char *filename, [dCSRmat](#) *A)
Read A from matrix disk file in IJ format.
- void [fasp_dcoo_read](#) (const char *filename, [dCSRmat](#) *A)
Read A from matrix disk file in IJ format – indices starting from 0.
- void [fasp_dcoo_read1](#) (const char *filename, [dCSRmat](#) *A)
Read A from matrix disk file in IJ format – indices starting from 1.
- void [fasp_dcoovec_bin_read](#) (const char *fni, const char *fnj, const char *fna, const char *fnb, [dCSRmat](#) *A, [dvector](#) *b)

- Read A from matrix disk files in IJ format (three binary files)*

 - void `fasp_dcoo_shift_read` (const char *filename, `dCSRmat` *A)
- Read A from matrix disk file in IJ format – indices starting from 0.*

 - void `fasp_dmtx_read` (const char *filename, `dCSRmat` *A)
- Read A from matrix disk file in MatrixMarket general format.*

 - void `fasp_dmtxsym_read` (const char *filename, `dCSRmat` *A)
- Read A from matrix disk file in MatrixMarket sym format.*

 - void `fasp_dstr_read` (const char *filename, `dSTRmat` *A)
- Read A from a disk file in `dSTRmat` format.*

 - void `fasp_dbsr_read` (const char *filename, `DBSRmat` *A)
- Read A from a disk file in `DBSRmat` format.*

 - void `fasp_dvecind_read` (const char *filename, `dvector` *b)
- Read b from matrix disk file.*

 - void `fasp_dvec_read` (const char *filename, `dvector` *b)
- Read b from a disk file in array format.*

 - void `fasp_ivecind_read` (const char *filename, `ivector` *b)
- Read b from matrix disk file.*

 - void `fasp_ivec_read` (const char *filename, `ivector` *b)
- Read b from a disk file in array format.*

 - void `fasp_dcsrvec_write1` (const char *filename, `dCSRmat` *A, `dvector` *b)
- Write A and b to a SINGLE disk file.*

 - void `fasp_dcsrvec_write2` (const char *filename, const char *filerhs, `dCSRmat` *A, `dvector` *b)
- Write A and b to two separate disk files.*

 - void `fasp_dcoo_write` (const char *filename, `dCSRmat` *A)
- Write a matrix to disk file in IJ format (coordinate format)*

 - void `fasp_dstr_write` (const char *filename, `dSTRmat` *A)
- Write a `dSTRmat` to a disk file.*

 - void `fasp_dbsr_print` (const char *filename, `DBSRmat` *A)
- Print a `DBSRmat` to a disk file in a readable format.*

 - void `fasp_dbsr_write` (const char *filename, `DBSRmat` *A)
- Write a `DBSRmat` to a disk file.*

 - void `fasp_dvec_write` (const char *filename, `dvector` *vec)
- Write a dvector to disk file.*

 - void `fasp_dvecind_write` (const char *filename, `dvector` *vec)
- Write a dvector to disk file in coordinate format.*

 - void `fasp_ivec_write` (const char *filename, `ivector` *vec)
- Write a ivector to disk file in coordinate format.*

 - void `fasp_dvec_print` (const INT n, `dvector` *u)
- Print first n entries of a vector of REAL type.*

 - void `fasp_ivec_print` (const INT n, `ivector` *u)
- Print first n entries of a vector of INT type.*

 - void `fasp_dcsr_print` (const `dCSRmat` *A)
- Print out a `dCSRmat` matrix in coordinate format.*

 - void `fasp_dcoo_print` (const `dCOOmat` *A)
- Print out a `dCOOmat` matrix in coordinate format.*

 - void `fasp_dbsr_write_coo` (const char *filename, const `DBSRmat` *A)
- Print out a `DBSRmat` matrix in coordinate format for matlab spy.*

- void [fasp_dcsr_write_coo](#) (const char *filename, const [dCSRmat](#) *A)
Print out a [dCSRmat](#) matrix in coordinate format for matlab spy.
- void [fasp_dcsr_write_mtx](#) (const char *filename, const [dCSRmat](#) *A)
Print out a [dCSRmat](#) matrix in coordinate format for MatrixMarket.
- void [fasp_dstr_print](#) (const [dSTRmat](#) *A)
Print out a [dSTRmat](#) matrix in coordinate format.
- void [fasp_matrix_read](#) (const char *filename, void *A)
Read matrix from different kinds of formats from both ASCII and binary files.
- void [fasp_matrix_read_bin](#) (const char *filename, void *A)
Read matrix in binary format.
- void [fasp_matrix_write](#) (const char *filename, void *A, const [INT](#) flag)
write matrix from different kinds of formats from both ASCII and binary files
- void [fasp_vector_read](#) (const char *filerhs, void *b)
Read RHS vector from different kinds of formats in ASCII or binary files.
- void [fasp_vector_write](#) (const char *filerhs, void *b, const [INT](#) flag)
write RHS vector from different kinds of formats in both ASCII and binary files
- void [fasp_hb_read](#) (const char *input_file, [dCSRmat](#) *A, [dvector](#) *b)
Read matrix and right-hans side from a HB format file.

Variables

- int [ilength](#)
- int [dlength](#)

9.59.1 Detailed Description

Matrix/vector input/output subroutines.

Note

Read, write or print a matrix or a vector in various formats

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxConvert.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaFormat.c](#), [BlaSparseBSR.c](#), [BlaSparseCOO.c](#), [BlaSparseCSR.c](#), and [BlaSpmvCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaiO.c](#).

9.59.2 Function Documentation

9.59.2.1 [fasp_dbsr_print\(\)](#)

```
void fasp_dbsr_print (
    const char * filename,
    dBSRmat * A )
```

Print a [dBSRmat](#) to a disk file in a readable format.

Parameters

<i>filename</i>	File name for A
<i>A</i>	Pointer to the dBSRmat matrix A

Author

Chensong Zhang

Date

01/07/2021

Definition at line [1292](#) of file [BlalO.c](#).

9.59.2.2 fasp_dbsr_read()

```
void fasp_dbsr_read (
    const char * filename,
    dBSRmat * A )
```

Read A from a disk file in [dBSRmat](#) format.

Parameters

<i>filename</i>	File name for matrix A
<i>A</i>	Pointer to the dBSRmat A

Note

This routine reads a [dBSRmat](#) matrix from a disk file in the following format:

File format:

- ROW, COL, NNZ
- nb: size of each block
- storage_manner: storage manner of each block
- ROW+1: length of IA
- IA(i), i=0:ROW
- NNZ: length of JA
- JA(i), i=0:NNZ-1
- NNZ*nb*nb: length of val
- val(i), i=0:NNZ*nb*nb-1

Author

Xiaozhe Hu

Date

10/29/2010

Definition at line [807](#) of file [BlalO.c](#).

9.59.2.3 fasp_dbsr_write()

```
void fasp_dbsr_write (
    const char * filename,
    dBSRmat * A )
```

Write a [dBSRmat](#) to a disk file.

Parameters

<i>filename</i>	File name for A
<i>A</i>	Pointer to the dBSRmat matrix A

Note

The routine writes the specified REAL vector in BSR format. Refer to the reading subroutine [fasp_dbsr_read](#).

Author

Shiquan Zhang

Date

10/29/2010

Definition at line [1336](#) of file [BlaiO.c](#).

9.59.2.4 fasp_dbsr_write_coo()

```
void fasp_dbsr_write_coo (
    const char * filename,
    const dBSRmat * A )
```

Print out a [dBSRmat](#) matrix in coordinate format for matlab spy.

Parameters

<i>filename</i>	Name of file to write to
<i>A</i>	Pointer to the dBSRmat matrix A

Author

Chunsheng Feng

Date

11/14/2013

Modified by Chensong Zhang on 06/14/2014: Fix index problem.

Definition at line [1568](#) of file [BlaiO.c](#).

9.59.2.5 fasp_dcoo_print()

```
void fasp_dcoo_print (
    const dCOOmat * A )
```

Print out a [dCOOmat](#) matrix in coordinate format.

Parameters

<i>A</i>	Pointer to the dCOOmat matrix <i>A</i>
----------	--

Author

Ziteng Wang

Date

12/24/2012

Definition at line [1545](#) of file [BlalO.c](#).

9.59.2.6 fasp_dcoo_read()

```
void fasp_dcoo_read (
    const char * filename,
    dCSRmat * A )
```

Read *A* from matrix disk file in IJ format – indices starting from 0.

Parameters

<i>filename</i>	File name for matrix
<i>A</i>	Pointer to the CSR matrix

Note

File format:

- nrow ncol nnz % number of rows, number of columns, and nnz
- i j a_ij % i, j a_ij in each line

After reading, it converts the matrix to [dCSRmat](#) format.

Author

Xuehai Huang, Chensong Zhang

Date

03/29/2009

Definition at line [332](#) of file [BlalO.c](#).

9.59.2.7 fasp_dcoo_read1()

```
void fasp_dcoo_read1 (
    const char * filename,
    dCSRmat * A )
```

Read *A* from matrix disk file in IJ format – indices starting from 1.

Parameters

<i>filename</i>	File name for matrix
<i>A</i>	Pointer to the CSR matrix

Note

File format:

- nrow ncol nnz % number of rows, number of columns, and nnz
- i j a_ij % i, j a_ij in each line

Author

Xiaozhe Hu, Chensong Zhang

Date

03/24/2013

Modified by Chensong Zhang on 01/12/2019: Convert COO to CSR
Definition at line 384 of file [BlaiO.c](#).

9.59.2.8 fasp_dcoo_shift_read()

```
void fasp_dcoo_shift_read (
    const char * filename,
    dCSRmat * A )
```

Read A from matrix disk file in IJ format – indices starting from 0.

Parameters

<i>filename</i>	File name for matrix
<i>A</i>	Pointer to the CSR matrix

Note

File format:

- nrow ncol nnz % number of rows, number of columns, and nnz
- i j a_ij % i, j a_ij in each line

i and j suppose to start with index 1!!!

After read in, it shifts the index to C fashion and converts the matrix to [dCSRmat](#) format.

Author

Xiaozhe Hu

Date

04/01/2014

Definition at line 514 of file [BlaiO.c](#).

9.59.2.9 fasp_dcoo_write()

```
void fasp_dcoo_write (
    const char * filename,
    dCSRmat * A )
```

Write a matrix to disk file in IJ format (coordinate format)

Parameters

<i>A</i>	pointer to the dCSRmat matrix
<i>filename</i>	char for vector file name

Note

The routine writes the specified REAL vector in COO format. Refer to the reading subroutine [fasp_dcoo_read](#).

File format:

- The first line of the file gives the number of rows, the number of columns, and the number of nonzeros.
- Then gives nonzero values in i j a(i,j) format.

Author

Chensong Zhang

Date

03/29/2009

Definition at line [1207](#) of file [BlalO.c](#).

9.59.2.10 fasp_dcoovec_bin_read()

```
void fasp_dcoovec_bin_read (
    const char * fni,
    const char * fnj,
    const char * fna,
    const char * fnb,
    dCSRmat * A,
    dvector * b )
```

Read A from matrix disk files in IJ format (three binary files)

Parameters

<i>fni</i>	File name for matrix i-index
<i>fnj</i>	File name for matrix j-index
<i>fna</i>	File name for matrix values
<i>fnb</i>	File name for vector values
<i>A</i>	Pointer to the CSR matrix
<i>b</i>	Pointer to the vector

Note

After reading, it converts the matrix to [dCSRmat](#) format.

Author

Chensong Zhang

Date

08/27/2022

Definition at line [437](#) of file [BlalO.c](#).

9.59.2.11 fasp_dcsr_print()

```
void fasp_dcsr_print (
    const dCSRmat * A )
```

Print out a [dCSRmat](#) matrix in coordinate format.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix A
----------	---

Author

Xuehai Huang

Date

03/29/2009

Definition at line [1523](#) of file [BlalO.c](#).

9.59.2.12 fasp_dcsr_read()

```
void fasp_dcsr_read (
    const char * filename,
    dCSRmat * A )
```

Read A from matrix disk file in IJ format.

Parameters

<i>filename</i>	Char for matrix file name
<i>A</i>	Pointer to the CSR matrix

Author

Ziteng Wang

Date

12/25/2012

Definition at line 252 of file [BlalO.c](#).**9.59.2.13 fasp_dcsr_write_coo()**

```
void fasp_dcsr_write_coo (
    const char * filename,
    const dCSRmat * A )
```

Print out a [dCSRmat](#) matrix in coordinate format for matlab spy.

Parameters

<i>filename</i>	Name of file to write to
<i>A</i>	Pointer to the dCSRmat matrix A

Author

Chunsheng Feng

Date

11/14/2013

Note

Output indices start from 1 instead of 0!

Definition at line 1623 of file [BlalO.c](#).**9.59.2.14 fasp_dcsr_write_mtx()**

```
void fasp_dcsr_write_mtx (
    const char * filename,
    const dCSRmat * A )
```

Print out a [dCSRmat](#) matrix in coordinate format for MatrixMarket.

Parameters

<i>filename</i>	Name of file to write to
<i>A</i>	Pointer to the dCSRmat matrix A

Author

Chensong Zhang

Date

08/28/2022

Note

Output indices start from 1 instead of 0!

Definition at line 1664 of file [BlaiO.c](#).

9.59.2.15 fasp_dcsrvec_read1()

```
void fasp_dcsrvec_read1 (
    const char * filename,
    dCSRmat * A,
    dvector * b )
```

Read A and b from a SINGLE disk file.

Parameters

<i>filename</i>	File name
<i>A</i>	Pointer to the CSR matrix
<i>b</i>	Pointer to the dvector

Note

This routine reads a [dCSRmat](#) matrix and a dvector vector from a single disk file. The difference between this and [fasp_dcoovec_read](#) is that this routine support non-square matrices.

File format:

- nrow ncol % number of rows and number of columns
- ia(j), j=0:nrow % row index
- ja(j), j=0:nnz-1 % column index
- a(j), j=0:nnz-1 % entry value
- n % number of entries
- b(j), j=0:n-1 % entry value

Author

Xuehai Huang

Date

03/29/2009

Modified by Chensong Zhang on 03/14/2012

Definition at line 63 of file [BlaiO.c](#).

9.59.2.16 fasp_dcsrvec_read2()

```
void fasp_dcsrvec_read2 (
    const char * filemat,
    const char * filerhs,
    dCSRmat * A,
    dvector * b )
```

Read A and b from two separate disk files.

Parameters

<i>filemat</i>	File name for matrix
<i>filerhs</i>	File name for right-hand side
<i>A</i>	Pointer to the dCSR matrix
<i>b</i>	Pointer to the dvector

Note

This routine reads a [dCSRmat](#) matrix and a dvector vector from a disk file.

CSR matrix file format:

- *nrow* % number of columns (rows)
- *ia(j)*, *j*=0:*nrow* % row index
- *ja(j)*, *j*=0:*nnz*-1 % column index
- *a(j)*, *j*=0:*nnz*-1 % entry value

RHS file format:

- *n* % number of entries
- *b(j)*, *j*=0:*nrow*-1 % entry value

Indices start from 1, NOT 0!!!

Author

Zhiyang Zhou

Date

2010/08/06

Modified by Chensong Zhang on 2012/01/05
Definition at line [164](#) of file [BlalO.c](#).

9.59.2.17 fasp_dcsrvec_write1()

```
void fasp_dcsrvec_writel (
    const char * filename,
    dCSRmat * A,
    dvector * b )
```

Write A and b to a SINGLE disk file.

Parameters

<i>filename</i>	File name
<i>A</i>	Pointer to the CSR matrix
<i>b</i>	Pointer to the dvector

Note

This routine writes a [dCSRmat](#) matrix and a dvector vector to a single disk file.

File format:

- nrow ncol % number of rows and number of columns
- ia(j), j=0:nrow % row index
- ja(j), j=0:nnz-1 % column index
- a(j), j=0:nnz-1 % entry value
- n % number of entries
- b(j), j=0:n-1 % entry value

Author

Feiteng Huang

Date

05/19/2012

Modified by Chensong on 12/26/2012

Definition at line [1079](#) of file [BlaiO.c](#).

9.59.2.18 fasp_dcsrvec_write2()

```
void fasp_dcsrvec_write2 (
    const char * filemat,
    const char * filerhs,
    dCSRmat * A,
    dvector * b )
```

Write A and b to two separate disk files.

Parameters

<i>filemat</i>	File name for matrix
<i>filerhs</i>	File name for right-hand side
<i>A</i>	Pointer to the dCSR matrix
<i>b</i>	Pointer to the dvector

Note

This routine writes a [dCSRmat](#) matrix and a dvector vector to two disk files.

CSR matrix file format:

- nrow % number of columns (rows)
- ia(j), j=0:nrow % row index
- ja(j), j=0:nnz-1 % column index
- a(j), j=0:nnz-1 % entry value

RHS file format:

- n % number of entries

- `b(j)`, `j=0:nrow-1` % entry value

Indices start from 1, NOT 0!!!

Author

Feiteng Huang

Date

05/19/2012

Definition at line 1145 of file [BlalO.c](#).

9.59.2.19 fasp_dmtx_read()

```
void fasp_dmtx_read (
    const char * filename,
    dCSRmat * A )
```

Read A from matrix disk file in MatrixMarket general format.

Parameters

<i>filename</i>	File name for matrix
<i>A</i>	Pointer to the CSR matrix

Note

File format: This routine reads a MatrixMarket general matrix from a mtx file. And it converts the matrix to [dCSRmat](#) format. For details of mtx format, please refer to <http://math.nist.gov/MatrixMarket/>.

Indices start from 1, NOT 0!!!

Author

Chensong Zhang

Date

09/05/2011

Definition at line 567 of file [BlalO.c](#).

9.59.2.20 fasp_dmtxsym_read()

```
void fasp_dmtxsym_read (
    const char * filename,
    dCSRmat * A )
```

Read A from matrix disk file in MatrixMarket sym format.

Parameters

<i>filename</i>	File name for matrix
<i>A</i>	Pointer to the CSR matrix

Note

File format: This routine reads a MatrixMarket symmetric matrix from a mtx file. And it converts the matrix to [dCSRmat](#) format. For details of mtx format, please refer to <http://math.nist.gov/MatrixMarket/>. Indices start from 1, NOT 0!!!

Author

Chensong Zhang

Date

09/02/2011

Definition at line 624 of file [BlalO.c](#).

9.59.2.21 fasp_dstr_print()

```
void fasp_dstr_print (  
    const dSTRmat * A )
```

Print out a [dSTRmat](#) matrix in coordinate format.

Parameters

<i>A</i>	Pointer to the dSTRmat matrix A
----------	---

Author

Ziteng Wang

Date

12/24/2012

Definition at line 1701 of file [BlalO.c](#).

9.59.2.22 fasp_dstr_read()

```
void fasp_dstr_read (  
    const char * filename,  
    dSTRmat * A )
```

Read A from a disk file in [dSTRmat](#) format.

Parameters

<i>filename</i>	File name for the matrix
<i>A</i>	Pointer to the dSTRmat

Note

This routine reads a [dSTRmat](#) matrix from a disk file. After done, it converts the matrix to [dCSRmat](#) format.

File format:

- nx, ny, nz

- nc: number of components
- nband: number of bands
- n: size of diagonal, you must have diagonal
- diag(j), j=0:n-1
- offset, length: offset and length of off-diag1
- offdiag(j), j=0:length-1

Author

Xuehai Huang

Date

03/29/2009

Definition at line 699 of file [BlalO.c](#).**9.59.2.23 fasp_dstr_write()**

```
void fasp_dstr_write (
    const char * filename,
    dSTRmat * A )
```

Write a [dSTRmat](#) to a disk file.**Parameters**

<i>filename</i>	File name for A
<i>A</i>	Pointer to the dSTRmat matrix A

NoteThe routine writes the specified REAL vector in STR format. Refer to the reading subroutine [fasp_dstr_read](#).**Author**

Shiquan Zhang

Date

03/29/2010

Definition at line 1241 of file [BlalO.c](#).**9.59.2.24 fasp_dvec_print()**

```
void fasp_dvec_print (
    const INT n,
    dvector * u )
```

Print first n entries of a vector of REAL type.

Parameters

<i>n</i>	An interger (if n=0, then print all entries)
<i>u</i>	Pointer to a dvector

Author

Chensong Zhang

Date

03/29/2009

Definition at line 1482 of file [BlaiO.c](#).

9.59.2.25 fasp_dvec_read()

```
void fasp_dvec_read (
    const char * filename,
    dvector * b )
```

Read b from a disk file in array format.

Parameters

<i>filename</i>	File name for vector b
<i>b</i>	Pointer to the dvector b (output)

Note

File Format:

- nrow
- val_j, j=0:nrow-1

Author

Chensong Zhang

Date

03/29/2009

Definition at line 938 of file [BlaiO.c](#).

9.59.2.26 fasp_dvec_write()

```
void fasp_dvec_write (
    const char * filename,
    dvector * vec )
```

Write a dvector to disk file.

Parameters

<i>vec</i>	Pointer to the dvector
<i>filename</i>	File name

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1388 of file [BlalO.c](#).

9.59.2.27 fasp_dvecind_read()

```
void fasp_dvecind_read (
    const char * filename,
    dvector * b )
```

Read b from matrix disk file.

Parameters

<i>filename</i>	File name for vector b
<i>b</i>	Pointer to the dvector b (output)

Note

File Format:

- nrow
- ind_j, val_j, j=0:nrow-1

Because the index is given, order is not important!

Author

Chensong Zhang

Date

03/29/2009

Definition at line 887 of file [BlalO.c](#).

9.59.2.28 fasp_dvecind_write()

```
void fasp_dvecind_write (
    const char * filename,
    dvector * vec )
```

Write a dvector to disk file in coordinate format.

Parameters

<i>vec</i>	Pointer to the dvector
<i>filename</i>	File name

Note

The routine writes the specified REAL vector in IJ format.

- The first line of the file is the length of the vector;
- After that, each line gives index and value of the entries.

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1420 of file [BlaiO.c](#).

9.59.2.29 fasp_hb_read()

```
fasp_hb_read (
    const char * input_file,
    dCSRmat * A,
    dvector * b )
```

Read matrix and right-hans side from a HB format file.

Parameters

<i>input_file</i>	File name of vector file
<i>A</i>	Pointer to the matrix
<i>b</i>	Pointer to the vector

Note

Modified from the C code hb_io_prb.c by John Burkardt, which is NOT part of the FASP project!

Author

Xiaoehe Hu

Date

05/30/2014

Definition at line 2206 of file [BlaiO.c](#).

9.59.2.30 fasp_ivec_print()

```
void fasp_ivec_print (
    const INT n,
    ivec * u )
```

Print first n entries of a vector of INT type.

Parameters

<i>n</i>	An interger (if n=0, then print all entries)
<i>u</i>	Pointer to an ivector

Author

Chensong Zhang

Date

03/29/2009

Definition at line 1503 of file [BlalO.c](#).

9.59.2.31 fasp_ivec_read()

```
void fasp_ivec_read (
    const char * filename,
    ivector * b )
```

Read b from a disk file in array format.

Parameters

<i>filename</i>	File name for vector b
<i>b</i>	Pointer to the dvector b (output)

Note

File Format:

- nrow
- val_j, j=0:nrow-1

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1029 of file [BlalO.c](#).

9.59.2.32 fasp_ivec_write()

```
void fasp_ivec_write (
    const char * filename,
    ivector * vec )
```

Write a ivector to disk file in coordinate format.

Parameters

<i>vec</i>	Pointer to the dvector
<i>filename</i>	File name

Note

The routine writes the specified INT vector in IJ format.

- The first line of the file is the length of the vector;
- After that, each line gives index and value of the entries.

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1452 of file [BlaiO.c](#).

9.59.2.33 fasp_ivecind_read()

```
void fasp_ivecind_read (  
    const char * filename,  
    ivector * b )
```

Read b from matrix disk file.

Parameters

<i>filename</i>	File name for vector b
<i>b</i>	Pointer to the dvector b (output)

Note

File Format:

- nrow
- ind_j, val_j ... j=0:nrow-1

Author

Chensong Zhang

Date

03/29/2009

Definition at line 989 of file [BlaiO.c](#).

9.59.2.34 fasp_matrix_read()

```
fasp_matrix_read (
    const char * filename,
    void * A )
```

Read matrix from different kinds of formats from both ASCII and binary files.

Parameters

<i>filename</i>	File name of matrix file
<i>A</i>	Pointer to the matrix

Note

Flags for matrix file format:

- fileflag % fileflag = 1: binary, fileflag = 0000: ASCII
- formatflag % a 3-digit number for internal use, see below
- matrix % different types of matrix

Meaning of formatflag:

- matrixflag % first digit of formatflag
 - matrixflag = 1: CSR format
 - matrixflag = 2: BSR format
 - matrixflag = 3: STR format
 - matrixflag = 4: COO format
 - matrixflag = 5: MTX format
 - matrixflag = 6: MTX symmetrical format
- ilength % third digit of formatflag, length of INT
- dlength % fourth digit of formatflag, length of REAL

Author

Ziteng Wang

Date

12/24/2012

Modified by Chensong Zhang on 05/01/2013

Definition at line 1735 of file [BlaiO.c](#).

9.59.2.35 fasp_matrix_read_bin()

```
void fasp_matrix_read_bin (
    const char * filename,
    void * A )
```

Read matrix in binary format.

Parameters

<i>filename</i>	File name of matrix file
<i>A</i>	Pointer to the matrix

Author

Xiaozhe Hu

Date

04/14/2013

Modified by Chensong Zhang on 05/01/2013: Use it to read binary files!!!
Definition at line 1849 of file [BlalO.c](#).

9.59.2.36 fasp_matrix_write()

```
fasp_matrix_write (
    const char * filename,
    void * A,
    const INT flag )
```

write matrix from different kinds of formats from both ASCII and binary files

Parameters

<i>filename</i>	File name of matrix file
<i>A</i>	Pointer to the matrix
<i>flag</i>	Type of file and matrix, a 3-digit number

Note

Meaning of flag:

- fileflag % fileflag = 1: binary, fileflag = 0: ASCII
- matrixflag
 - matrixflag = 1: CSR format
 - matrixflag = 2: BSR format
 - matrixflag = 3: STR format

Matrix file format:

- fileflag % fileflag = 1: binary, fileflag = 0000: ASCII
- formatflag % a 3-digit number
- matrixflag % different kinds of matrix judged by formatflag

Author

Ziteng Wang

Date

12/24/2012

Definition at line 1921 of file [BlalO.c](#).

9.59.2.37 fasp_vector_read()

```
fasp_vector_read (
    const char * filerhs,
    void * b )
```

Read RHS vector from different kinds of formats in ASCII or binary files.

Parameters

<i>filerhs</i>	File name of vector file
<i>b</i>	Pointer to the vector

Note

Matrix file format:

- fileflag % fileflag = 1: binary, fileflag = 0000: ASCII
- formatflag % a 3-digit number
- vector % different kinds of vector judged by formatflag

Meaning of formatflag:

- vectorflag % first digit of formatflag
 - vectorflag = 1: dvec format
 - vectorflag = 2: ivec format
 - vectorflag = 3: dvecind format
 - vectorflag = 4: ivecind format
- ilength % second digit of formatflag, length of INT
- dlength % third digit of formatflag, length of REAL

Author

Ziteng Wang

Date

12/24/2012

Definition at line 2011 of file [BlaiO.c](#).

9.59.2.38 fasp_vector_write()

```
fasp_vector_write (
    const char * filerhs,
    void * b,
    const INT flag )
```

write RHS vector from different kinds of formats in both ASCII and binary files

Parameters

<i>filerhs</i>	File name of vector file
<i>b</i>	Pointer to the vector
<i>flag</i>	Type of file and vector, a 2-digit number

Note

Meaning of the flags

- fileflag % fileflag = 1: binary, fileflag = 0: ASCII

- vectorflag
 - * vectorflag = 1: dvec format
 - * vectorflag = 2: ivec format
 - * vectorflag = 3: dvecind format
 - * vectorflag = 4: ivecind format

Matrix file format:

- fileflag % fileflag = 1: binary, fileflag = 0000: ASCII
 - formatflag % a 2-digit number
- vectorflag % different kinds of vector judged by formatflag

Author

Ziteng Wang

Date

12/24/2012

Modified by Chensong Zhang on 05/02/2013: fix a bug when writing in binary format
Definition at line [2119](#) of file [BlaIO.c](#).

9.59.3 Variable Documentation

9.59.3.1 dlength

int dlength
Length of REAL in byte
Definition at line [24](#) of file [BlaIO.c](#).

9.59.3.2 ilength

int ilength
Length of INT in byte
Definition at line [23](#) of file [BlaIO.c](#).

9.60 BlaIO.c

[Go to the documentation of this file.](#)

```
00001
00018 #include "fasp.h"
00019 #include "fasp_funcs.h"
00020 #include "hb_io.h"
00021
00022 // Flags which indicates lengths of INT and REAL numbers
00023 int ilength;
00024 int dlength;
00026 /*-----*/
00027 /*--  Declare Private Functions  --*/
00028 /*-----*/
00029
00030 #include "BlaIOUtil.inl"
00031
00032 /*-----*/
00033 /*--      Public Functions      --*/
```

```

00034 /*-----*/
00035
00063 void fasp_dcsrvec_read1(const char* filename, dCSRmat* A, dvector* b)
00064 {
00065     int i, m, n, idata;
00066     REAL ddata;
00067
00068     // Open input disk file
00069     FILE* fp = fopen(filename, "r");
00070
00071     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00072
00073     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00074
00075     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00076
00077     // Read CSR matrix
00078     if (fscanf(fp, "%d %d", &m, &n) > 0) {
00079         A->row = m;
00080         A->col = n;
00081     } else {
00082         fasp_chkerr(ERROR_WRONG_FILE, filename);
00083     }
00084
00085     A->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00086     for (i = 0; i <= m; ++i) {
00087         if (fscanf(fp, "%d", &idata) > 0)
00088             A->IA[i] = idata;
00089         else {
00090             fasp_chkerr(ERROR_WRONG_FILE, filename);
00091         }
00092     }
00093
00094     INT nnz = A->IA[m] - A->IA[0];
00095
00096     A->nnz = nnz;
00097     A->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00098     A->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
00099
00100     for (i = 0; i < nnz; ++i) {
00101         if (fscanf(fp, "%d", &idata) > 0)
00102             A->JA[i] = idata;
00103         else {
00104             fasp_chkerr(ERROR_WRONG_FILE, filename);
00105         }
00106     }
00107
00108     for (i = 0; i < nnz; ++i) {
00109         if (fscanf(fp, "%lf", &ddata) > 0)
00110             A->val[i] = ddata;
00111         else {
00112             fasp_chkerr(ERROR_WRONG_FILE, filename);
00113         }
00114     }
00115
00116     // Read RHS vector
00117     if (fscanf(fp, "%d", &m) > 0) b->row = m;
00118
00119     b->val = (REAL*)fasp_mem_calloc(m, sizeof(REAL));
00120
00121     for (i = 0; i < m; ++i) {
00122         if (fscanf(fp, "%lf", &ddata) > 0)
00123             b->val[i] = ddata;
00124         else {
00125             fasp_chkerr(ERROR_WRONG_FILE, filename);
00126         }
00127     }
00128
00129     fclose(fp);
00130 }
00131
00164 void fasp_dcsrvec_read2(const char* filemat, const char* filerhs, dCSRmat* A,
00165                         dvector* b)
00166 {
00167     int i, n, tempi;
00168
00169     /* read the matrix from file */
00170     FILE* fp = fopen(filemat, "r");
00171
00172     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filemat);
00173

```

```

00174     printf("%s: reading file %s ...\n", __FUNCTION__, filemat);
00175
00176     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00177
00178     if (fscanf(fp, "%d\n", &n) > 0) {
00179         A->row = n;
00180         A->col = n;
00181         A->IA = (INT*)fasp_mem_calloc(n + 1, sizeof(INT));
00182     } else {
00183         fasp_chkerr(ERROR_WRONG_FILE, filemat);
00184     }
00185
00186     for (i = 0; i <= n; ++i) {
00187         if (fscanf(fp, "%d\n", &tempi) > 0)
00188             A->IA[i] = tempi - 1;
00189         else {
00190             fasp_chkerr(ERROR_WRONG_FILE, filemat);
00191         }
00192     }
00193
00194     INT nz = A->IA[n];
00195     A->nnz = nz;
00196     A->JA = (INT*)fasp_mem_calloc(nz, sizeof(INT));
00197     A->val = (REAL*)fasp_mem_calloc(nz, sizeof(REAL));
00198
00199     for (i = 0; i < nz; ++i) {
00200         if (fscanf(fp, "%d\n", &tempi) > 0)
00201             A->JA[i] = tempi - 1;
00202         else {
00203             fasp_chkerr(ERROR_WRONG_FILE, filemat);
00204         }
00205     }
00206
00207     for (i = 0; i < nz; ++i) {
00208         if (fscanf(fp, "%le\n", &(A->val[i])) <= 0) {
00209             fasp_chkerr(ERROR_WRONG_FILE, filemat);
00210         }
00211     }
00212
00213     fclose(fp);
00214
00215     /* Read the rhs from file */
00216     b->row = n;
00217     b->val = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
00218
00219     fp = fopen(filerhs, "r");
00220
00221     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);
00222
00223     printf("%s: reading file %s ...\n", __FUNCTION__, filerhs);
00224
00225     if (fscanf(fp, "%d\n", &n) < 0) fasp_chkerr(ERROR_WRONG_FILE, filerhs);
00226
00227     if (n != b->row) {
00228         printf("### WARNING: rhs size = %d, matrix size = %d!\n", n, b->row);
00229         fasp_chkerr(ERROR_MAT_SIZE, filemat);
00230     }
00231
00232     for (i = 0; i < n; ++i) {
00233         if (fscanf(fp, "%le\n", &(b->val[i])) <= 0) {
00234             fasp_chkerr(ERROR_WRONG_FILE, filerhs);
00235         }
00236     }
00237
00238     fclose(fp);
00239 }
00240
00252 void fasp_dcsr_read(const char* filename, dCSRmat* A)
00253 {
00254     int i, m, idata;
00255     REAL ddata;
00256
00257     // Open input disk file
00258     FILE* fp = fopen(filename, "r");
00259
00260     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00261
00262     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00263
00264     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00265

```



```

00266 // Read CSR matrix
00267 if (fscanf(fp, "%d", &m) > 0)
00268     A->row = A->col = m;
00269 else {
00270     fasp_chkerr(ERROR_WRONG_FILE, filename);
00271 }
00272
00273 A->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00274 for (i = 0; i <= m; ++i) {
00275     if (fscanf(fp, "%d", &idata) > 0)
00276         A->IA[i] = idata;
00277     else {
00278         fasp_chkerr(ERROR_WRONG_FILE, filename);
00279     }
00280 }
00281
00282 // If IA starts from 1, shift by -1
00283 if (A->IA[0] == 1)
00284     for (i = 0; i <= m; ++i) A->IA[i]--;
00285
00286 INT nnz = A->IA[m] - A->IA[0];
00287
00288 A->nnz = nnz;
00289 A->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00290 A->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
00291
00292 for (i = 0; i < nnz; ++i) {
00293     if (fscanf(fp, "%d", &idata) > 0)
00294         A->JA[i] = idata;
00295     else {
00296         fasp_chkerr(ERROR_WRONG_FILE, filename);
00297     }
00298 }
00299
00300 // If JA starts from 1, shift by -1
00301 if (A->JA[0] == 1)
00302     for (i = 0; i < nnz; ++i) A->JA[i]--;
00303
00304 for (i = 0; i < nnz; ++i) {
00305     if (fscanf(fp, "%lf", &ddata) > 0)
00306         A->val[i] = ddata;
00307     else {
00308         fasp_chkerr(ERROR_WRONG_FILE, filename);
00309     }
00310 }
00311
00312 fclose(fp);
00313 }
00314
00332 void fasp_dcoo_read(const char* filename, dCSRmat* A)
00333 {
00334     int i, j, k, m, n, nnz;
00335     REAL value;
00336
00337     FILE* fp = fopen(filename, "r");
00338
00339     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00340
00341     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00342
00343     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00344
00345     if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
00346         fasp_chkerr(ERROR_WRONG_FILE, filename);
00347     }
00348
00349     dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);
00350
00351     for (k = 0; k < nnz; k++) {
00352         if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00353             Atmp.rowind[k] = i;
00354             Atmp.colind[k] = j;
00355             Atmp.val[k] = value;
00356         } else {
00357             fasp_chkerr(ERROR_WRONG_FILE, filename);
00358         }
00359     }
00360
00361     fclose(fp);
00362
00363     fasp_format_dcoo_dcsr(&Atmp, A);

```

```

00364     fasp_dcoo_free(&Atmp);
00365 }
00366
00384 void fasp_dcoo_readl(const char* filename, dCSRmat* A)
00385 {
00386     int i, j, k, m, n, nnz;
00387     REAL value;
00388
00389     FILE* fp = fopen(filename, "r");
00390
00391     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00392
00393     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00394
00395     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00396
00397     if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
00398         fasp_chkerr(ERROR_WRONG_FILE, filename);
00399     }
00400
00401     dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);
00402
00403     for (k = 0; k < nnz; k++) {
00404         if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00405             Atmp.rowind[k] = i - 1;
00406             Atmp.colind[k] = j - 1;
00407             Atmp.val[k] = value;
00408         } else {
00409             fasp_chkerr(ERROR_WRONG_FILE, filename);
00410         }
00411     }
00412
00413     fclose(fp);
00414
00415     fasp_format_dcoo_dcsr(&Atmp, A);
00416     fasp_dcoo_free(&Atmp);
00417 }
00418
00437 void fasp_dcoovec_bin_read(const char* fni, const char* fnj, const char* fna,
00438                           const char* fnb, dCSRmat* A, dvector* b)
00439 {
00440     size_t n, type, nnz, i;
00441     FILE* fp;
00442
00443     fp = fopen(fnb, "rb");
00444     if (fp == NULL) {
00445         fasp_chkerr(ERROR_WRONG_FILE, fnb);
00446     }
00447     printf("%s: reading file %s ...\n", __FUNCTION__, fnb);
00448     fread(&n, sizeof(size_t), 1, fp);
00449     b->row = n;
00450     b->val = (double*)fasp_mem_calloc(n, sizeof(double));
00451     fread(b->val, sizeof(double), n, fp);
00452     fclose(fp);
00453
00454     fp = fopen(fni, "rb");
00455     if (fp == NULL) {
00456         fasp_chkerr(ERROR_WRONG_FILE, fni);
00457     }
00458     printf("%s: reading file %s ...\n", __FUNCTION__, fni);
00459     fread(&type, sizeof(size_t), 1, fp);
00460     fread(&nnz, sizeof(size_t), 1, fp);
00461     dCOOmat Atmp = fasp_dcoo_create(n, n, nnz);
00462     Atmp.rowind = (int*)fasp_mem_calloc(nnz, sizeof(int));
00463     fread(Atmp.rowind, sizeof(int), nnz, fp);
00464     for (i = 0; i < nnz; i++) Atmp.rowind[i] = Atmp.rowind[i] - 1;
00465     fclose(fp);
00466
00467     fp = fopen(fnj, "rb");
00468     if (fp == NULL) {
00469         fasp_chkerr(ERROR_WRONG_FILE, fnj);
00470     }
00471     printf("%s: reading file %s ...\n", __FUNCTION__, fnj);
00472     fread(&type, sizeof(size_t), 1, fp);
00473     fread(&nnz, sizeof(size_t), 1, fp);
00474     Atmp.colind = (int*)fasp_mem_calloc(nnz, sizeof(int));
00475     fread(Atmp.colind, sizeof(int), nnz, fp);
00476     for (i = 0; i < nnz; i++) Atmp.colind[i] = Atmp.colind[i] - 1;
00477     fclose(fp);
00478
00479     fp = fopen(fna, "rb");

```

```

00480     if (fp == NULL) {
00481         fasp_chkerr(ERROR_WRONG_FILE, fna);
00482     }
00483     printf("%s: reading file %s ...\n", __FUNCTION__, fna);
00484     fread(&type, sizeof(size_t), 1, fp);
00485     fread(&nnz, sizeof(size_t), 1, fp);
00486     Atmp.val = (double*)fasp_mem_calloc(nnz, sizeof(double));
00487     fread(Atmp.val, sizeof(double), nnz, fp);
00488     fclose(fp);
00489
00490     fasp_format_dcoo_dcsr(&Atmp, A);
00491     fasp_dcoo_free(&Atmp);
00492 }
00493
00514 void fasp_dcoo_shift_read(const char* filename, dCSRmat* A)
00515 {
00516     int i, j, k, m, n, nnz;
00517     REAL value;
00518
00519     FILE* fp = fopen(filename, "r");
00520
00521     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00522
00523     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00524
00525     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00526
00527     if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
00528         fasp_chkerr(ERROR_WRONG_FILE, filename);
00529     }
00530
00531     dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);
00532
00533     for (k = 0; k < nnz; k++) {
00534         if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00535             Atmp.rowind[k] = i - 1;
00536             Atmp.colind[k] = j - 1;
00537             Atmp.val[k] = value;
00538         } else {
00539             fasp_chkerr(ERROR_WRONG_FILE, filename);
00540         }
00541     }
00542
00543     fclose(fp);
00544
00545     fasp_format_dcoo_dcsr(&Atmp, A);
00546     fasp_dcoo_free(&Atmp);
00547 }
00548
00567 void fasp_dmtx_read(const char* filename, dCSRmat* A)
00568 {
00569     int i, j, m, n, nnz;
00570     INT innz; // index of nonzeros
00571     REAL value;
00572
00573     FILE* fp = fopen(filename, "r");
00574
00575     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00576
00577     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00578
00579     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00580
00581     if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
00582         fasp_chkerr(ERROR_WRONG_FILE, filename);
00583     }
00584
00585     dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);
00586
00587     innz = 0;
00588
00589     while (innz < nnz) {
00590         if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00591             Atmp.rowind[innz] = i - 1;
00592             Atmp.colind[innz] = j - 1;
00593             Atmp.val[innz] = value;
00594             innz = innz + 1;
00595         } else {
00596             fasp_chkerr(ERROR_WRONG_FILE, filename);
00597         }
00598     }
00599 }

```

```

00599
00600     fclose(fp);
00601
00602     fasp_format_dcoo_dcsr(&Atmp, A);
00603     fasp_dcoo_free(&Atmp);
00604 }
00605
00624 void fasp_dmtxsym_read(const char* filename, dCSRmat* A)
00625 {
00626     int i, j, m, n, nnz;
00627     int innz; // index of nonzeros
00628     REAL value;
00629
00630     FILE* fp = fopen(filename, "r");
00631
00632     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00633
00634     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00635
00636     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00637
00638     if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
00639         fasp_chkerr(ERROR_WRONG_FILE, filename);
00640     }
00641
00642     nnz = 2 * (nnz - m) + m; // adjust for sym problem
00643     dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);
00644
00645     innz = 0;
00646
00647     while (innz < nnz) {
00648         if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00649
00650             if (i == j) {
00651                 Atmp.rowind[innz] = i - 1;
00652                 Atmp.colind[innz] = j - 1;
00653                 Atmp.val[innz] = value;
00654                 innz = innz + 1;
00655             } else {
00656                 Atmp.rowind[innz] = i - 1;
00657                 Atmp.rowind[innz + 1] = j - 1;
00658                 Atmp.colind[innz] = j - 1;
00659                 Atmp.colind[innz + 1] = i - 1;
00660                 Atmp.val[innz] = value;
00661                 Atmp.val[innz + 1] = value;
00662                 innz = innz + 2;
00663             }
00664             if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
00665                 fasp_chkerr(ERROR_WRONG_FILE, filename);
00666             }
00667         }
00668     }
00669     fclose(fp);
00670
00671     fasp_format_dcoo_dcsr(&Atmp, A);
00672     fasp_dcoo_free(&Atmp);
00673 }
00674
00699 void fasp_dstr_read(const char* filename, dSTRmat* A)
00700 {
00701     int nx, ny, nz, nxy, ngrid, nband, nc, offset;
00702     int i, k, n;
00703     REAL value;
00704
00705     FILE* fp = fopen(filename, "r");
00706
00707     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00708
00709     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00710
00711     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00712
00713     // read dimension of the problem
00714     if (fscanf(fp, "%d %d %d", &nx, &ny, &nz) > 0) {
00715         A->nx = nx;
00716         A->ny = ny;
00717         A->nz = nz;
00718     } else {
00719         fasp_chkerr(ERROR_WRONG_FILE, filename);
00720     }
00721 }

```

```

00722     nxy      = nx * ny;
00723     ngrid    = nxy * nz;
00724     A->nxy    = nxy;
00725     A->ngrid  = ngrid;
00726
00727     // read number of components
00728     if (fscanf(fp, "%d", &nc) > 0)
00729         A->nc = nc;
00730     else {
00731         fasp_chkerr(ERROR_WRONG_FILE, filename);
00732     }
00733
00734     // read number of bands
00735     if (fscanf(fp, "%d", &nband) > 0)
00736         A->nband = nband;
00737     else {
00738         fasp_chkerr(ERROR_WRONG_FILE, filename);
00739     }
00740
00741     A->offsets = (INT*)fasp_mem_calloc(nband, sizeof(INT));
00742
00743     // read diagonal
00744     if (fscanf(fp, "%d", &n) > 0) {
00745         A->diag = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
00746     } else {
00747         fasp_chkerr(ERROR_WRONG_FILE, filename);
00748     }
00749
00750     for (i = 0; i < n; ++i) {
00751         if (fscanf(fp, "%le", &value) > 0)
00752             A->diag[i] = value;
00753         else {
00754             fasp_chkerr(ERROR_WRONG_FILE, filename);
00755         }
00756     }
00757
00758     // read offdiags
00759     k      = nband;
00760     A->offdiag = (REAL**)fasp_mem_calloc(nband, sizeof(REAL*));
00761     while (k-- > 0) {
00762         // read number band k
00763         if (fscanf(fp, "%d %d", &offset, &n) > 0) {
00764             A->offsets[nband - k - 1] = offset;
00765         } else {
00766             fasp_chkerr(ERROR_WRONG_FILE, filename);
00767         }
00768
00769         A->offdiag[nband - k - 1] = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
00770         for (i = 0; i < n; ++i) {
00771             if (fscanf(fp, "%le", &value) > 0) {
00772                 A->offdiag[nband - k - 1][i] = value;
00773             } else {
00774                 fasp_chkerr(ERROR_WRONG_FILE, filename);
00775             }
00776         }
00777     }
00778
00779     fclose(fp);
00780 }
00781
00807 void fasp_dbsr_read(const char* filename, DBSRmat* A)
00808 {
00809     int    ROW, COL, NNZ, nb, storage_manner;
00810     int    i, n;
00811     int    index;
00812     REAL   value;
00813     size_t status;
00814
00815     FILE* fp = fopen(filename, "r");
00816
00817     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00818
00819     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00820
00821     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00822
00823     status = fscanf(fp, "%d %d %d", &ROW, &COL, &NNZ); // dimensions of the problem
00824     fasp_chkerr(status, filename);
00825     A->ROW = ROW;
00826     A->COL = COL;
00827     A->NNZ = NNZ;

```

```

00828
00829     status = fscanf(fp, "%d", &nb); // read the size of each block
00830     fasp_chkerr(status, filename);
00831     A->nb = nb;
00832
00833     status = fscanf(fp, "%d", &storage_manner); // read the storage_manner
00834     fasp_chkerr(status, filename);
00835     A->storage_manner = storage_manner;
00836
00837     // allocate memory space
00838     fasp_dbsr_alloc(ROW, COL, NNZ, nb, storage_manner, A);
00839
00840     // read IA
00841     status = fscanf(fp, "%d", &n);
00842     fasp_chkerr(status, filename);
00843     for (i = 0; i < n; ++i) {
00844         status = fscanf(fp, "%d", &index);
00845         fasp_chkerr(status, filename);
00846         A->IA[i] = index;
00847     }
00848
00849     // read JA
00850     status = fscanf(fp, "%d", &n);
00851     fasp_chkerr(status, filename);
00852     for (i = 0; i < n; ++i) {
00853         status = fscanf(fp, "%d", &index);
00854         fasp_chkerr(status, filename);
00855         A->JA[i] = index;
00856     }
00857
00858     // read val
00859     status = fscanf(fp, "%d", &n);
00860     fasp_chkerr(status, filename);
00861     for (i = 0; i < n; ++i) {
00862         status = fscanf(fp, "%le", &value);
00863         fasp_chkerr(status, filename);
00864         A->val[i] = value;
00865     }
00866
00867     fclose(fp);
00868 }
00869
00887 void fasp_dvecind_read(const char* filename, dvector* b)
00888 {
00889     int    i, n, index;
00890     REAL   value;
00891     size_t status;
00892
00893     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00894
00895     FILE* fp = fopen(filename, "r");
00896
00897     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00898
00899     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00900
00901     status = fscanf(fp, "%d", &n);
00902     fasp_dvec_alloc(n, b);
00903
00904     for (i = 0; i < n; ++i) {
00905
00906         status = fscanf(fp, "%d %le", &index, &value);
00907
00908         if (value > BIGREAL || index >= n) {
00909             fasp_dvec_free(b);
00910             fclose(fp);
00911
00912             printf("### ERROR: Wrong index = %d or value = %lf\n", index, value);
00913             fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
00914         }
00915
00916         b->val[index] = value;
00917     }
00918
00919     fclose(fp);
00920     fasp_chkerr(status, filename);
00921 }
00922
00938 void fasp_dvec_read(const char* filename, dvector* b)
00939 {
00940     int    i, n;

```

```

00941     REAL    value;
00942     size_t  status;
00943
00944     FILE* fp = fopen(filename, "r");
00945
00946     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00947
00948     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00949
00950     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
00951
00952     status = fscanf(fp, "%d", &n);
00953
00954     fasp_dvec_alloc(n, b);
00955
00956     for (i = 0; i < n; ++i) {
00957
00958         status = fscanf(fp, "%le", &value);
00959         b->val[i] = value;
00960
00961         if (value > BIGREAL) {
00962             fasp_dvec_free(b);
00963             fclose(fp);
00964
00965             printf("### ERROR: Wrong value = %lf!\n", value);
00966             fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
00967         }
00968     }
00969
00970     fclose(fp);
00971     fasp_chkerr(status, filename);
00972 }
00973
00989 void fasp_ivecind_read(const char* filename, ivector* b)
00990 {
00991     int    i, n, index, value;
00992     size_t status;
00993
00994     FILE* fp = fopen(filename, "r");
00995
00996     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
00997
00998     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
00999
01000     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
01001
01002     status = fscanf(fp, "%d", &n);
01003     fasp_ivec_alloc(n, b);
01004
01005     for (i = 0; i < n; ++i) {
01006         status = fscanf(fp, "%d %d", &index, &value);
01007         b->val[index] = value;
01008     }
01009
01010     fclose(fp);
01011     fasp_chkerr(status, filename);
01012 }
01013
01029 void fasp_ivec_read(const char* filename, ivector* b)
01030 {
01031     int    i, n, value;
01032     size_t status;
01033
01034     FILE* fp = fopen(filename, "r");
01035
01036     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01037
01038     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
01039
01040     skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020
01041
01042     status = fscanf(fp, "%d", &n);
01043     fasp_ivec_alloc(n, b);
01044
01045     for (i = 0; i < n; ++i) {
01046         status = fscanf(fp, "%d", &value);
01047         b->val[i] = value;
01048     }
01049
01050     fclose(fp);
01051     fasp_chkerr(status, filename);

```

```

01052 }
01053
01079 void fasp_dcsrvec_writel(const char* filename, dCSRmat* A, dvector* b)
01080 {
01081     INT m = A->row, n = A->col, nnz = A->nnz;
01082     INT i;
01083
01084     FILE* fp = fopen(filename, "w");
01085
01086     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01087
01088     /* write the matrix to file */
01089     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
01090
01091     fprintf(fp, "%d %d\n", m, n);
01092     for (i = 0; i < m + 1; ++i) {
01093         fprintf(fp, "%d\n", A->IA[i]);
01094     }
01095     for (i = 0; i < nnz; ++i) {
01096         fprintf(fp, "%d\n", A->JA[i]);
01097     }
01098     for (i = 0; i < nnz; ++i) {
01099         fprintf(fp, "%le\n", A->val[i]);
01100     }
01101
01102     m = b->row;
01103
01104     /* write the rhs to file */
01105     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01106
01107     fprintf(fp, "%d\n", m);
01108
01109     for (i = 0; i < m; ++i) fprintf(fp, "%le\n", b->val[i]);
01110
01111     fclose(fp);
01112 }
01113
01145 void fasp_dcsrvec_write2(const char* filemat, const char* filerhs, dCSRmat* A,
01146                         dvector* b)
01147 {
01148     INT m = A->row, nnz = A->nnz;
01149     INT i;
01150
01151     FILE* fp = fopen(filemat, "w");
01152
01153     /* write the matrix to file */
01154     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filemat);
01155
01156     printf("%s: writing to file %s ...\n", __FUNCTION__, filemat);
01157
01158     fprintf(fp, "%d\n", m);
01159     for (i = 0; i < m + 1; ++i) {
01160         fprintf(fp, "%d\n", A->IA[i] + 1);
01161     }
01162     for (i = 0; i < nnz; ++i) {
01163         fprintf(fp, "%d\n", A->JA[i] + 1);
01164     }
01165     for (i = 0; i < nnz; ++i) {
01166         fprintf(fp, "%le\n", A->val[i]);
01167     }
01168
01169     fclose(fp);
01170
01171     m = b->row;
01172
01173     fp = fopen(filerhs, "w");
01174
01175     /* write the rhs to file */
01176     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);
01177
01178     printf("%s: writing to file %s ...\n", __FUNCTION__, filerhs);
01179
01180     fprintf(fp, "%d\n", m);
01181
01182     for (i = 0; i < m; ++i) fprintf(fp, "%le\n", b->val[i]);
01183
01184     fclose(fp);
01185 }
01186
01207 void fasp_dcoo_write(const char* filename, dCSRmat* A)
01208 {

```



```

01209     const INT m = A->row, n = A->col;
01210     INT      i, j;
01211
01212     FILE* fp = fopen(filename, "w");
01213
01214     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01215
01216     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01217
01218     fprintf(fp, "%d %d %d\n", m, n, A->nnz);
01219     for (i = 0; i < m; ++i) {
01220         for (j = A->IA[i]; j < A->IA[i + 1]; j++)
01221             fprintf(fp, "%d %d %0.15e\n", i, A->JA[j], A->val[j]);
01222     }
01223
01224     fclose(fp);
01225 }
01226
01241 void fasp_dstr_write(const char* filename, dSTRmat* A)
01242 {
01243     const INT nx = A->nx, ny = A->ny, nz = A->nz;
01244     const INT ngrid = A->ngrid, nband = A->nband, nc = A->nc;
01245
01246     INT* offsets = A->offsets;
01247
01248     INT i, k, n;
01249
01250     FILE* fp = fopen(filename, "w");
01251
01252     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01253
01254     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01255
01256     fprintf(fp, "%d %d %d\n", nx, ny, nz); // write dimension of the problem
01257
01258     fprintf(fp, "%d\n", nc); // read number of components
01259
01260     fprintf(fp, "%d\n", nband); // write number of bands
01261
01262     // write diagonal
01263     n = ngrid * nc * nc; // number of nonzeros in each band
01264     fprintf(fp, "%d\n", n); // number of diagonal entries
01265     for (i = 0; i < n; ++i) fprintf(fp, "%le\n", A->diag[i]);
01266
01267     // write offdiags
01268     k = nband;
01269     while (k--) {
01270         INT offset = offsets[nband - k - 1];
01271         n = (ngrid - ABS(offset)) * nc * nc; // number of nonzeros in each band
01272         fprintf(fp, "%d %d\n", offset, n); // read number band k
01273         for (i = 0; i < n; ++i) {
01274             fprintf(fp, "%le\n", A->offdiag[nband - k - 1][i]);
01275         }
01276     }
01277
01278     fclose(fp);
01279 }
01280
01292 void fasp_dbsr_print(const char* filename, dBSRmat* A)
01293 {
01294     const INT ROW = A->ROW;
01295     const INT nb = A->nb;
01296     const INT nb2 = nb * nb;
01297
01298     INT* ia = A->IA;
01299     INT* ja = A->JA;
01300     REAL* val = A->val;
01301
01302     INT i, j, k, ind;
01303
01304     FILE* fp = fopen(filename, "w");
01305
01306     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01307
01308     printf("%s: printing to file %s ...\n", __FUNCTION__, filename);
01309
01310     for (i = 0; i < ROW; i++) {
01311         for (k = ia[i]; k < ia[i + 1]; k++) {
01312             j = ja[k];
01313             fprintf(fp, "A[%d,%d]=\n", i, j);
01314             for (ind = 0; ind < nb2; ind++) {

```

```

01315         fprintf(fp, "%.10E ", val[k * nb2 + ind]);
01316     }
01317     fprintf(fp, "\n");
01318 }
01319 }
01320 }
01321
01336 void fasp_dbsr_write(const char* filename, dBSRmat* A)
01337 {
01338     const INT ROW = A->ROW, COL = A->COL, NNZ = A->NNZ;
01339     const INT nb = A->nb, storage_manner = A->storage_manner;
01340
01341     INT* ia = A->IA;
01342     INT* ja = A->JA;
01343     REAL* val = A->val;
01344
01345     INT i, n;
01346
01347     FILE* fp = fopen(filename, "w");
01348
01349     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01350
01351     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01352
01353     fprintf(fp, "%d %d %d\n", ROW, COL, NNZ); // write dimension of the block matrix
01354
01355     fprintf(fp, "%d\n", nb); // write the size of each block
01356
01357     fprintf(fp, "%d\n", storage_manner); // write storage manner of each block
01358
01359     // write A->IA
01360     n = ROW + 1; // length of A->IA
01361     fprintf(fp, "%d\n", n); // length of A->IA
01362     for (i = 0; i < n; ++i) fprintf(fp, "%d\n", ia[i]);
01363
01364     // write A->JA
01365     n = NNZ; // length of A->JA
01366     fprintf(fp, "%d\n", n); // length of A->JA
01367     for (i = 0; i < n; ++i) fprintf(fp, "%d\n", ja[i]);
01368
01369     // write A->val
01370     n = NNZ * nb * nb; // length of A->val
01371     fprintf(fp, "%d\n", n); // length of A->val
01372     for (i = 0; i < n; ++i) fprintf(fp, "%le\n", val[i]);
01373
01374     fclose(fp);
01375 }
01376
01388 void fasp_dvec_write(const char* filename, dvector* vec)
01389 {
01390     INT m = vec->row, i;
01391
01392     FILE* fp = fopen(filename, "w");
01393
01394     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01395
01396     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01397
01398     fprintf(fp, "%d\n", m);
01399
01400     for (i = 0; i < m; ++i) fprintf(fp, "%0.15e\n", vec->val[i]);
01401
01402     fclose(fp);
01403 }
01404
01420 void fasp_dvecind_write(const char* filename, dvector* vec)
01421 {
01422     INT m = vec->row, i;
01423
01424     FILE* fp = fopen(filename, "w");
01425
01426     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01427
01428     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01429
01430     fprintf(fp, "%d\n", m);
01431
01432     for (i = 0; i < m; ++i) fprintf(fp, "%d %le\n", i, vec->val[i]);
01433
01434     fclose(fp);
01435 }

```

```

01436
01452 void fasp_ivec_write(const char* filename, ivector* vec)
01453 {
01454     INT m = vec->row, i;
01455
01456     FILE* fp = fopen(filename, "w");
01457
01458     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01459
01460     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01461
01462     // write number of nonzeros
01463     fprintf(fp, "%d\n", m);
01464
01465     // write index and value each line
01466     for (i = 0; i < m; ++i) fprintf(fp, "%d %d\n", i, vec->val[i] + 1);
01467
01468     fclose(fp);
01469 }
01470
01482 void fasp_dvec_print(const INT n, dvector* u)
01483 {
01484     INT i;
01485     INT NumPrint = n;
01486
01487     if (n <= 0) NumPrint = u->row; // print all
01488
01489     for (i = 0; i < NumPrint; ++i) printf("vec_%d = %15.10E\n", i, u->val[i]);
01490 }
01491
01503 void fasp_ivec_print(const INT n, ivector* u)
01504 {
01505     INT i;
01506     INT NumPrint = n;
01507
01508     if (n <= 0) NumPrint = u->row; // print all
01509
01510     for (i = 0; i < NumPrint; ++i) printf("vec_%d = %d\n", i, u->val[i]);
01511 }
01512
01523 void fasp_dcsr_print(const dCSRmat* A)
01524 {
01525     const INT m = A->row, n = A->col;
01526     INT i, j;
01527
01528     printf("nrow = %d, ncol = %d, nnz = %d\n", m, n, A->nnz);
01529     for (i = 0; i < m; ++i) {
01530         for (j = A->IA[i]; j < A->IA[i + 1]; j++)
01531             printf("A_(%d,%d) = %+.10E\n", i, A->JA[j], A->val[j]);
01532     }
01533 }
01534
01545 void fasp_dcoo_print(const dCOOmat* A)
01546 {
01547     INT k;
01548
01549     printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
01550     for (k = 0; k < A->nnz; k++) {
01551         printf("A_(%d,%d) = %+.10E\n", A->rowind[k], A->colind[k], A->val[k]);
01552     }
01553 }
01554
01568 void fasp_dbsr_write_coo(const char* filename, const dBSRmat* A)
01569 {
01570
01571     INT i, j, k, l;
01572     INT nb, nb2;
01573     nb = A->nb;
01574     nb2 = nb * nb;
01575
01576     FILE* fp = fopen(filename, "w");
01577
01578     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01579
01580 #if DEBUG_MODE > PRINT_MIN
01581     printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d, nb = %d\n", A->ROW, A->COL,
01582           A->NNZ, A->nb);
01583     printf("### DEBUG: storage_manner = %d\n", A->storage_manner);
01584 #endif
01585
01586     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);

```

```

01587
01588 // write dimension of the block matrix
01589 fprintf(fp, "%d %d %d\n", A->ROW,
01590     A->COL, A->NNZ);
01591 // write the size of each block
01592 fprintf(fp, "%d\n", A->nb);
01593 // write storage manner of each block
01594 fprintf(fp, "%d\n", A->storage_manner);
01595
01596 for (i = 0; i < A->ROW; i++) {
01597     for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
01598         for (k = 0; k < A->nb; k++) {
01599             for (l = 0; l < A->nb; l++) {
01600                 fprintf(fp, "%d %d %.10E\n", i * nb + k + 1, A->JA[j] * nb + l + 1,
01601                     A->val[j * nb2 + k * nb + l]);
01602             }
01603         }
01604     }
01605 }
01606
01607 fclose(fp);
01608 }
01609
01623 void fasp_dcsr_write_coo(const char* filename, const dCSRmat* A)
01624 {
01625     INT i, j;
01626
01627     #if DEBUG_MODE > PRINT_MIN
01628     printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
01629     #endif
01630
01631     FILE* fp = fopen(filename, "w");
01632
01633     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01634
01635     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01636
01637     // write dimension of the matrix
01638     fprintf(fp, "%d %d %d\n", A->row, A->col,
01639         A->nnz);
01640
01641     for (i = 0; i < A->row; i++) {
01642         for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
01643             fprintf(fp, "%d %d %.15E\n", i + 1, A->JA[j] + 1, A->val[j]);
01644         }
01645     }
01646
01647     fclose(fp);
01648 }
01649
01650
01664 void fasp_dcsr_write_mtx(const char* filename, const dCSRmat* A)
01665 {
01666     INT i, j;
01667
01668     #if DEBUG_MODE > PRINT_MIN
01669     printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
01670     #endif
01671
01672     FILE* fp = fopen(filename, "w");
01673
01674     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01675
01676     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01677
01678     // write dimension of the matrix
01679     fprintf(fp, "%d %d %d\n", A->row, A->col, A->nnz);
01680
01681     for (i = 0; i < A->row; i++) {
01682         for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
01683             fprintf(fp, "%d %d %.15E\n", i + 1, A->JA[j] + 1, A->val[j]);
01684         }
01685     }
01686
01687     fclose(fp);
01688 }
01689
01690
01701 void fasp_dstr_print(const dSTRmat* A)
01702 {
01703     // TODO: To be added later! --Chensong

```

```

01704 }
01705
01735 void fasp_matrix_read(const char* filename, void* A)
01736 {
01737     int    index, flag;
01738     SHORT  EndianFlag;
01740     size_t status;
01741
01742     FILE* fp = fopen(filename, "rb");
01743
01744     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01745
01746     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
01747
01748     status = fread(&index, sizeof(INT), 1, fp);
01749     fasp_chkerr(status, filename);
01750
01751     // matrix stored in ASCII format
01752     if (index == 808464432) {
01753
01754         fclose(fp);
01755         fp = fopen(filename, "r"); // reopen file of reading file in ASCII
01756
01757         status = fscanf(fp, "%d\n", &flag); // jump over the first line
01758         fasp_chkerr(status, __FUNCTION__);
01759
01760         status = fscanf(fp, "%d\n", &flag); // reading the format information
01761         fasp_chkerr(status, __FUNCTION__);
01762
01763         flag = (INT)flag / 100;
01764
01765         switch (flag) {
01766             case 0:
01767                 fasp_dcsr_read_s(fp, (dCSRmat*)A);
01768                 break;
01769             case 1:
01770                 fasp_dcoo_read_s(fp, (dCSRmat*)A);
01771                 break;
01772             case 2:
01773                 fasp_dbsr_read_s(fp, (dBSRmat*)A);
01774                 break;
01775             case 3:
01776                 fasp_dstr_read_s(fp, (dSTRmat*)A);
01777                 break;
01778             case 4:
01779                 fasp_dcoo_read_s(fp, (dCSRmat*)A);
01780                 break;
01781             case 5:
01782                 fasp_dmtx_read_s(fp, (dCSRmat*)A);
01783                 break;
01784             case 6:
01785                 fasp_dmtxsym_read_s(fp, (dCSRmat*)A);
01786                 break;
01787             default:
01788                 printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
01789                 fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
01790         }
01791
01792         fclose(fp);
01793         return;
01794     }
01795
01796     // matrix stored in binary format
01797
01798     // test Endian consistence of machine and file
01799     EndianFlag = index;
01800
01801     status = fread(&index, sizeof(INT), 1, fp);
01802     fasp_chkerr(status, filename);
01803
01804     index = endian_convert_int(index, sizeof(INT), EndianFlag);
01805     flag = (INT)index / 100;
01806     ilength = (INT)(index - flag * 100) / 10;
01807     dlength = index % 10;
01808
01809     switch (flag) {
01810         case 1:
01811             fasp_dcsr_read_b(fp, (dCSRmat*)A, EndianFlag);
01812             break;
01813         case 2:

```

```

01814         fasp_dbsr_read_b(fp, (dBSRmat*)A, EndianFlag);
01815         break;
01816     case 3:
01817         fasp_dstr_read_b(fp, (dSTRmat*)A, EndianFlag);
01818         break;
01819     case 4:
01820         fasp_dcoo_read_b(fp, (dCSRmat*)A, EndianFlag);
01821         break;
01822     case 5:
01823         fasp_dmtx_read_b(fp, (dCSRmat*)A, EndianFlag);
01824         break;
01825     case 6:
01826         fasp_dmtxsym_read_b(fp, (dCSRmat*)A, EndianFlag);
01827         break;
01828     default:
01829         printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
01830         fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
01831     }
01832 }
01833 fclose(fp);
01834 }
01835
01849 void fasp_matrix_read_bin(const char* filename, void* A)
01850 {
01851     int    index, flag;
01852     SHORT  EndianFlag = 1;
01853     size_t status;
01854
01855     FILE* fp = fopen(filename, "rb");
01856
01857     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01858
01859     printf("%s: reading file %s ...\n", __FUNCTION__, filename);
01860
01861     status = fread(&index, sizeof(INT), 1, fp);
01862     fasp_chkerr(status, filename);
01863
01864     index = endian_convert_int(index, sizeof(INT), EndianFlag);
01865
01866     flag    = (INT)index / 100;
01867     ilength = (int)(index - flag * 100) / 10;
01868     dlength = index % 10;
01869
01870     switch (flag) {
01871     case 1:
01872         fasp_dcoo_read_b(fp, (dCSRmat*)A, EndianFlag);
01873         break;
01874     case 2:
01875         fasp_dbsr_read_b(fp, (dBSRmat*)A, EndianFlag);
01876         break;
01877     case 3:
01878         fasp_dstr_read_b(fp, (dSTRmat*)A, EndianFlag);
01879         break;
01880     case 4:
01881         fasp_dcsr_read_b(fp, (dCSRmat*)A, EndianFlag);
01882         break;
01883     case 5:
01884         fasp_dmtx_read_b(fp, (dCSRmat*)A, EndianFlag);
01885         break;
01886     case 6:
01887         fasp_dmtxsym_read_b(fp, (dCSRmat*)A, EndianFlag);
01888         break;
01889     default:
01890         printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
01891         fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
01892     }
01893
01894     fclose(fp);
01895 }
01896
01921 void fasp_matrix_write(const char* filename, void* A, const INT flag)
01922 {
01923     INT    fileflag, matrixflag;
01924     FILE* fp;
01925
01926     matrixflag = flag % 100;
01927     fileflag   = (INT)flag / 100;
01928
01929     // write matrix in ASCII file
01930     if (!fileflag) {
01931

```

```

01932     fp = fopen(filename, "w");
01933
01934     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01935
01936     printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01937
01938     fprintf(fp, "%d%d%d\n", fileflag, fileflag, fileflag, fileflag);
01939
01940     fprintf(fp, "%d%d%d\n", matrixflag, (int)sizeof(INT), (int)sizeof(REAL));
01941
01942     switch (matrixflag) {
01943     case 1:
01944         fasp_dcsr_write_s(fp, (dCSRmat*)A);
01945         break;
01946     case 2:
01947         fasp_dbsr_write_s(fp, (dBSRmat*)A);
01948         break;
01949     case 3:
01950         fasp_dstr_write_s(fp, (dSTRmat*)A);
01951         break;
01952     default:
01953         printf("### WARNING: Unknown matrix flag %d\n", matrixflag);
01954     }
01955     fclose(fp);
01956     return;
01957 }
01958
01959 // write matrix in binary file
01960 fp = fopen(filename, "wb");
01961
01962 if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);
01963
01964 printf("%s: writing to file %s ...\n", __FUNCTION__, filename);
01965
01966 INT putflag = fileflag * 100 + sizeof(INT) * 10 + sizeof(REAL);
01967 fwrite(&putflag, sizeof(INT), 1, fp);
01968
01969 switch (matrixflag) {
01970 case 1:
01971     fasp_dcsr_write_b(fp, (dCSRmat*)A);
01972     break;
01973 case 2:
01974     fasp_dbsr_write_b(fp, (dBSRmat*)A);
01975     break;
01976 case 3:
01977     fasp_dstr_write_b(fp, (dSTRmat*)A);
01978     break;
01979 default:
01980     printf("### WARNING: Unknown matrix flag %d\n", matrixflag);
01981 }
01982
01983 fclose(fp);
01984 }
01985
02011 void fasp_vector_read(const char* filerhs, void* b)
02012 {
02013     int    index, flag;
02014     SHORT  EndianFlag;
02015     size_t status;
02016
02017     FILE* fp = fopen(filerhs, "rb");
02018
02019     if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);
02020
02021     printf("%s: reading file %s ...\n", __FUNCTION__, filerhs);
02022
02023     status = fread(&index, sizeof(INT), 1, fp);
02024     fasp_chkerr(status, filerhs);
02025
02026     // vector stored in ASCII
02027     if (index == 808464432) {
02028
02029         fclose(fp);
02030         fp = fopen(filerhs, "r");
02031
02032         if (!fscanf(fp, "%d\n", &flag))
02033             printf("### ERROR: File format problem in %s!\n", __FUNCTION__);
02034         // TODO: Check why skip this flag ??? --Chensong
02035
02036         if (!fscanf(fp, "%d\n", &flag))
02037             printf("### ERROR: File format problem in %s!\n", __FUNCTION__);

```

```

02038     flag = (int)flag / 100;
02039
02040     switch (flag) {
02041     case 1:
02042         fasp_dvec_read_s(fp, (dvector*)b);
02043         break;
02044     case 2:
02045         fasp_ivec_read_s(fp, (ivector*)b);
02046         break;
02047     case 3:
02048         fasp_dvecind_read_s(fp, (dvector*)b);
02049         break;
02050     case 4:
02051         fasp_ivecind_read_s(fp, (ivector*)b);
02052         break;
02053     }
02054     fclose(fp);
02055     return;
02056 }
02057
02058 // vector stored in binary
02059 EndianFlag = index;
02060 status = fread(&index, sizeof(INT), 1, fp);
02061 fasp_chkerr(status, filerhs);
02062
02063 index = endian_convert_int(index, sizeof(INT), EndianFlag);
02064 flag = (int)index / 100;
02065 ilength = (int)(index - 100 * flag) / 10;
02066 dlength = index % 10;
02067
02068 switch (flag) {
02069 case 1:
02070     fasp_dvec_read_b(fp, (dvector*)b, EndianFlag);
02071     break;
02072 case 2:
02073     fasp_ivec_read_b(fp, (ivector*)b, EndianFlag);
02074     break;
02075 case 3:
02076     fasp_dvecind_read_b(fp, (dvector*)b, EndianFlag);
02077     break;
02078 case 4:
02079     fasp_ivecind_read_b(fp, (ivector*)b, EndianFlag);
02080     break;
02081 default:
02082     printf("### ERROR: Unknown flag %d in %s!\n", flag, filerhs);
02083     fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
02084 }
02085
02086 fclose(fp);
02087 }
02088
02119 void fasp_vector_write(const char* filerhs, void* b, const INT flag)
02120 {
02121
02122     INT fileflag, vectorflag;
02123     FILE* fp;
02124
02125     fileflag = (int)flag / 10;
02126     vectorflag = (int)flag % 10;
02127
02128     // write vector in ASCII
02129     if (!fileflag) {
02130         fp = fopen(filerhs, "w");
02131
02132         if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);
02133
02134         printf("%s: writing to file %s ...\n", __FUNCTION__, filerhs);
02135
02136         fprintf(fp, "%d%d%d\n", fileflag, fileflag, fileflag);
02137
02138         fprintf(fp, "%d%d%d\n", vectorflag, (int)sizeof(INT), (int)sizeof(REAL));
02139
02140         switch (vectorflag) {
02141         case 1:
02142             fasp_dvec_write_s(fp, (dvector*)b);
02143             break;
02144         case 2:
02145             fasp_ivec_write_s(fp, (ivector*)b);
02146             break;
02147         case 3:
02148             fasp_dvecind_write_s(fp, (dvector*)b);

```



```

02149         break;
02150     case 4:
02151         fasp_ivecind_write_s(fp, (ivector*)b);
02152         break;
02153     default:
02154         printf("### WARNING: Unknown vector flag %d\n", vectorflag);
02155     }
02156
02157     fclose(fp);
02158     return;
02159 }
02160
02161 // write vector in binary
02162 fp = fopen(filerhs, "wb");
02163
02164 if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);
02165
02166 printf("%s: writing to file %s ...\n", __FUNCTION__, filerhs);
02167
02168 INT putflag = vectorflag * 100 + sizeof(INT) * 10 + sizeof(REAL);
02169 fwrite(&putflag, sizeof(INT), 1, fp);
02170
02171 switch (vectorflag) {
02172     case 1:
02173         fasp_dvec_write_b(fp, (dvector*)b);
02174         break;
02175     case 2:
02176         fasp_ivec_write_b(fp, (ivector*)b);
02177         break;
02178     case 3:
02179         fasp_dvecind_write_b(fp, (dvector*)b);
02180         break;
02181     case 4:
02182         fasp_ivecind_write_b(fp, (ivector*)b);
02183         break;
02184     default:
02185         printf("### WARNING: Unknown vector flag %d\n", vectorflag);
02186 }
02187
02188 fclose(fp);
02189 }
02190
02206 void fasp_hb_read(const char* input_file, dCSRmat* A, dvector* b)
02207 {
02208     //-----
02209     // Setup local variables
02210     //-----
02211     // variables for FASP
02212     dCSRmat tempA;
02213
02214     // variables for hb_io
02215
02216     int* colptr = NULL;
02217     double* exact = NULL;
02218     double* guess = NULL;
02219     int i;
02220     int indcrd;
02221     char* indfmt = NULL;
02222     FILE* input;
02223     int j;
02224     char* key = NULL;
02225     char* mxtype = NULL;
02226     int ncol;
02227     int neltvl;
02228     int nnzero;
02229     int nrhs;
02230     int nrhsix;
02231     int nrow;
02232     int ptrcrd;
02233     char* ptrfmt = NULL;
02234     int rhscrd;
02235     char* rhsfmt = NULL;
02236     int* rhsind = NULL;
02237     int* rhsptr = NULL;
02238     char* rhstyp = NULL;
02239     double* rhsval = NULL;
02240     double* rhsvec = NULL;
02241     int* rowind = NULL;
02242     char* title = NULL;
02243     int totcrd;
02244     int valcrd;

```

```

02245     char*   valfmt = NULL;
02246     double* values = NULL;
02247
02248     printf("\n");
02249     printf("HB_FILE_READ reads all the data in an HB file.\n");
02250
02251     printf("\n");
02252     printf("Reading the file '%s'\n", input_file);
02253
02254     input = fopen(input_file, "rt");
02255
02256     if (!input) {
02257         printf("### ERROR: Fail to open the file [%s]\n", input_file);
02258         fasp_chkerf(ERROR_OPEN_FILE, __FUNCTION__);
02259     }
02260
02261     //-----
02262     // Reading...
02263     //-----
02264     hb_file_read(input, &title, &key, &totcrd, &ptrcrd, &indcrd, &valcrd, &rhscrd,
02265                  &mxtyp, &nrow, &ncol, &nnzero, &neltvl, &ptrfmt, &indfmt, &valfmt,
02266                  &rhsfmt, &rhstyp, &nrhs, &nrhsix, &colptr, &rowind, &values, &rhsval,
02267                  &rhsptr, &rhsind, &rhsvec, &guess, &exact);
02268
02269     //-----
02270     // Printing if needed
02271     //-----
02272     #if DEBUG_MODE > PRINT_MIN
02273     /*
02274     Print out the header information.
02275     */
02276     hb_header_print(title, key, totcrd, ptrcrd, indcrd, valcrd, rhscrd, mxtyp, nrow,
02277                    ncol, nnzero, neltvl, ptrfmt, indfmt, valfmt, rhsfmt, rhstyp, nrhs,
02278                    nrhsix);
02279     /*
02280     Print the structure information.
02281     */
02282     hb_structure_print(ncol, mxtyp, nnzero, neltvl, colptr, rowind);
02283
02284     /*
02285     Print the values.
02286     */
02287     hb_values_print(ncol, colptr, mxtyp, nnzero, neltvl, values);
02288
02289     if (0 < rhscrd) {
02290         /*
02291         Print a bit of the right hand sides.
02292         */
02293         if (rhstyp[0] == 'F') {
02294             r8mat_print_some(nrow, nrhs, rhsval, 1, 1, 5, 5, " Part of RHS");
02295         } else if (rhstyp[0] == 'M' && mxtyp[2] == 'A') {
02296             i4vec_print_part(nrhs + 1, rhsptr, 10, " Part of RHSPTR");
02297             i4vec_print_part(nrhsix, rhsind, 10, " Part of RHSIND");
02298             r8vec_print_part(nrhsix, rhsvec, 10, " Part of RHSVEC");
02299         } else if (rhstyp[0] == 'M' && mxtyp[2] == 'E') {
02300             r8mat_print_some(nnzero, nrhs, rhsval, 1, 1, 5, 5, " Part of RHS");
02301         }
02302         /*
02303         Print a bit of the starting guesses.
02304         */
02305         if (rhstyp[1] == 'G') {
02306             r8mat_print_some(nrow, nrhs, guess, 1, 1, 5, 5, " Part of GUESS");
02307         }
02308         /*
02309         Print a bit of the exact solutions.
02310         */
02311         if (rhstyp[2] == 'X') {
02312             r8mat_print_some(nrow, nrhs, exact, 1, 1, 5, 5, " Part of EXACT");
02313         }
02314     }
02315     #endif
02316
02317     //-----
02318     // Closing
02319     //-----
02320     fclose(input);
02321
02322     //-----
02323     // Convert to FASP format
02324     //-----
02325

```

```

02326 // convert matrix
02327 if (ncol != nrow) {
02328     printf("### ERROR: The matrix is not square! [%s]\n", __FUNCTION__);
02329     goto FINISHED;
02330 }
02331
02332 tempA = fasp_dcsr_create(nrow, ncol, nnzero);
02333
02334 for (i = 0; i <= ncol; i++) tempA.IA[i] = colptr[i] - 1;
02335 for (i = 0; i < nnzero; i++) tempA.JA[i] = rowind[i] - 1;
02336 fasp_darray_cp(nnzero, values, tempA.val);
02337
02338 // if the matrix is symmetric
02339 if (mxtype[1] == 'S') {
02340
02341     // A = A' + A
02342     dCSRmat tempA_tran;
02343     fasp_dcsr_trans(&tempA, &tempA_tran);
02344     fasp_blas_dcsr_add(&tempA, 1.0, &tempA_tran, 1.0, A);
02345     fasp_dcsr_free(&tempA);
02346     fasp_dcsr_free(&tempA_tran);
02347
02348     // modify diagonal entries
02349     for (i = 0; i < A->row; i++) {
02350
02351         for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
02352
02353             if (A->JA[j] == i) {
02354                 A->val[j] = A->val[j] / 2;
02355                 break;
02356             }
02357         }
02358     }
02359 }
02360 // if the matrix is not symmetric
02361 else {
02362     fasp_dcsr_trans(&tempA, A);
02363     fasp_dcsr_free(&tempA);
02364 }
02365
02366 // convert right hand side
02367
02368 if (nrhs == 0) {
02369
02370     printf("### ERROR: No right hand side! [%s]\n", __FUNCTION__);
02371     goto FINISHED;
02372 } else if (nrhs > 1) {
02373
02374     printf("### ERROR: More than one right hand side! [%s]\n", __FUNCTION__);
02375     goto FINISHED;
02376 } else {
02377
02378     fasp_dvec_alloc(nrow, b);
02379     fasp_darray_cp(nrow, rhsval, b->val);
02380 }
02381
02382 //-----
02383 // Cleanning
02384 //-----
02385 FINISHED:
02386 if (colptr) free(colptr);
02387 if (exact) free(exact);
02388 if (guess) free(guess);
02389 if (rhsind) free(rhsind);
02390 if (rhsptr) free(rhsptr);
02391 if (rhsval) free(rhsval);
02392 if (rhsvec) free(rhsvec);
02393 if (rowind) free(rowind);
02394 if (values) free(values);
02395
02396 return;
02397 }
02398
02399 /*-----*/
02400 /*--          End of File          --*/
02401 /*-----*/

```

9.61 BlaOrderingCSR.c File Reference

Generating ordering using algebraic information.

```
#include "fasp.h"
```

Functions

- void [fasp_dcsr_CMK_order](#) (const [dCSRmat](#) *A, [INT](#) *order, [INT](#) *oindex)
Ordering vertices of matrix graph corresponding to A.
- void [fasp_dcsr_RCMK_order](#) (const [dCSRmat](#) *A, [INT](#) *order, [INT](#) *oindex, [INT](#) *rorder)
Reverse CMK ordering.

9.61.1 Detailed Description

Generating ordering using algebraic information.

Note

This file contains Level-1 (Bla) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaOrderingCSR.c](#).

9.61.2 Function Documentation

9.61.2.1 [fasp_dcsr_CMK_order\(\)](#)

```
void fasp_dcsr_CMK_order (
    const dCSRmat * A,
    INT * order,
    INT * oindex )
```

Ordering vertices of matrix graph corresponding to A.

Parameters

<i>A</i>	Pointer to matrix
<i>oindex</i>	Pointer to index of vertices in order
<i>order</i>	Pointer to vertices with increasing degree

Author

Zheng Li, Chensong Zhang

Date

05/28/2014

Definition at line 37 of file [BlaOrderingCSR.c](#).

9.61.2.2 fasp_dcsr_RCMK_order()

```
void fasp_dcsr_RCMK_order (
    const dCSRmat * A,
    INT * order,
    INT * oindex,
    INT * rorder )
```

Reverse CMK ordering.

Parameters

<i>A</i>	Pointer to matrix
<i>order</i>	Pointer to vertices with increasing degree
<i>oindex</i>	Pointer to index of vertices in order
<i>rorder</i>	Pointer to reverse order

Author

Zheng Li, Chensong Zhang

Date

10/10/2014

Definition at line 87 of file [BlaOrderingCSR.c](#).

9.62 BlaOrderingCSR.c

[Go to the documentation of this file.](#)

```
00001
00013 #include "fasp.h"
00014
00015 /*-----*/
00016 /*--  Declare Private Functions  --*/
00017 /*-----*/
00018
00019 static void CMK_ordering (const dCSRmat *, INT, INT, INT, INT, INT *, INT *);
00020
00021 /*-----*/
00022 /*--      Public Functions      --*/
00023 /*-----*/
00024
00037 void fasp_dcsr_CMK_order (const dCSRmat *A,
00038                          INT *order,
00039                          INT *oindex)
00040 {
00041     const INT *ia = A->IA;
00042     const INT row = A->row;
00043
00044     INT i, loc, s, vt, mindg, innz;
00045
00046     s = 0;
00047     vt = 0;
00048     mindg = row+1;
00049
00050     // select node with minimal degree
00051     for (i=0; i<row; ++i) {
00052         innz = ia[i+1] - ia[i];
00053         if (innz > 1) {
00054             oindex[i] = -innz;
00055             if (innz < mindg) {
00056                 mindg = innz;
00057                 vt = i;
00058             }
00059         }
00060         else { // order those diagonal rows first
```

```

00061         oindex[i] = s;
00062         order[s] = i;
00063         s ++;
00064     }
00065 }
00066
00067 loc = s;
00068
00069 // start to order
00070 CMK_ordering (A, loc, s, vt, mindg, oindex, order);
00071 }
00072
00087 void fasp_dcsr_RCMK_order (const dCSRmat *A,
00088                             INT          *order,
00089                             INT          *oindex,
00090                             INT          *rorder)
00091 {
00092     INT i;
00093     INT row = A->row;
00094
00095     // Form CMK order
00096     fasp_dcsr_CMK_order(A, order, oindex);
00097
00098     // Reverse CMK order
00099     for (i=0; i<row; ++i) rorder[i] = order[row-1-i];
00100 }
00101
00102 /*-----*/
00103 /*--      Private Functions      --*/
00104 /*-----*/
00105
00123 static void CMK_ordering (const dCSRmat *A,
00124                           INT          loc,
00125                           INT          s,
00126                           INT          jj,
00127                           INT          mindg,
00128                           INT          *oindex,
00129                           INT          *order)
00130 {
00131     const INT row = A->row;
00132     const INT *ia = A->IA;
00133     const INT *ja = A->JA;
00134
00135     INT i, j, spl, k;
00136     SHORT flag = 1;
00137
00138     if (s < row) {
00139         order[s] = jj;
00140         oindex[jj] = s;
00141     }
00142
00143     while (loc <= s && s < row) {
00144         i = order[loc];
00145         spl = s+1;
00146         // neighbor nodes are priority.
00147         for (j=ia[i]+1; j<ia[i+1]; ++j) {
00148             k = ja[j];
00149             if (oindex[k] < 0){
00150                 s++;
00151                 order[s] = k;
00152             }
00153         }
00154         // ordering neighbor nodes by increasing degree
00155         if (s > spl) {
00156             while (flag) {
00157                 flag = 0;
00158                 for (i=spl+1; i<=s; ++i) {
00159                     if (oindex[order[i]] > oindex[order[i-1]]) {
00160                         j = order[i];
00161                         order[i] = order[i-1];
00162                         order[i-1] = j;
00163                         flag = 1;
00164                     }
00165                 }
00166             }
00167         }
00168         for (i=spl; i<=s; ++i) oindex[order[i]] = i;
00169
00170         loc ++;
00171     }
00172 }

```

```

00173
00174     // deal with remainder
00175     if (s < row) {
00176         jj = 0;
00177         i = 0;
00178         while (jj == 0) {
00179             i ++;
00180             if (i >= row) {
00181                 mindg++;
00182                 i = 0;
00183             }
00184             if (oindex[i] < 0 && (ia[i+1]-ia[i] == mindg)) {
00185                 jj = i;
00186             }
00187         }
00188     }
00189     s ++;
00190
00191     CMK_ordering (A, loc, s, jj, mindg, oindex, order);
00192 }
00193 }
00194
00195 /*-----*/
00196 /*--          End of File          --*/
00197 /*-----*/

```

9.63 BlaSchwarzSetup.c File Reference

Setup phase for the Schwarz methods.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [INT fasp_swz_dcsr_setup](#) ([SWZ_data](#) *swzdata, [SWZ_param](#) *swzparam)
Setup phase for the Schwarz methods.
- [void fasp_dcsr_swz_forward](#) ([SWZ_data](#) *swzdata, [SWZ_param](#) *swzparam, [dvector](#) *x, [dvector](#) *b)
Schwarz smoother: forward sweep.
- [void fasp_dcsr_swz_backward](#) ([SWZ_data](#) *swzdata, [SWZ_param](#) *swzparam, [dvector](#) *x, [dvector](#) *b)
Schwarz smoother: backward sweep.

9.63.1 Detailed Description

Setup phase for the Schwarz methods.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#), [AuxVector.c](#), [BlaSparseCSR.c](#), [BlaSparseUtil.c](#), and [KryPvgmres.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSchwarzSetup.c](#).

9.63.2 Function Documentation

9.63.2.1 fasp_dcsr_swz_backward()

```
void fasp_dcsr_swz_backward (
    SWZ_data * swzdata,
    SWZ_param * swzparam,
    dvector * x,
    dvector * b )
```

Schwarz smoother: backward sweep.

Parameters

<i>swzdata</i>	Pointer to the Schwarz data
<i>swzparam</i>	Pointer to the Schwarz parameter
<i>x</i>	Pointer to solution vector
<i>b</i>	Pointer to right hand

Author

Zheng Li, Chensong Zhang

Date

2014/10/5

Definition at line 325 of file [BlaSchwarzSetup.c](#).

9.63.2.2 fasp_dcsr_swz_forward()

```
void fasp_dcsr_swz_forward (
    SWZ_data * swzdata,
    SWZ_param * swzparam,
    dvector * x,
    dvector * b )
```

Schwarz smoother: forward sweep.

Parameters

<i>swzdata</i>	Pointer to the Schwarz data
<i>swzparam</i>	Pointer to the Schwarz parameter
<i>x</i>	Pointer to solution vector
<i>b</i>	Pointer to right hand

Author

Zheng Li, Chensong Zhang

Date

2014/10/5

Definition at line 216 of file [BlaSchwarzSetup.c](#).

9.63.2.3 fasp_swz_dcsr_setup()

```
INT fasp_swz_dcsr_setup (
    SWZ_data * swzdata,
    SWZ_param * swzparam )
```

Setup phase for the Schwarz methods.

Parameters

<i>swzdata</i>	Pointer to the Schwarz data
<i>swzparam</i>	Type of the Schwarz method

Returns

FASP_SUCCESS if succeed

Author

Ludmil, Xiaozhe Hu

Date

03/22/2011

Modified by Zheng Li on 10/09/2014

Definition at line 47 of file [BlaSchwarzSetup.c](#).

9.64 BlaSchwarzSetup.c

[Go to the documentation of this file.](#)

```
00001
00015 #include <math.h>
00016 #include <time.h>
00017
00018 #include "fasp.h"
00019 #include "fasp_funcs.h"
00020
00021 /*-----*/
00022 /*--  Declare Private Functions  --*/
00023 /*-----*/
00024
00025 static void SWZ_level (const INT, dCSRmat *, INT *, INT *, INT *, INT *, const INT);
00026 static void SWZ_block (SWZ_data *, const INT, const INT *, const INT *, INT *);
00027
00028 /*-----*/
00029 /*--      Public Functions      --*/
00030 /*-----*/
00031
00047 INT fasp_swz_dcsr_setup (SWZ_data *swzdata,
00048                          SWZ_param *swzparam)
00049 {
00050     // information about A
00051     dCSRmat A = swzdata->A;
00052     INT n = A.row;
00053
00054     INT blksolver = swzparam->SWZ_blksolver;
00055     INT maxlev = swzparam->SWZ_maxlvl;
00056
00057     // local variables
00058     INT i;
00059     INT inroot = -10, nsizei = -10, nsizeall = -10, nlvl = 0;
00060     INT *jb = NULL;
00061     ivector MIS;
00062
00063     // data for Schwarz method
00064     INT nblk;
```

```

00065     INT *iblock = NULL, *jblock = NULL, *mask = NULL, *maxa = NULL;
00066
00067     // return
00068     INT flag = FASP_SUCCESS;
00069
00070     swzdata->swzparam = swzparam;
00071
00072     #if DEBUG_MODE > 0
00073     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00074     #endif
00075
00076     // allocate memory
00077     maxa = (INT *)fasp_mem_calloc(n, sizeof(INT));
00078     mask = (INT *)fasp_mem_calloc(n, sizeof(INT));
00079     iblock = (INT *)fasp_mem_calloc(n, sizeof(INT));
00080     jblock = (INT *)fasp_mem_calloc(n, sizeof(INT));
00081
00082     nsizeall=0;
00083     memset(mask, 0, sizeof(INT)*n);
00084     memset(iblock, 0, sizeof(INT)*n);
00085     memset(maxa, 0, sizeof(INT)*n);
00086
00087     maxa[0]=0;
00088
00089     // select root nodes
00090     MIS = fasp_sparse_mis(&A);
00091
00092     /*-----*/
00093     // find the blocks
00094     /*-----*/
00095
00096     // first pass: do a maxlev level sets out for each node
00097     for ( i = 0; i < MIS.row; i++ ) {
00098         inroot = MIS.val[i];
00099         SWZ_level(inroot, &A, mask, &nlvl, maxa, jblock, maxlev);
00100         nsizei=maxa[nlvl];
00101         nsizeall+=nsizei;
00102     }
00103
00104     #if DEBUG_MODE > 1
00105     printf("### DEBUG: nsizeall = %d\n", nsizeall);
00106     #endif
00107
00108     // calculated the size of jblock up to here
00109     jblock = (INT *)fasp_mem_realloc(jblock, (nsizeall+n)*sizeof(INT));
00110
00111     // second pass: redo the same again, but this time we store in jblock
00112     maxa[0]=0;
00113     iblock[0]=0;
00114     nsizeall=0;
00115     jb=jblock;
00116     for (i=0; i<MIS.row; i++) {
00117         inroot = MIS.val[i];
00118         SWZ_level(inroot, &A, mask, &nlvl, maxa, jb, maxlev);
00119         nsizei=maxa[nlvl];
00120         iblock[i+1]=iblock[i]+nsizei;
00121         nsizeall+=nsizei;
00122         jb+=nsizei;
00123     }
00124     nblk = MIS.row;
00125
00126     #if DEBUG_MODE > 1
00127     printf("### DEBUG: nsizeall = %d, %d\n", nsizeall, iblock[nblk]);
00128     #endif
00129
00130     /*-----*/
00131     // LU decomposition of blocks
00132     /*-----*/
00133
00134     memset(mask, 0, sizeof(INT)*n);
00135
00136     swzdata->blk_data = (dCSRmat*)fasp_mem_calloc(nblk, sizeof(dCSRmat));
00137
00138     SWZ_block(swzdata, nblk, iblock, jblock, mask);
00139
00140     // Setup for each block solver
00141     switch (blksolver) {
00142
00143     #if WITH_MUMPS
00144         case SOLVER_MUMPS: {
00145             /* use MUMPS direct solver on each block */

```

```

00146         dCSRmat *blk = swzdata->blk_data;
00147         Mumps_data *mumps = (Mumps_data*) fasp_mem_calloc(nblk, sizeof(Mumps_data));
00148         for (i=0; i<nblk; ++i)
00149             mumps[i] = fasp_mumps_factorize(&blk[i], NULL, NULL, PRINT_NONE);
00150         swzdata->mumps = mumps;
00151
00152         break;
00153     }
00154 #endif
00155
00156 #if WITH_UMFPACK
00157     case SOLVER_UMFPACK: {
00158         /* use UMFPACK direct solver on each block */
00159         dCSRmat *blk = swzdata->blk_data;
00160         void **numeric = (void**) fasp_mem_calloc(nblk, sizeof(void*));
00161         dCSRmat Ac_tran;
00162         for (i=0; i<nblk; ++i) {
00163             Ac_tran = fasp_dcsr_create(blk[i].row, blk[i].col, blk[i].nnz);
00164             fasp_dcsr_transz(&blk[i], NULL, &Ac_tran);
00165             fasp_dcsr_cp(&Ac_tran, &blk[i]);
00166             numeric[i] = fasp_umfpack_factorize(&blk[i], 0);
00167         }
00168         swzdata->numeric = numeric;
00169         fasp_dcsr_free(&Ac_tran);
00170
00171         break;
00172     }
00173 #endif
00174
00175     default: {
00176         /* do nothing for iterative methods */
00177     }
00178 }
00179
00180 #if DEBUG_MODE > 1
00181 printf("### DEBUG: n = %d, #blocks = %d, max block size = %d\n",
00182        n, nblk, swzdata->maxbs);
00183 #endif
00184
00185 /*-----*/
00186 // return
00187 /*-----*/
00188 swzdata->nblk = nblk;
00189 swzdata->iblock = iblock;
00190 swzdata->jblock = jblock;
00191 swzdata->mask = mask;
00192 swzdata->maxa = maxa;
00193 swzdata->SWZ_type = swzparam->SWZ_type;
00194
00195 #if DEBUG_MODE > 0
00196 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00197 #endif
00198
00199 return flag;
00200 }
00201
00202 void fasp_dcsr_swz_forward (SWZ_data *swzdata,
00203                            SWZ_param *swzparam,
00204                            dvector *x,
00205                            dvector *b)
00206 {
00207     INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;
00208
00209     // Schwarz partition
00210     INT nblk = swzdata->nblk;
00211     dCSRmat *blk = swzdata->blk_data;
00212     INT *iblock = swzdata->iblock;
00213     INT *jblock = swzdata->jblock;
00214     INT *mask = swzdata->mask;
00215     INT blksolver = swzparam->SWZ_blksolver;
00216
00217     // Schwarz data
00218     dCSRmat A = swzdata->A;
00219     INT *ia = A.ia;
00220     INT *ja = A.ja;
00221     REAL *val = A.val;
00222
00223     // Local solution and right hand vectors
00224     dvector rhs = swzdata->rhsloc1;
00225     dvector u = swzdata->xloc1;
00226 }

```

```

00241 #if WITH_UMFPACK
00242     void **numeric = swzdata->numeric;
00243 #endif
00244
00245 #if WITH_MUMPS
00246     Mumps_data *mumps = swzdata->mumps;
00247 #endif
00248
00249     for (is=0; is<nblk; ++is) {
00250         // Form the right hand of each block
00251         ibl0 = iblock[is];
00252         ibl1 = iblock[is+1];
00253         nloc = ibl1-ibl0;
00254         for (i=0; i<nloc; ++i) {
00255             iblk = ibl0 + i;
00256             ki = jblock[iblk];
00257             mask[ki] = i+1;
00258         }
00259
00260         for (i=0; i<nloc; ++i) {
00261             iblk = ibl0 + i;
00262             ki = jblock[iblk];
00263             rhs.val[i] = b->val[ki];
00264             iaa = ia[ki]-1;
00265             iab = ia[ki+1]-1;
00266             for (kij = iaa; kij<iab; ++kij) {
00267                 kj = ja[kij]-1;
00268                 j = mask[kj];
00269                 if (j == 0) {
00270                     rhs.val[i] -= val[kij]*x->val[kj];
00271                 }
00272             }
00273         }
00274
00275         // Solve each block
00276         switch (blksolver) {
00277
00278 #if WITH_MUMPS
00279             case SOLVER_MUMPS: {
00280                 /* use MUMPS direct solver on each block */
00281                 fasp_mumps_solve(&blk[is], &rhs, &u, mumps[is], 0);
00282                 break;
00283             }
00284 #endif
00285
00286 #if WITH_UMFPACK
00287             case SOLVER_UMFPACK: {
00288                 /* use UMFPACK direct solver on each block */
00289                 fasp_umfpack_solve(&blk[is], &rhs, &u, numeric[is], 0);
00290                 break;
00291             }
00292 #endif
00293
00294             default: {
00295                 /* use iterative solver on each block */
00296                 u.row = blk[is].row;
00297                 rhs.row = blk[is].row;
00298                 fasp_dvec_set(u.row, &u, 0);
00299                 fasp_solver_dcsr_pvgmres(&blk[is], &rhs, &u, NULL, 1e-8, 100, 20, 1, 0);
00300             }
00301
00302             //zero the mask so that everything is as it was
00303             for (i=0; i<nloc; ++i) {
00304                 iblk = ibl0 + i;
00305                 ki = jblock[iblk];
00306                 mask[ki] = 0;
00307                 x->val[ki] = u.val[i];
00308             }
00309         }
00310
00311 void fasp_dcsr_swz_backward (SWZ_data *swzdata,
00312                             SWZ_param *swzparam,
00313                             dvector *x,
00314                             dvector *b)
00315 {
00316     INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;
00317
00318     // Schwarz partition
00319     INT nblk = swzdata->nblk;
00320     dCSRmat *blk = swzdata->blk_data;
00321     INT *iblock = swzdata->iblock;

```

```

00336     INT      *jblock      = swzdata->jblock;
00337     INT      *mask        = swzdata->mask;
00338     INT      blksolver = swzparam->SWZ_blksolver;
00339
00340     // Schwarz data
00341     dCSRmat A = swzdata->A;
00342     INT      *ia = A.IA;
00343     INT      *ja = A.JA;
00344     REAL     *val = A.val;
00345
00346     // Local solution and right hand vectors
00347     dvector rhs = swzdata->rhsloc1;
00348     dvector u   = swzdata->xloc1;
00349
00350     #if WITH_UMFPACK
00351     void **numeric = swzdata->numeric;
00352     #endif
00353
00354     #if WITH_MUMPS
00355     Mumps_data *mumps = swzdata->mumps;
00356     #endif
00357
00358     for (is=nblk-1; is>=0; --is) {
00359         // Form the right hand of each block
00360         ibl0 = iblock[is];
00361         ibl1 = iblock[is+1];
00362         nloc = ibl1-ibl0;
00363         for (i=0; i<nloc; ++i) {
00364             iblk = ibl0 + i;
00365             ki    = jblock[iblk];
00366             mask[ki] = i+1;
00367         }
00368
00369         for (i=0; i<nloc; ++i) {
00370             iblk = ibl0 + i;
00371             ki    = jblock[iblk];
00372             rhs.val[i] = b->val[ki];
00373             iaa = ia[ki]-1;
00374             iab = ia[ki+1]-1;
00375             for (kij = iaa; kij<iab; ++kij) {
00376                 kj = ja[kij]-1;
00377                 j   = mask[kj];
00378                 if (j == 0) {
00379                     rhs.val[i] -= val[kij]*x->val[kj];
00380                 }
00381             }
00382         }
00383
00384         // Solve each block
00385         switch (blksolver) {
00386
00387         #if WITH_MUMPS
00388             case SOLVER_MUMPS: {
00389                 /* use MUMPS direct solver on each block */
00390                 fasp_mumps_solve(&blk[is], &rhs, &u, mumps[is], 0);
00391                 break;
00392             }
00393         #endif
00394
00395         #if WITH_UMFPACK
00396             case SOLVER_UMFPACK: {
00397                 /* use UMFPACK direct solver on each block */
00398                 fasp_umfpack_solve(&blk[is], &rhs, &u, numeric[is], 0);
00399                 break;
00400             }
00401         #endif
00402
00403             default:
00404                 /* use iterative solver on each block */
00405                 rhs.row = blk[is].row;
00406                 u.row   = blk[is].row;
00407                 fasp_dvec_set(u.row, &u, 0);
00408                 fasp_solver_dcsr_pvgmres (&blk[is], &rhs, &u, NULL, 1e-8, 100, 20, 1, 0);
00409             }
00410
00411         //zero the mask so that everything is as it was
00412         for (i=0; i<nloc; ++i) {
00413             iblk = ibl0 + i;
00414             ki    = jblock[iblk];
00415             mask[ki] = 0;
00416             x->val[ki] = u.val[i];
00417         }
00418     }

```

```

00417     }
00418 }
00419
00420 /*-----*/
00421 /*--      Private Functions      --*/
00422 /*-----*/
00423
00441 static void SWZ_level (const INT    inroot,
00442                       dCSRmat      *A,
00443                       INT           *mask,
00444                       INT           *nlvl,
00445                       INT           *iblock,
00446                       INT           *jblock,
00447                       const INT     maxlev)
00448 {
00449     INT *ia = A->IA;
00450     INT *ja = A->JA;
00451     INT nnz = A->nnz;
00452     INT i, j, lvl, lbegin, lvlend, nsize, node;
00453     INT jstrt, jstop, nbr, lvsize;
00454
00455     // This is diagonal
00456     if (ia[inroot+1]-ia[inroot] <= 1) {
00457         lvl = 0;
00458         iblock[lvl] = 0;
00459         jblock[iblock[lvl]] = inroot;
00460         lvl++;
00461         iblock[lvl] = 1;
00462     }
00463     else {
00464         // input node as root node (level 0)
00465         lvl = 0;
00466         jblock[0] = inroot;
00467         lvlend = 0;
00468         nsize = 1;
00469         // mark root node
00470         mask[inroot] = 1;
00471
00472         lvsize = nnz;
00473
00474         // form the level hierarchy for root node(level1, level2, ... maxlev)
00475         while (lvsize > 0 && lvl < maxlev) {
00476             lbegin = lvlend;
00477             lvlend = nsize;
00478             iblock[lvl] = lbegin;
00479             lvl++;
00480             for(i=lbegin; i<lvlend; ++i) {
00481                 node = jblock[i];
00482                 jstrt = ia[node]-1;
00483                 jstop = ia[node+1]-1;
00484                 for (j = jstrt; j<jstop; ++j) {
00485                     nbr = ja[j]-1;
00486                     if (mask[nbr] == 0) {
00487                         jblock[nsize] = nbr;
00488                         mask[nbr] = lvl;
00489                         nsize++;
00490                     }
00491                 }
00492             }
00493             lvsize = nsize - lvlend;
00494         }
00495
00496         iblock[lvl] = nsize;
00497
00498         // reset mask array
00499         for (i = 0; i< nsize; ++i) {
00500             node = jblock[i];
00501             mask[node] = 0;
00502         }
00503     }
00504
00505     *nlvl = lvl;
00506 }
00507
00523 static void SWZ_block (SWZ_data      *swzdata,
00524                       const INT      nblk,
00525                       const INT      *iblock,
00526                       const INT      *jblock,
00527                       INT            *mask)
00528 {
00529     INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;

```

```

00530     INT maxbs = 0, count, nnz;
00531
00532     dCSRmat A = swzdata->A;
00533     dCSRmat *blk = swzdata->blk_data;
00534
00535     INT *ia = A.IA;
00536     INT *ja = A.JA;
00537     REAL *val = A.val;
00538
00539     // get maximal block size
00540     for (is=0; is<nblk; ++is) {
00541         ibl0 = iblock[is];
00542         ibl1 = iblock[is+1];
00543         nloc = ibl1-ibl0;
00544         maxbs = MAX(maxbs, nloc);
00545     }
00546
00547     swzdata->maxbs = maxbs;
00548
00549     // allocate memory for each sub_block's right hand
00550     swzdata->xloc1 = fasp_dvec_create(maxbs);
00551     swzdata->rhsloc1 = fasp_dvec_create(maxbs);
00552
00553     for (is=0; is<nblk; ++is) {
00554         ibl0 = iblock[is];
00555         ibl1 = iblock[is+1];
00556         nloc = ibl1-ibl0;
00557         count = 0;
00558         for (i=0; i<nloc; ++i) {
00559             iblk = ibl0 + i;
00560             ki = jblock[iblk];
00561             iaa = ia[ki]-1;
00562             iab = ia[ki+1]-1;
00563             count += iab - iaa;
00564             mask[ki] = i+1;
00565         }
00566
00567         blk[is] = fasp_dcsr_create(nloc, nloc, count);
00568         blk[is].IA[0] = 0;
00569         nnz = 0;
00570
00571         for (i=0; i<nloc; ++i) {
00572             iblk = ibl0 + i;
00573             ki = jblock[iblk];
00574             iaa = ia[ki]-1;
00575             iab = ia[ki+1]-1;
00576             for (kij = iaa; kij<iab; ++kij) {
00577                 kj = ja[kij]-1;
00578                 j = mask[kj];
00579                 if (j != 0) {
00580                     blk[is].JA[nnz] = j-1;
00581                     blk[is].val[nnz] = val[kij];
00582                     nnz ++;
00583                 }
00584             }
00585             blk[is].IA[i+1] = nnz;
00586         }
00587
00588         blk[is].nnz = nnz;
00589
00590         // zero the mask so that everything is as it was
00591         for (i=0; i<nloc; ++i) {
00592             iblk = ibl0 + i;
00593             ki = jblock[iblk];
00594             mask[ki] = 0;
00595         }
00596     }
00597 }
00598
00599 /*-----*/
00600 /*--      End of File      --*/
00601 /*-----*/

```

9.65 BlaSmallMat.c File Reference

BLAS operations for *small* dense matrices.

```
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- void `fasp_blas_smat_axm` (REAL *a, const INT n, const REAL alpha)
*Compute $a = \alpha * a$ (in place)*
- void `fasp_blas_smat_add` (const REAL *a, const REAL *b, const INT n, const REAL alpha, const REAL beta, REAL *c)
*Compute $c = \alpha * a + \beta * b$.*
- void `fasp_blas_smat_m xv_nc2` (const REAL *a, const REAL *b, REAL *c)
*Compute the product of a 2*2 matrix a and a array b, stored in c.*
- void `fasp_blas_smat_m xv_nc3` (const REAL *a, const REAL *b, REAL *c)
*Compute the product of a 3*3 matrix a and a array b, stored in c.*
- void `fasp_blas_smat_m xv_nc4` (const REAL *a, const REAL *b, REAL *c)
*Compute the product of a 4*4 matrix a and a array b, stored in c.*
- void `fasp_blas_smat_m xv_nc5` (const REAL *a, const REAL *b, REAL *c)
*Compute the product of a 5*5 matrix a and a array b, stored in c.*
- void `fasp_blas_smat_m xv_nc7` (const REAL *a, const REAL *b, REAL *c)
*Compute the product of a 7*7 matrix a and a array b, stored in c.*
- void `fasp_blas_smat_m xv` (const REAL *a, const REAL *b, REAL *c, const INT n)
Compute the product of a small full matrix a and a array b, stored in c.
- void `fasp_blas_smat_mul_nc2` (const REAL *a, const REAL *b, REAL *c)
Compute the matrix product of two 2 matrices a and b, stored in c.*
- void `fasp_blas_smat_mul_nc3` (const REAL *a, const REAL *b, REAL *c)
*Compute the matrix product of two 3*3 matrices a and b, stored in c.*
- void `fasp_blas_smat_mul_nc4` (const REAL *a, const REAL *b, REAL *c)
*Compute the matrix product of two 4*4 matrices a and b, stored in c.*
- void `fasp_blas_smat_mul_nc5` (const REAL *a, const REAL *b, REAL *c)
*Compute the matrix product of two 5*5 matrices a and b, stored in c.*
- void `fasp_blas_smat_mul_nc7` (const REAL *a, const REAL *b, REAL *c)
*Compute the matrix product of two 7*7 matrices a and b, stored in c.*
- void `fasp_blas_smat_mul` (const REAL *a, const REAL *b, REAL *c, const INT n)
Compute the matrix product of two small full matrices a and b, stored in c.
- void `fasp_blas_smat_ypAx_nc2` (const REAL *A, const REAL *x, REAL *y)
*Compute $y := y + Ax$, where 'A' is a 2*2 dense matrix.*
- void `fasp_blas_smat_ypAx_nc3` (const REAL *A, const REAL *x, REAL *y)
*Compute $y := y + Ax$, where 'A' is a 3*3 dense matrix.*
- void `fasp_blas_smat_ypAx_nc4` (const REAL *A, const REAL *x, REAL *y)
*Compute $y := y + Ax$, where 'A' is a 4*4 dense matrix.*
- void `fasp_blas_smat_ypAx_nc5` (const REAL *A, const REAL *x, REAL *y)
*Compute $y := y + Ax$, where 'A' is a 5*5 dense matrix.*
- void `fasp_blas_smat_ypAx_nc7` (const REAL *A, const REAL *x, REAL *y)
*Compute $y := y + Ax$, where 'A' is a 7*7 dense matrix.*
- void `fasp_blas_smat_ypAx` (const REAL *A, const REAL *x, REAL *y, const INT n)
*Compute $y := y + Ax$, where 'A' is a n*n dense matrix.*
- void `fasp_blas_smat_ymAx_nc2` (const REAL *A, const REAL *x, REAL *y)

- Compute $y := y - Ax$, where 'A' is a 2*2 dense matrix.*

 - void [fasp_blas_smat_ymAx_nc3](#) (const [REAL](#) *A, const [REAL](#) *x, [REAL](#) *y)
- Compute $y := y - Ax$, where 'A' is a 3*3 dense matrix.*

 - void [fasp_blas_smat_ymAx_nc4](#) (const [REAL](#) *A, const [REAL](#) *x, [REAL](#) *y)
- Compute $y := y - Ax$, where 'A' is a 4*4 dense matrix.*

 - void [fasp_blas_smat_ymAx_nc5](#) (const [REAL](#) *A, const [REAL](#) *x, [REAL](#) *y)
- Compute $y := y - Ax$, where 'A' is a 5*5 dense matrix.*

 - void [fasp_blas_smat_ymAx_nc7](#) (const [REAL](#) *A, const [REAL](#) *x, [REAL](#) *y)
- Compute $y := y - Ax$, where 'A' is a 7*7 dense matrix.*

 - void [fasp_blas_smat_ymAx](#) (const [REAL](#) *A, const [REAL](#) *x, [REAL](#) *y, const [INT](#) n)
- Compute $y := y - Ax$, where 'A' is a n*n dense matrix.*

 - void [fasp_blas_smat_aAxpby](#) (const [REAL](#) alpha, const [REAL](#) *A, const [REAL](#) *x, const [REAL](#) beta, [REAL](#) *y, const [INT](#) n)

*Compute $y := \alpha * A * x + \beta * y$*

9.65.1 Detailed Description

BLAS operations for *small* dense matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), and [PreDataInit.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Warning

These routines are designed for full matrices only!

This file contains very long lines. Not print friendly!

Definition in file [BlaSmallMat.c](#).

9.65.2 Function Documentation

9.65.2.1 [fasp_blas_smat_aAxpby\(\)](#)

```
void fasp_blas_smat_aAxpby (
    const REAL alpha,
    const REAL * A,
    const REAL * x,
    const REAL beta,
    REAL * y,
    const INT n )
```

Compute $y := \alpha * A * x + \beta * y$

Parameters

<i>alpha</i>	REAL factor alpha
--------------	-------------------

Parameters

<i>A</i>	Pointer to the REAL array which stands for a n*n full matrix
<i>x</i>	Pointer to the REAL array with length n
<i>beta</i>	REAL factor beta
<i>y</i>	Pointer to the REAL array with length n
<i>n</i>	Length of array x and y

Author

Zhiyang Zhou, Chensong Zhang

Date

2010/10/25

Definition at line 1064 of file [BlaSmallMat.c](#).

9.65.2.2 fasp_blas_smat_add()

```
void fasp_blas_smat_add (
    const REAL * a,
    const REAL * b,
    const INT n,
    const REAL alpha,
    const REAL beta,
    REAL * c )
```

Compute $c = \alpha * a + \beta * b$.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>b</i>	Pointer to the REAL array which stands a n*n matrix
<i>n</i>	Dimension of the matrix
<i>alpha</i>	Scalar
<i>beta</i>	Scalar
<i>c</i>	Pointer to the REAL array which stands a n*n matrix

Author

Xiaozhe Hu, Chensong Zhang

Date

05/26/2014

Definition at line 65 of file [BlaSmallMat.c](#).

9.65.2.3 fasp_blas_smat_axm()

```
void fasp_blas_smat_axm (
    REAL * a,
```

```
const INT n,
const REAL alpha )
```

Compute $a = \alpha * a$ (in place)

Parameters

<i>a</i>	Pointer to the REAL array which stands a $n \times n$ matrix
<i>n</i>	Dimension of the matrix
<i>alpha</i>	Scalar

Author

Xiaozhe Hu, Chensong Zhang

Date

05/26/2014

Definition at line 37 of file [BlaSmallMat.c](#).

9.65.2.4 fasp_blas_smat_mul()

```
void fasp_blas_smat_mul (
    const REAL * a,
    const REAL * b,
    REAL * c,
    const INT n )
```

Compute the matrix product of two small full matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a $n \times n$ matrix
<i>b</i>	Pointer to the REAL array which stands a $n \times n$ matrix
<i>c</i>	Pointer to the REAL array which stands a $n \times n$ matrix
<i>n</i>	Dimension of the matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

04/21/2010

Author

Li Zhao, the case of adding $n = 4$

Date

04/18/2021

Definition at line 540 of file [BlaSmallMat.c](#).

9.65.2.5 fasp_blas_smat_mul_nc2()

```
void fasp_blas_smat_mul_nc2 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the matrix product of two 2* matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>b</i>	Pointer to the REAL array which stands a n*n matrix
<i>c</i>	Pointer to the REAL array which stands a n*n matrix

Author

Xiaozhe Hu

Date

18/11/2011

Definition at line 275 of file [BlaSmallMat.c](#).

9.65.2.6 fasp_blas_smat_mul_nc3()

```
void fasp_blas_smat_mul_nc3 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the matrix product of two 3*3 matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>b</i>	Pointer to the REAL array which stands a n*n matrix
<i>c</i>	Pointer to the REAL array which stands a n*n matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 304 of file [BlaSmallMat.c](#).

9.65.2.7 fasp_blas_smat_mul_nc4()

```
void fasp_blas_smat_mul_nc4 (  
    const REAL * a,
```

```
const REAL * b,  
REAL * c )
```

Compute the matrix product of two 4*4 matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>b</i>	Pointer to the REAL array which stands a n*n matrix
<i>c</i>	Pointer to the REAL array which stands a n*n matrix

Author

Li Zhao

Date

04/18/2021

Definition at line 341 of file [BlasSmallMat.c](#).

9.65.2.8 fasp_blas_smat_mul_nc5()

```
void fasp_blas_smat_mul_nc5 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the matrix product of two 5*5 matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 5*5 matrix
<i>b</i>	Pointer to the REAL array which stands a 5*5 matrix
<i>c</i>	Pointer to the REAL array which stands a 5*5 matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 388 of file [BlasSmallMat.c](#).

9.65.2.9 fasp_blas_smat_mul_nc7()

```
void fasp_blas_smat_mul_nc7 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the matrix product of two 7*7 matrices a and b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 7*7 matrix
<i>b</i>	Pointer to the REAL array which stands a 7*7 matrix
<i>c</i>	Pointer to the REAL array which stands a 7*7 matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line [447](#) of file [BlaSmallMat.c](#).

9.65.2.10 fasp_blas_smat_m xv()

```
void fasp_blas_smat_m xv (
    const REAL * a,
    const REAL * b,
    REAL * c,
    const INT n )
```

Compute the product of a small full matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>b</i>	Pointer to the REAL array with length n
<i>c</i>	Pointer to the REAL array with length n
<i>n</i>	Dimension of the matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

04/21/2010

Author

Li Zhao, the case of adding n = 4

Date

04/18/2021

Definition at line [221](#) of file [BlaSmallMat.c](#).

9.65.2.11 fasp_blas_smat_m xv_nc2()

```
void fasp_blas_smat_m xv_nc2 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the product of a 2*2 matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 2*2 matrix
<i>b</i>	Pointer to the REAL array with length 2
<i>c</i>	Pointer to the REAL array with length 2

Author

Xiaozhe Hu

Date

18/11/2010

Definition at line 93 of file [BlaSmallMat.c](#).

9.65.2.12 fasp_blas_smat_m xv_nc3()

```
void fasp_blas_smat_m xv_nc3 (  
    const REAL * a,  
    const REAL * b,  
    REAL * c )
```

Compute the product of a 3*3 matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 3*3 matrix
<i>b</i>	Pointer to the REAL array with length 3
<i>c</i>	Pointer to the REAL array with length 3

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 115 of file [BlaSmallMat.c](#).

9.65.2.13 fasp_blas_smat_m xv_nc4()

```
void fasp_blas_smat_m xv_nc4 (  
    const REAL * a,
```

```
const REAL * b,
REAL * c )
```

Compute the product of a 4*4 matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 4*4 matrix
<i>b</i>	Pointer to the REAL array with length 4
<i>c</i>	Pointer to the REAL array with length 4

Author

Li Zhao

Date

04/18/2021

Definition at line 138 of file [BlaSmallMat.c](#).

9.65.2.14 fasp_blas_smat_m xv_nc5()

```
void fasp_blas_smat_m xv_nc5 (
    const REAL * a,
    const REAL * b,
    REAL * c )
```

Compute the product of a 5*5 matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 5*5 matrix
<i>b</i>	Pointer to the REAL array with length 5
<i>c</i>	Pointer to the REAL array with length 5

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 162 of file [BlaSmallMat.c](#).

9.65.2.15 fasp_blas_smat_m xv_nc7()

```
void fasp_blas_smat_m xv_nc7 (
    const REAL * a,
    const REAL * b,
    REAL * c )
```

Compute the product of a 7*7 matrix a and a array b, stored in c.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 7*7 matrix
<i>b</i>	Pointer to the REAL array with length 7
<i>c</i>	Pointer to the REAL array with length 7

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 188 of file [BlaSmallMat.c](#).

9.65.2.16 fasp_blas_smat_ymAx()

```
void fasp_blas_smat_ymAx (
    const REAL * A,
    const REAL * x,
    REAL * y,
    const INT n )
```

Compute $y := y - Ax$, where 'A' is a $n \times n$ dense matrix.

Parameters

<i>A</i>	Pointer to the $n \times n$ dense matrix
<i>x</i>	Pointer to the REAL array with length n
<i>y</i>	Pointer to the REAL array with length n
<i>n</i>	the dimension of the dense matrix

Author

Zhiyang Zhou, Xiaozhe Hu, Chensong Zhang

Date

2010/10/25

Modified by Chensong Zhang on 01/25/2017

Definition at line 962 of file [BlaSmallMat.c](#).

9.65.2.17 fasp_blas_smat_ymAx_nc2()

```
void fasp_blas_smat_ymAx_nc2 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y - Ax$, where 'A' is a 2×2 dense matrix.

Parameters

<i>A</i>	Pointer to the 2*2 dense matrix
<i>x</i>	Pointer to the REAL array with length 3
<i>y</i>	Pointer to the REAL array with length 3

Author

Xiaozhe Hu

Date

18/11/2011

Note

Works for 2-component

Definition at line 820 of file [BlaSmallMat.c](#).**9.65.2.18 fasp_blas_smat_ymAx_nc3()**

```
void fasp_blas_smat_ymAx_nc3 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y - Ax$, where 'A' is a 3*3 dense matrix.**Parameters**

<i>A</i>	Pointer to the 3*3 dense matrix
<i>x</i>	Pointer to the REAL array with length 3
<i>y</i>	Pointer to the REAL array with length 3

Author

Xiaozhe Hu, Zhiyang Zhou

Date

01/06/2011

Note

Works for 3-component

Definition at line 846 of file [BlaSmallMat.c](#).**9.65.2.19 fasp_blas_smat_ymAx_nc4()**

```
void fasp_blas_smat_ymAx_nc4 (
    const REAL * A,
```

```
const REAL * x,  
REAL * y )
```

Compute $y := y - Ax$, where 'A' is a 4*4 dense matrix.

Parameters

<i>A</i>	Pointer to the 4*4 dense matrix
<i>x</i>	Pointer to the REAL array with length 4
<i>y</i>	Pointer to the REAL array with length 4

Author

Li Zhao

Date

04/18/2021

Note

Works for 4-component

Definition at line 873 of file [BlaSmallMat.c](#).**9.65.2.20 fasp_blas_smat_ymAx_nc5()**

```
void fasp_blas_smat_ymAx_nc5 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y - Ax$, where 'A' is a 5*5 dense matrix.**Parameters**

<i>A</i>	Pointer to the 5*5 dense matrix
<i>x</i>	Pointer to the REAL array with length 5
<i>y</i>	Pointer to the REAL array with length 5

Author

Xiaozhe Hu, Zhiyang Zhou

Date

01/06/2011

Note

Works for 5-component

Definition at line 900 of file [BlaSmallMat.c](#).**9.65.2.21 fasp_blas_smat_ymAx_nc7()**

```
void fasp_blas_smat_ymAx_nc7 (
    const REAL * A,
```

```
const REAL * x,  
REAL * y )
```

Compute $y := y - Ax$, where 'A' is a 7*7 dense matrix.

Parameters

<i>A</i>	Pointer to the 7*7 dense matrix
<i>x</i>	Pointer to the REAL array with length 7
<i>y</i>	Pointer to the REAL array with length 7

Author

Xiaozhe Hu, Zhiyang Zhou

Date

01/06/2011

Note

Works for 7-component

Definition at line 929 of file [BlaSmallMat.c](#).

9.65.2.22 fasp_blas_smat_ypAx()

```
void fasp_blas_smat_ypAx (
    const REAL * A,
    const REAL * x,
    REAL * y,
    const INT n )
```

Compute $y := y + Ax$, where 'A' is a $n \times n$ dense matrix.

Parameters

<i>A</i>	Pointer to the $n \times n$ dense matrix
<i>x</i>	Pointer to the REAL array with length n
<i>y</i>	Pointer to the REAL array with length n
<i>n</i>	Dimension of the dense matrix

Author

Zhiyang Zhou, Chensong Zhang

Date

2010/10/25

Modified by Chensong Zhang on 01/25/2017

Definition at line 720 of file [BlaSmallMat.c](#).

9.65.2.23 fasp_blas_smat_ypAx_nc2()

```
void fasp_blas_smat_ypAx_nc2 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y + Ax$, where 'A' is a 2*2 dense matrix.

Parameters

<i>A</i>	Pointer to the 3*3 dense matrix
<i>x</i>	Pointer to the REAL array with length 3
<i>y</i>	Pointer to the REAL array with length 3

Author

Xiaozhe Hu

Date

2011/11/18

Definition at line 589 of file [BlaSmallMat.c](#).**9.65.2.24 fasp_blas_smat_ypAx_nc3()**

```
void fasp_blas_smat_ypAx_nc3 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y + Ax$, where 'A' is a 3*3 dense matrix.

Parameters

<i>A</i>	Pointer to the 3*3 dense matrix
<i>x</i>	Pointer to the REAL array with length 3
<i>y</i>	Pointer to the REAL array with length 3

Author

Zhiyang Zhou, Xiaozhe Hu

Date

2010/10/25

Definition at line 613 of file [BlaSmallMat.c](#).**9.65.2.25 fasp_blas_smat_ypAx_nc4()**

```
void fasp_blas_smat_ypAx_nc4 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y + Ax$, where 'A' is a 4*4 dense matrix.

Parameters

<i>A</i>	Pointer to the 4*4 dense matrix
<i>x</i>	Pointer to the REAL array with length 4
<i>y</i>	Pointer to the REAL array with length 4

Author

Li Zhao

Date

2021/04/18

Definition at line 637 of file [BlaSmallMat.c](#).**9.65.2.26 fasp_blas_smat_ypAx_nc5()**

```
void fasp_blas_smat_ypAx_nc5 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y + Ax$, where 'A' is a 5*5 dense matrix.**Parameters**

A	Pointer to the 5*5 dense matrix
x	Pointer to the REAL array with length 5
y	Pointer to the REAL array with length 5

Author

Zhiyang Zhou, Xiaozhe Hu, Chensong Zhang

Date

2010/10/25

Definition at line 662 of file [BlaSmallMat.c](#).**9.65.2.27 fasp_blas_smat_ypAx_nc7()**

```
void fasp_blas_smat_ypAx_nc7 (
    const REAL * A,
    const REAL * x,
    REAL * y )
```

Compute $y := y + Ax$, where 'A' is a 7*7 dense matrix.**Parameters**

A	Pointer to the 7*7 dense matrix
x	Pointer to the REAL array with length 7
y	Pointer to the REAL array with length 7

Author

Zhiyang Zhou, Xiaozhe Hu, Chensong Zhang

Date

2010/10/25

Definition at line 688 of file [BlaSmallMat.c](#).

9.66 BlaSmallMat.c

[Go to the documentation of this file.](#)

```

00001
00018 #include "fasp.h"
00019 #include "fasp_functs.h"
00020
00021 /*-----*/
00022 /*--      Public Functions      --*/
00023 /*-----*/
00024
00037 void fasp_blas_smat_axm (REAL      *a,
00038                          const INT  n,
00039                          const REAL alpha)
00040 {
00041     const INT  n2 = n*n;
00042     INT        i;
00043
00044     for ( i = 0; i < n2; i++ ) a[i] *= alpha;
00045
00046     return;
00047 }
00048
00065 void fasp_blas_smat_add (const REAL *a,
00066                         const REAL *b,
00067                         const INT  n,
00068                         const REAL alpha,
00069                         const REAL beta,
00070                         REAL      *c)
00071 {
00072     const INT  n2 = n*n;
00073     INT        i;
00074
00075     for ( i = 0; i < n2; i++ ) c[i] = alpha * a[i] + beta * b[i];
00076
00077     return;
00078 }
00079
00080
00093 void fasp_blas_smat_mxx_nc2 (const REAL *a,
00094                             const REAL *b,
00095                             REAL      *c)
00096 {
00097     const REAL b0 = b[0], b1 = b[1];
00098
00099     c[0] = a[0]*b0 + a[1]*b1;
00100     c[1] = a[2]*b0 + a[3]*b1;
00101 }
00102
00115 void fasp_blas_smat_mxx_nc3 (const REAL *a,
00116                             const REAL *b,
00117                             REAL      *c)
00118 {
00119     const REAL b0 = b[0], b1 = b[1], b2 = b[2];
00120
00121     c[0] = a[0]*b0 + a[1]*b1 + a[2]*b2;
00122     c[1] = a[3]*b0 + a[4]*b1 + a[5]*b2;
00123     c[2] = a[6]*b0 + a[7]*b1 + a[8]*b2;
00124 }
00125
00138 void fasp_blas_smat_mxx_nc4 (const REAL *a,
00139                             const REAL *b,
00140                             REAL      *c)
00141 {
00142     const REAL b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3];
00143
00144     c[0] = a[0] *b0 + a[1] *b1 + a[2] *b2 + a[3] *b3;
00145     c[1] = a[4] *b0 + a[5] *b1 + a[6] *b2 + a[7] *b3;
00146     c[2] = a[8] *b0 + a[9] *b1 + a[10]*b2 + a[11]*b3;
00147     c[3] = a[12]*b0 + a[13]*b1 + a[14]*b2 + a[15]*b3;
00148 }
00149

```

```

00162 void fasp_blas_smat_m xv_nc5 (const REAL *a,
00163                                const REAL *b,
00164                                REAL *c)
00165 {
00166     const REAL b0 = b[0], b1 = b[1], b2 = b[2];
00167     const REAL b3 = b[3], b4 = b[4];
00168
00169     c[0] = a[0]*b0 + a[1]*b1 + a[2]*b2 + a[3]*b3 + a[4]*b4;
00170     c[1] = a[5]*b0 + a[6]*b1 + a[7]*b2 + a[8]*b3 + a[9]*b4;
00171     c[2] = a[10]*b0 + a[11]*b1 + a[12]*b2 + a[13]*b3 + a[14]*b4;
00172     c[3] = a[15]*b0 + a[16]*b1 + a[17]*b2 + a[18]*b3 + a[19]*b4;
00173     c[4] = a[20]*b0 + a[21]*b1 + a[22]*b2 + a[23]*b3 + a[24]*b4;
00174 }
00175
00188 void fasp_blas_smat_m xv_nc7 (const REAL *a,
00189                                const REAL *b,
00190                                REAL *c)
00191 {
00192     const REAL b0 = b[0], b1 = b[1], b2 = b[2];
00193     const REAL b3 = b[3], b4 = b[4], b5 = b[5], b6 = b[6];
00194
00195     c[0] = a[0]*b0 + a[1]*b1 + a[2]*b2 + a[3]*b3 + a[4]*b4 + a[5]*b5 + a[6]*b6;
00196     c[1] = a[7]*b0 + a[8]*b1 + a[9]*b2 + a[10]*b3 + a[11]*b4 + a[12]*b5 + a[13]*b6;
00197     c[2] = a[14]*b0 + a[15]*b1 + a[16]*b2 + a[17]*b3 + a[18]*b4 + a[19]*b5 + a[20]*b6;
00198     c[3] = a[21]*b0 + a[22]*b1 + a[23]*b2 + a[24]*b3 + a[25]*b4 + a[26]*b5 + a[27]*b6;
00199     c[4] = a[28]*b0 + a[29]*b1 + a[30]*b2 + a[31]*b3 + a[32]*b4 + a[33]*b5 + a[34]*b6;
00200     c[5] = a[35]*b0 + a[36]*b1 + a[37]*b2 + a[38]*b3 + a[39]*b4 + a[40]*b5 + a[41]*b6;
00201     c[6] = a[42]*b0 + a[43]*b1 + a[44]*b2 + a[45]*b3 + a[46]*b4 + a[47]*b5 + a[48]*b6;
00202 }
00203
00221 void fasp_blas_smat_m xv (const REAL *a,
00222                            const REAL *b,
00223                            REAL *c,
00224                            const INT n)
00225 {
00226     switch (n) {
00227     case 2:
00228         fasp_blas_smat_m xv_nc2(a, b, c);
00229         break;
00230
00231     case 3:
00232         fasp_blas_smat_m xv_nc3(a, b, c);
00233         break;
00234
00235     case 4:
00236         fasp_blas_smat_m xv_nc4(a, b, c);
00237         break;
00238
00239     case 5:
00240         fasp_blas_smat_m xv_nc5(a, b, c);
00241         break;
00242
00243     case 7:
00244         fasp_blas_smat_m xv_nc7(a, b, c);
00245         break;
00246
00247     default:
00248         {
00249             INT i,j,in=0;
00250             REAL temp;
00251
00252             for (i=0; i<n; ++i, in+=n) {
00253                 temp = 0.0;
00254                 for (j=0; j<n; ++j) temp += a[in+j]*b[j];
00255                 c[i]=temp;
00256             } // end for i
00257         }
00258         break;
00259     }
00260     return;
00261 }
00262
00275 void fasp_blas_smat_m ul_nc2 (const REAL *a,
00276                                const REAL *b,
00277                                REAL *c)
00278 {
00279     const REAL a0 = a[0], a1 = a[1];
00280     const REAL a2 = a[2], a3 = a[3];
00281
00282     const REAL b0 = b[0], b1 = b[1];
00283     const REAL b2 = b[2], b3 = b[3];

```

```

00284
00285     c[0] = a0*b0 + a1*b2;
00286     c[1] = a0*b1 + a1*b3;
00287     c[2] = a2*b0 + a3*b2;
00288     c[3] = a2*b1 + a3*b3;
00289
00290 }
00291
00304 void fasp_blas_smat_mul_nc3 (const REAL *a,
00305                               const REAL *b,
00306                               REAL *c)
00307 {
00308     const REAL a0 = a[0], a1 = a[1], a2 = a[2];
00309     const REAL a3 = a[3], a4 = a[4], a5 = a[5];
00310     const REAL a6 = a[6], a7 = a[7], a8 = a[8];
00311
00312     const REAL b0 = b[0], b1 = b[1], b2 = b[2];
00313     const REAL b3 = b[3], b4 = b[4], b5 = b[5];
00314     const REAL b6 = b[6], b7 = b[7], b8 = b[8];
00315
00316     c[0] = a0*b0 + a1*b3 + a2*b6;
00317     c[1] = a0*b1 + a1*b4 + a2*b7;
00318     c[2] = a0*b2 + a1*b5 + a2*b8;
00319
00320     c[3] = a3*b0 + a4*b3 + a5*b6;
00321     c[4] = a3*b1 + a4*b4 + a5*b7;
00322     c[5] = a3*b2 + a4*b5 + a5*b8;
00323
00324     c[6] = a6*b0 + a7*b3 + a8*b6;
00325     c[7] = a6*b1 + a7*b4 + a8*b7;
00326     c[8] = a6*b2 + a7*b5 + a8*b8;
00327 }
00328
00341 void fasp_blas_smat_mul_nc4 (const REAL *a,
00342                               const REAL *b,
00343                               REAL *c)
00344 {
00345     const REAL a0 = a[0], a1 = a[1], a2 = a[2], a3 = a[3];
00346     const REAL a4 = a[4], a5 = a[5], a6 = a[6], a7 = a[7];
00347     const REAL a8 = a[8], a9 = a[9], a10 = a[10], a11 = a[11];
00348     const REAL a12 = a[12], a13 = a[13], a14 = a[14], a15 = a[15];
00349
00350     const REAL b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3];
00351     const REAL b4 = b[4], b5 = b[5], b6 = b[6], b7 = b[7];
00352     const REAL b8 = b[8], b9 = b[9], b10 = b[10], b11 = b[11];
00353     const REAL b12 = b[12], b13 = b[13], b14 = b[14], b15 = b[15];
00354
00355     c[0] = a0*b0 + a1*b4 + a2*b8 + a3*b12;
00356     c[1] = a0*b1 + a1*b5 + a2*b9 + a3*b13;
00357     c[2] = a0*b2 + a1*b6 + a2*b10 + a3*b14;
00358     c[3] = a0*b3 + a1*b7 + a2*b11 + a3*b15;
00359
00360     c[4] = a4*b0 + a5*b4 + a6*b8 + a7*b12;
00361     c[5] = a4*b1 + a5*b5 + a6*b9 + a7*b13;
00362     c[6] = a4*b2 + a5*b6 + a6*b10 + a7*b14;
00363     c[7] = a4*b3 + a5*b7 + a6*b11 + a7*b15;
00364
00365     c[8] = a8*b0 + a9*b4 + a10*b8 + a11*b12;
00366     c[9] = a8*b1 + a9*b5 + a10*b9 + a11*b13;
00367     c[10] = a8*b2 + a9*b6 + a10*b10 + a11*b14;
00368     c[11] = a8*b3 + a9*b7 + a10*b11 + a11*b15;
00369
00370     c[12] = a12*b0 + a13*b4 + a14*b8 + a15*b12;
00371     c[13] = a12*b1 + a13*b5 + a14*b9 + a15*b13;
00372     c[14] = a12*b2 + a13*b6 + a14*b10 + a15*b14;
00373     c[15] = a12*b3 + a13*b7 + a14*b11 + a15*b15;
00374 }
00375
00388 void fasp_blas_smat_mul_nc5 (const REAL *a,
00389                               const REAL *b,
00390                               REAL *c)
00391 {
00392     const REAL a0 = a[0], a1 = a[1], a2 = a[2], a3 = a[3], a4 = a[4];
00393     const REAL a5 = a[5], a6 = a[6], a7 = a[7], a8 = a[8], a9 = a[9];
00394     const REAL a10 = a[10], a11 = a[11], a12 = a[12], a13 = a[13], a14 = a[14];
00395     const REAL a15 = a[15], a16 = a[16], a17 = a[17], a18 = a[18], a19 = a[19];
00396     const REAL a20 = a[20], a21 = a[21], a22 = a[22], a23 = a[23], a24 = a[24];
00397
00398     const REAL b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3], b4 = b[4];
00399     const REAL b5 = b[5], b6 = b[6], b7 = b[7], b8 = b[8], b9 = b[9];
00400     const REAL b10 = b[10], b11 = b[11], b12 = b[12], b13 = b[13], b14 = b[14];

```

```

00401     const REAL b15 = b[15], b16 = b[16], b17 = b[17], b18 = b[18], b19 = b[19];
00402     const REAL b20 = b[20], b21 = b[21], b22 = b[22], b23 = b[23], b24 = b[24];
00403
00404     c[0] = a0*b0 + a1*b5 + a2*b10 + a3*b15 + a4*b20;
00405     c[1] = a0*b1 + a1*b6 + a2*b11 + a3*b16 + a4*b21;
00406     c[2] = a0*b2 + a1*b7 + a2*b12 + a3*b17 + a4*b22;
00407     c[3] = a0*b3 + a1*b8 + a2*b13 + a3*b18 + a4*b23;
00408     c[4] = a0*b4 + a1*b9 + a2*b14 + a3*b19 + a4*b24;
00409
00410     c[5] = a5*b0 + a6*b5 + a7*b10 + a8*b15 + a9*b20;
00411     c[6] = a5*b1 + a6*b6 + a7*b11 + a8*b16 + a9*b21;
00412     c[7] = a5*b2 + a6*b7 + a7*b12 + a8*b17 + a9*b22;
00413     c[8] = a5*b3 + a6*b8 + a7*b13 + a8*b18 + a9*b23;
00414     c[9] = a5*b4 + a6*b9 + a7*b14 + a8*b19 + a9*b24;
00415
00416     c[10] = a10*b0 + a11*b5 + a12*b10 + a13*b15 + a14*b20;
00417     c[11] = a10*b1 + a11*b6 + a12*b11 + a13*b16 + a14*b21;
00418     c[12] = a10*b2 + a11*b7 + a12*b12 + a13*b17 + a14*b22;
00419     c[13] = a10*b3 + a11*b8 + a12*b13 + a13*b18 + a14*b23;
00420     c[14] = a10*b4 + a11*b9 + a12*b14 + a13*b19 + a14*b24;
00421
00422     c[15] = a15*b0 + a16*b5 + a17*b10 + a18*b15 + a19*b20;
00423     c[16] = a15*b1 + a16*b6 + a17*b11 + a18*b16 + a19*b21;
00424     c[17] = a15*b2 + a16*b7 + a17*b12 + a18*b17 + a19*b22;
00425     c[18] = a15*b3 + a16*b8 + a17*b13 + a18*b18 + a19*b23;
00426     c[19] = a15*b4 + a16*b9 + a17*b14 + a18*b19 + a19*b24;
00427
00428     c[20] = a20*b0 + a21*b5 + a22*b10 + a23*b15 + a24*b20;
00429     c[21] = a20*b1 + a21*b6 + a22*b11 + a23*b16 + a24*b21;
00430     c[22] = a20*b2 + a21*b7 + a22*b12 + a23*b17 + a24*b22;
00431     c[23] = a20*b3 + a21*b8 + a22*b13 + a23*b18 + a24*b23;
00432     c[24] = a20*b4 + a21*b9 + a22*b14 + a23*b19 + a24*b24;
00433 }
00434
00447 void fasp_blas_smat_mul_nc7 (const REAL *a,
00448                               const REAL *b,
00449                               REAL *c)
00450 {
00451     const REAL a0 = a[0], a1 = a[1], a2 = a[2], a3 = a[3], a4 = a[4], a5 = a[5], a6 = a[6];
00452     const REAL a7 = a[7], a8 = a[8], a9 = a[9], a10 = a[10], a11 = a[11], a12 = a[12], a13 = a[13];
00453     const REAL a14 = a[14], a15 = a[15], a16 = a[16], a17 = a[17], a18 = a[18], a19 = a[19], a20 = a[20];
00454     const REAL a21 = a[21], a22 = a[22], a23 = a[23], a24 = a[24], a25 = a[25], a26 = a[26], a27 = a[27];
00455     const REAL a28 = a[28], a29 = a[29], a30 = a[30], a31 = a[31], a32 = a[32], a33 = a[33], a34 = a[34];
00456     const REAL a35 = a[35], a36 = a[36], a37 = a[37], a38 = a[38], a39 = a[39], a40 = a[40], a41 = a[41];
00457     const REAL a42 = a[42], a43 = a[43], a44 = a[44], a45 = a[45], a46 = a[46], a47 = a[47], a48 = a[48];
00458
00459     const REAL b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3], b4 = b[4], b5 = b[5], b6 = b[6];
00460     const REAL b7 = b[7], b8 = b[8], b9 = b[9], b10 = b[10], b11 = b[11], b12 = b[12], b13 = b[13];
00461     const REAL b14 = b[14], b15 = b[15], b16 = b[16], b17 = b[17], b18 = b[18], b19 = b[19], b20 = b[20];
00462     const REAL b21 = b[21], b22 = b[22], b23 = b[23], b24 = b[24], b25 = b[25], b26 = b[26], b27 = b[27];
00463     const REAL b28 = b[28], b29 = b[29], b30 = b[30], b31 = b[31], b32 = b[32], b33 = b[33], b34 = b[34];
00464     const REAL b35 = b[35], b36 = b[36], b37 = b[37], b38 = b[38], b39 = b[39], b40 = b[40], b41 = b[41];
00465     const REAL b42 = b[42], b43 = b[43], b44 = b[44], b45 = b[45], b46 = b[46], b47 = b[47], b48 = b[48];
00466
00467     c[0] = a0*b0 + a1*b7 + a2*b14 + a3*b21 + a4*b28 + a5*b35 + a6*b42;
00468     c[1] = a0*b1 + a1*b8 + a2*b15 + a3*b22 + a4*b29 + a5*b36 + a6*b43;
00469     c[2] = a0*b2 + a1*b9 + a2*b16 + a3*b23 + a4*b30 + a5*b37 + a6*b44;
00470     c[3] = a0*b3 + a1*b10 + a2*b17 + a3*b24 + a4*b31 + a5*b38 + a6*b45;
00471     c[4] = a0*b4 + a1*b11 + a2*b18 + a3*b25 + a4*b32 + a5*b39 + a6*b46;
00472     c[5] = a0*b5 + a1*b12 + a2*b19 + a3*b26 + a4*b33 + a5*b40 + a6*b47;
00473     c[6] = a0*b6 + a1*b13 + a2*b20 + a3*b27 + a4*b34 + a5*b41 + a6*b48;
00474
00475     c[7] = a7*b0 + a8*b7 + a9*b14 + a10*b21 + a11*b28 + a12*b35 + a13*b42;
00476     c[8] = a7*b1 + a8*b8 + a9*b15 + a10*b22 + a11*b29 + a12*b36 + a13*b43;
00477     c[9] = a7*b2 + a8*b9 + a9*b16 + a10*b23 + a11*b30 + a12*b37 + a13*b44;
00478     c[10] = a7*b3 + a8*b10 + a9*b17 + a10*b24 + a11*b31 + a12*b38 + a13*b45;
00479     c[11] = a7*b4 + a8*b11 + a9*b18 + a10*b25 + a11*b32 + a12*b39 + a13*b46;
00480     c[12] = a7*b5 + a8*b12 + a9*b19 + a10*b26 + a11*b33 + a12*b40 + a13*b47;
00481     c[13] = a7*b6 + a8*b13 + a9*b20 + a10*b27 + a11*b34 + a12*b41 + a13*b48;
00482
00483     c[14] = a14*b0 + a15*b7 + a16*b14 + a17*b21 + a18*b28 + a19*b35 + a20*b42;
00484     c[15] = a14*b1 + a15*b8 + a16*b15 + a17*b22 + a18*b29 + a19*b36 + a20*b43;
00485     c[16] = a14*b2 + a15*b9 + a16*b16 + a17*b23 + a18*b30 + a19*b37 + a20*b44;
00486     c[17] = a14*b3 + a15*b10 + a16*b17 + a17*b24 + a18*b31 + a19*b38 + a20*b45;
00487     c[18] = a14*b4 + a15*b11 + a16*b18 + a17*b25 + a18*b32 + a19*b39 + a20*b46;
00488     c[19] = a14*b5 + a15*b12 + a16*b19 + a17*b26 + a18*b33 + a19*b40 + a20*b47;
00489     c[20] = a14*b6 + a15*b13 + a16*b20 + a17*b27 + a18*b34 + a19*b41 + a20*b48;
00490
00491     c[21] = a21*b0 + a22*b7 + a23*b14 + a24*b21 + a25*b28 + a26*b35 + a27*b42;
00492     c[22] = a21*b1 + a22*b8 + a23*b15 + a24*b22 + a25*b29 + a26*b36 + a27*b43;
00493     c[23] = a21*b2 + a22*b9 + a23*b16 + a24*b23 + a25*b30 + a26*b37 + a27*b44;

```

```

00494 c[24] = a21*b3 + a22*b10 + a23*b17 + a24*b24 + a25*b31 + a26*b38 + a27*b45;
00495 c[25] = a21*b4 + a22*b11 + a23*b18 + a24*b25 + a25*b32 + a26*b39 + a27*b46;
00496 c[26] = a21*b5 + a22*b12 + a23*b19 + a24*b26 + a25*b33 + a26*b40 + a27*b47;
00497 c[27] = a21*b6 + a22*b13 + a23*b20 + a24*b27 + a25*b34 + a26*b41 + a27*b48;
00498
00499 c[28] = a28*b0 + a29*b7 + a30*b14 + a31*b21 + a32*b28 + a33*b35 + a34*b42;
00500 c[29] = a28*b1 + a29*b8 + a30*b15 + a31*b22 + a32*b29 + a33*b36 + a34*b43;
00501 c[30] = a28*b2 + a29*b9 + a30*b16 + a31*b23 + a32*b30 + a33*b37 + a34*b44;
00502 c[31] = a28*b3 + a29*b10 + a30*b17 + a31*b24 + a32*b31 + a33*b38 + a34*b45;
00503 c[32] = a28*b4 + a29*b11 + a30*b18 + a31*b25 + a32*b32 + a33*b39 + a34*b46;
00504 c[33] = a28*b5 + a29*b12 + a30*b19 + a31*b26 + a32*b33 + a33*b40 + a34*b47;
00505 c[34] = a28*b6 + a29*b13 + a30*b20 + a31*b27 + a32*b34 + a33*b41 + a34*b48;
00506
00507 c[35] = a35*b0 + a36*b7 + a37*b14 + a38*b21 + a39*b28 + a40*b35 + a41*b42;
00508 c[36] = a35*b1 + a36*b8 + a37*b15 + a38*b22 + a39*b29 + a40*b36 + a41*b43;
00509 c[37] = a35*b2 + a36*b9 + a37*b16 + a38*b23 + a39*b30 + a40*b37 + a41*b44;
00510 c[38] = a35*b3 + a36*b10 + a37*b17 + a38*b24 + a39*b31 + a40*b38 + a41*b45;
00511 c[39] = a35*b4 + a36*b11 + a37*b18 + a38*b25 + a39*b32 + a40*b39 + a41*b46;
00512 c[40] = a35*b5 + a36*b12 + a37*b19 + a38*b26 + a39*b33 + a40*b40 + a41*b47;
00513 c[41] = a35*b6 + a36*b13 + a37*b20 + a38*b27 + a39*b34 + a40*b41 + a41*b48;
00514
00515 c[42] = a42*b0 + a43*b7 + a44*b14 + a45*b21 + a46*b28 + a47*b35 + a48*b42;
00516 c[43] = a42*b1 + a43*b8 + a44*b15 + a45*b22 + a46*b29 + a47*b36 + a48*b43;
00517 c[44] = a42*b2 + a43*b9 + a44*b16 + a45*b23 + a46*b30 + a47*b37 + a48*b44;
00518 c[45] = a42*b3 + a43*b10 + a44*b17 + a45*b24 + a46*b31 + a47*b38 + a48*b45;
00519 c[46] = a42*b4 + a43*b11 + a44*b18 + a45*b25 + a46*b32 + a47*b39 + a48*b46;
00520 c[47] = a42*b5 + a43*b12 + a44*b19 + a45*b26 + a46*b33 + a47*b40 + a48*b47;
00521 c[48] = a42*b6 + a43*b13 + a44*b20 + a45*b27 + a46*b34 + a47*b41 + a48*b48;
00522 }
00523
00540 void fasp_blas_smat_mul (const REAL *a,
00541                          const REAL *b,
00542                          REAL *c,
00543                          const INT n)
00544 {
00545
00546     switch (n) {
00547     case 2:
00548         fasp_blas_smat_mul_nc2(a, b, c); break;
00549
00550     case 3:
00551         fasp_blas_smat_mul_nc3(a, b, c); break;
00552
00553     case 5:
00554         fasp_blas_smat_mul_nc5(a, b, c); break;
00555
00556     case 7:
00557         fasp_blas_smat_mul_nc7(a, b, c); break;
00558
00559     default: {
00560         const INT n2 = n*n;
00561         INT i,j,k;
00562         REAL temp;
00563
00564         for (i=0; i<n2; i+=n) {
00565             for (j=0; j<n; ++j) {
00566                 temp = 0.0; // Fixed by Chensong. Feb/22/2011.
00567                 for (k=0; k<n; ++k) temp += a[i+k]*b[k*n+j];
00568                 c[i+j] = temp;
00569             } // end for j
00570         } // end for i
00571     }
00572     break;
00573 }
00574 return;
00575 }
00576
00589 void fasp_blas_smat_ypAx_nc2 (const REAL *A,
00590                              const REAL *x,
00591                              REAL *y)
00592 {
00593     const REAL x0 = x[0], x1 = x[1];
00594
00595     y[0] += A[0]*x0 + A[1]*x1;
00596     y[1] += A[2]*x0 + A[3]*x1;
00597
00598     return;
00599 }
00600
00613 void fasp_blas_smat_ypAx_nc3 (const REAL *A,
00614                              const REAL *x,

```

```

00615                                     REAL          *y)
00616 {
00617     const REAL  x0 = x[0], x1 = x[1], x2 = x[2];
00618
00619     y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2;
00620     y[1] += A[3]*x0 + A[4]*x1 + A[5]*x2;
00621     y[2] += A[6]*x0 + A[7]*x1 + A[8]*x2;
00622     return;
00623 }
00624
00637 void fasp_blas_smat_ypAx_nc4 (const REAL  *A,
00638                               const REAL  *x,
00639                               REAL        *y)
00640 {
00641     const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00642
00643     y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3;
00644     y[1] += A[4]*x0 + A[5]*x1 + A[6]*x2 + A[7]*x3;
00645     y[2] += A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
00646     y[3] += A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3;
00647     return;
00648 }
00649
00662 void fasp_blas_smat_ypAx_nc5 (const REAL  *A,
00663                               const REAL  *x,
00664                               REAL        *y)
00665 {
00666     const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
00667
00668     y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
00669     y[1] += A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
00670     y[2] += A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
00671     y[3] += A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
00672     y[4] += A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
00673     return;
00674 }
00675
00688 void fasp_blas_smat_ypAx_nc7 (const REAL  *A,
00689                               const REAL  *x,
00690                               REAL        *y)
00691 {
00692     const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00693     const REAL  x4 = x[4], x5 = x[5], x6 = x[6];
00694
00695     y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5 + A[6]*x6;
00696     y[1] += A[7]*x0 + A[8]*x1 + A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
00697     y[2] += A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
00698     y[3] += A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
00699     y[4] += A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
00700     y[5] += A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
00701     y[6] += A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
00702     return;
00703 }
00704
00720 void fasp_blas_smat_ypAx (const REAL  *A,
00721                            const REAL  *x,
00722                            REAL        *y,
00723                            const INT   n)
00724 {
00725     switch (n) {
00726     case 1:
00727     {
00728         y[0] += A[0]*x[0];
00729         break;
00730     }
00731     case 2:
00732     {
00733         const REAL  x0 = x[0], x1 = x[1];
00734         y[0] += A[0]*x0 + A[1]*x1;
00735         y[1] += A[2]*x0 + A[3]*x1;
00736         break;
00737     }
00738     case 3:
00739     {
00740         const REAL  x0 = x[0], x1 = x[1], x2 = x[2];
00741         y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2;
00742         y[1] += A[3]*x0 + A[4]*x1 + A[5]*x2;
00743         y[2] += A[6]*x0 + A[7]*x1 + A[8]*x2;
00744         break;
00745     }
00746     case 4:

```

```

00747     {
00748         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00749         y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3;
00750         y[1] += A[4]*x0 + A[5]*x1 + A[6]*x2 + A[7]*x3;
00751         y[2] += A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
00752         y[3] += A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3;
00753         break;
00754     }
00755     case 5:
00756     {
00757         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
00758         y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
00759         y[1] += A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
00760         y[2] += A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
00761         y[3] += A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
00762         y[4] += A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
00763         break;
00764     }
00765     case 6:
00766     {
00767         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00768         const REAL x4 = x[4], x5 = x[5];
00769         y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5;
00770         y[1] += A[6]*x0 + A[7]*x1 + A[8]*x2 + A[9]*x3 + A[10]*x4 + A[11]*x5;
00771         y[2] += A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3 + A[16]*x4 + A[17]*x5;
00772         y[3] += A[18]*x0 + A[19]*x1 + A[20]*x2 + A[21]*x3 + A[22]*x4 + A[23]*x5;
00773         y[4] += A[24]*x0 + A[25]*x1 + A[26]*x2 + A[27]*x3 + A[28]*x4 + A[29]*x5;
00774         y[5] += A[30]*x0 + A[31]*x1 + A[32]*x2 + A[33]*x3 + A[34]*x4 + A[35]*x5;
00775         break;
00776     }
00777     case 7:
00778     {
00779         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00780         const REAL x4 = x[4], x5 = x[5], x6 = x[6];
00781         y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5 + A[6]*x6;
00782         y[1] += A[7]*x0 + A[8]*x1 + A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
00783         y[2] += A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
00784         y[3] += A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
00785         y[4] += A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
00786         y[5] += A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
00787         y[6] += A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
00788         break;
00789     }
00790     default: /* For everything beyond 7 */
00791     {
00792         INT i,j,k;
00793
00794         for ( k = i = 0; i < n; i++, k+=n ) {
00795             for ( j = 0; j < n; j++ ) {
00796                 y[i] += A[k+j]*x[j];
00797             }
00798         }
00799         break;
00800     }
00801 }
00802
00803 return;
00804 }
00805
00820 void fasp_blas_smat_ymAx_nc2 (const REAL *A,
00821                               const REAL *x,
00822                               REAL *y)
00823 {
00824     const REAL x0 = x[0], x1 = x[1];
00825
00826     y[0] -= A[0]*x0 + A[1]*x1;
00827     y[1] -= A[2]*x0 + A[3]*x1;
00828
00829     return;
00830 }
00831
00846 void fasp_blas_smat_ymAx_nc3 (const REAL *A,
00847                               const REAL *x,
00848                               REAL *y)
00849 {
00850     const REAL x0 = x[0], x1 = x[1], x2 = x[2];
00851
00852     y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2;
00853     y[1] -= A[3]*x0 + A[4]*x1 + A[5]*x2;
00854     y[2] -= A[6]*x0 + A[7]*x1 + A[8]*x2;
00855

```



```

00856     return;
00857 }
00858
00873 void fasp_blas_smat_ymAx_nc4 (const REAL *A,
00874                               const REAL *x,
00875                               REAL *y)
00876 {
00877     const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00878
00879     y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3;
00880     y[1] -= A[4]*x0 + A[5]*x1 + A[6]*x2 + A[7]*x3;
00881     y[2] -= A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
00882     y[3] -= A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3;
00883     return;
00884 }
00885
00900 void fasp_blas_smat_ymAx_nc5 (const REAL *A,
00901                               const REAL *x,
00902                               REAL *y)
00903 {
00904     const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
00905
00906     y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
00907     y[1] -= A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
00908     y[2] -= A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
00909     y[3] -= A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
00910     y[4] -= A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
00911
00912     return;
00913 }
00914
00929 void fasp_blas_smat_ymAx_nc7 (const REAL *A,
00930                               const REAL *x,
00931                               REAL *y)
00932 {
00933     const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00934     const REAL x4 = x[4], x5 = x[5], x6 = x[6];
00935
00936     y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5 + A[6]*x6;
00937     y[1] -= A[7]*x0 + A[8]*x1 + A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
00938     y[2] -= A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
00939     y[3] -= A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
00940     y[4] -= A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
00941     y[5] -= A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
00942     y[6] -= A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
00943
00944     return;
00945 }
00946
00962 void fasp_blas_smat_ymAx (const REAL *A,
00963                           const REAL *x,
00964                           REAL *y,
00965                           const INT n)
00966 {
00967     switch (n) {
00968     case 1:
00969     {
00970         y[0] -= A[0]*x[0];
00971         break;
00972     }
00973     case 2:
00974     {
00975         const REAL x0 = x[0], x1 = x[1];
00976         y[0] -= A[0]*x0 + A[1]*x1;
00977         y[1] -= A[2]*x0 + A[3]*x1;
00978         break;
00979     }
00980     case 3:
00981     {
00982         const REAL x0 = x[0], x1 = x[1], x2 = x[2];
00983         y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2;
00984         y[1] -= A[3]*x0 + A[4]*x1 + A[5]*x2;
00985         y[2] -= A[6]*x0 + A[7]*x1 + A[8]*x2;
00986         break;
00987     }
00988     case 4:
00989     {
00990         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
00991         y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3;
00992         y[1] -= A[4]*x0 + A[5]*x1 + A[6]*x2 + A[7]*x3;
00993         y[2] -= A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;

```

```

00994         y[3] -= A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3;
00995         break;
00996     }
00997     case 5:
00998     {
00999         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
01000         y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
01001         y[1] -= A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
01002         y[2] -= A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
01003         y[3] -= A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
01004         y[4] -= A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
01005         break;
01006     }
01007     case 6:
01008     {
01009         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
01010         const REAL x4 = x[4], x5 = x[5];
01011         y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5;
01012         y[1] -= A[6]*x0 + A[7]*x1 + A[8]*x2 + A[9]*x3 + A[10]*x4 + A[11]*x5;
01013         y[2] -= A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3 + A[16]*x4 + A[17]*x5;
01014         y[3] -= A[18]*x0 + A[19]*x1 + A[20]*x2 + A[21]*x3 + A[22]*x4 + A[23]*x5;
01015         y[4] -= A[24]*x0 + A[25]*x1 + A[26]*x2 + A[27]*x3 + A[28]*x4 + A[29]*x5;
01016         y[5] -= A[30]*x0 + A[31]*x1 + A[32]*x2 + A[33]*x3 + A[34]*x4 + A[35]*x5;
01017         break;
01018     }
01019     case 7:
01020     {
01021         const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
01022         const REAL x4 = x[4], x5 = x[5], x6 = x[6];
01023         y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4 + A[5]*x5 + A[6]*x6;
01024         y[1] -= A[7]*x0 + A[8]*x1 + A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
01025         y[2] -= A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
01026         y[3] -= A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
01027         y[4] -= A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
01028         y[5] -= A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
01029         y[6] -= A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
01030         break;
01031     }
01032     default: // Everything beyond 7
01033     {
01034         INT i,j,k;
01035
01036         for ( k = i = 0; i < n; i++, k+=n ) {
01037             for ( j = 0; j < n; j++ ) {
01038                 y[i] -= A[k+j]*x[j];
01039             }
01040         }
01041         break;
01042     }
01043 }
01044
01045 return;
01046 }
01047
01064 void fasp_blas_smat_aAxpby (const REAL alpha,
01065                             const REAL *A,
01066                             const REAL *x,
01067                             const REAL beta,
01068                             REAL *y,
01069                             const INT n)
01070 {
01071     INT i,j,k;
01072     REAL tmp = 0.0;
01073
01074     if ( alpha == 0 ) {
01075         for ( i = 0; i < n; i ++ ) y[i] *= beta;
01076         return;
01077     }
01078
01079     // y := (beta/alpha)y
01080     tmp = beta / alpha;
01081     if ( tmp != 1.0 ) {
01082         for ( i = 0; i < n; i ++ ) y[i] *= tmp;
01083     }
01084
01085     // y := y + A*x
01086     for ( k = i = 0; i < n; i++, k+=n ) {
01087         for ( j = 0; j < n; j ++ ) {
01088             y[i] += A[k+j]*x[j];
01089         }
01090     }

```

```

01091
01092     // y := alpha*y
01093     if ( alpha != 1.0 ) {
01094         for ( i = 0; i < n; i ++ ) y[i] *= alpha;
01095     }
01096 }
01097
01098 /*-----*/
01099 /*--      End of File      --*/
01100 /*-----*/

```

9.67 BlaSmallMatInv.c File Reference

Find inversion of *small* dense matrices in row-major format.

```

#include "fasp.h"
#include "fasp_functs.h"

```

Macros

- #define [SWAP](#)(a, b) {temp=(a);(a)=(b);(b)=temp;}

Functions

- void [fasp_smat_inv_nc2](#) (REAL *a)
Compute the inverse matrix of a 2*2 full matrix A (in place)
- void [fasp_smat_inv_nc3](#) (REAL *a)
Compute the inverse matrix of a 3*3 full matrix A (in place)
- void [fasp_smat_inv_nc4](#) (REAL *a)
Compute the inverse matrix of a 4*4 full matrix A (in place)
- void [fasp_smat_inv_nc5](#) (REAL *a)
Compute the inverse matrix of a 5*5 full matrix A (in place)
- void [fasp_smat_inv_nc7](#) (REAL *a)
Compute the inverse matrix of a 7*7 matrix a.
- void [fasp_smat_inv_nc](#) (REAL *a, const INT n)
Compute the inverse of a matrix using Gauss Elimination.
- [SHORT fasp_smat_invp_nc](#) (REAL *a, const INT n)
Compute the inverse of a matrix using Gauss Elimination with Pivoting.
- [SHORT fasp_smat_inv](#) (REAL *a, const INT n)
Compute the inverse matrix of a small full matrix a.
- [REAL fasp_smat_Linf](#) (const REAL *A, const INT n)
Compute the L infinity norm of A.
- void [fasp_smat_identity_nc2](#) (REAL *a)
Set a 2*2 full matrix to be a identity.
- void [fasp_smat_identity_nc3](#) (REAL *a)
Set a 3*3 full matrix to be a identity.
- void [fasp_smat_identity_nc5](#) (REAL *a)
Set a 5*5 full matrix to be a identity.
- void [fasp_smat_identity_nc7](#) (REAL *a)
Set a 7*7 full matrix to be a identity.
- void [fasp_smat_identity](#) (REAL *a, const INT n, const INT n2)
Set a n*n full matrix to be a identity.

9.67.1 Detailed Description

Find inversion of *small* dense matrices in row-major format.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSmallMatInv.c](#).

9.67.2 Macro Definition Documentation

9.67.2.1 SWAP

```
#define SWAP(  
    a,  
    b ) {temp=(a); (a)=(b); (b)=temp; }
```

swap two numbers

Definition at line 17 of file [BlaSmallMatInv.c](#).

9.67.3 Function Documentation

9.67.3.1 fasp_smat_identity()

```
void fasp_smat_identity (  
    REAL * a,  
    const INT n,  
    const INT n2 )
```

Set a n*n full matrix to be a identity.

Parameters

<i>a</i>	Pointer to the REAL vector which stands for a n*n full matrix
<i>n</i>	Size of full matrix
<i>n2</i>	Length of the REAL vector which stores the n*n full matrix

Author

Xiaozhe Hu

Date

2010/12/25

Definition at line 754 of file [BlaSmallMatInv.c](#).

9.67.3.2 fasp_smat_identity_nc2()

```
void fasp_smat_identity_nc2 (
    REAL * a )
```

Set a 2*2 full matrix to be a identity.

Parameters

<i>a</i>	Pointer to the REAL vector which stands for a 2*2 full matrix
----------	---

Author

Xiaozhe Hu

Date

2011/11/18

Definition at line 674 of file [BlaSmallMatInv.c](#).

9.67.3.3 fasp_smat_identity_nc3()

```
void fasp_smat_identity_nc3 (
    REAL * a )
```

Set a 3*3 full matrix to be a identity.

Parameters

<i>a</i>	Pointer to the REAL vector which stands for a 3*3 full matrix
----------	---

Author

Xiaozhe Hu

Date

2010/12/25

Definition at line 691 of file [BlaSmallMatInv.c](#).

9.67.3.4 fasp_smat_identity_nc5()

```
void fasp_smat_identity_nc5 (
    REAL * a )
```

Set a 5*5 full matrix to be a identity.

Parameters

<i>a</i>	Pointer to the REAL vector which stands for a 5*5 full matrix
----------	---

Author

Xiaozhe Hu

Date

2010/12/25

Definition at line 708 of file [BlaSmallMatInv.c](#).**9.67.3.5 fasp_smat_identity_nc7()**

```
void fasp_smat_identity_nc7 (
    REAL * a )
```

Set a 7*7 full matrix to be a identity.

Parameters

<i>a</i>	Pointer to the REAL vector which stands for a 7*7 full matrix
----------	---

Author

Xiaozhe Hu

Date

2010/12/25

Definition at line 729 of file [BlaSmallMatInv.c](#).**9.67.3.6 fasp_smat_inv()**

```
SHORT fasp_smat_inv (
    REAL * a,
    const INT n )
```

Compute the inverse matrix of a small full matrix a.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>n</i>	Dimension of the matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

04/21/2010

Definition at line 603 of file [BlaSmallMatInv.c](#).

9.67.3.7 fasp_smat_inv_nc()

```
void fasp_smat_inv_nc (  
    REAL * a,  
    const INT n )
```

Compute the inverse of a matrix using Gauss Elimination.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>n</i>	Dimension of the matrix

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 441 of file [BlaSmallMatInv.c](#).

9.67.3.8 fasp_smat_inv_nc2()

```
void fasp_smat_inv_nc2 (  
    REAL * a )
```

Compute the inverse matrix of a 2*2 full matrix A (in place)

Parameters

<i>a</i>	Pointer to the REAL array which stands a 2*2 matrix
----------	---

Author

Xiaozhe Hu

Date

18/11/2011

Definition at line 33 of file [BlaSmallMatInv.c](#).

9.67.3.9 fasp_smat_inv_nc3()

```
void fasp_smat_inv_nc3 (  
    REAL * a )
```

Compute the inverse matrix of a 3*3 full matrix A (in place)

Parameters

<i>a</i>	Pointer to the REAL array which stands a 3*3 matrix
----------	---

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 67 of file [BlaSmallMatInv.c](#).

9.67.3.10 fasp_smat_inv_nc4()

```
void fasp_smat_inv_nc4 (  
    REAL * a )
```

Compute the inverse matrix of a 4*4 full matrix A (in place)

Parameters

<i>a</i>	Pointer to the REAL array which stands a 4*4 matrix
----------	---

Author

Xiaozhe Hu

Date

01/12/2013

Modified by Hongxuan Zhang on 06/13/2014: Fix a bug in M23.

Definition at line 111 of file [BlaSmallMatInv.c](#).

9.67.3.11 fasp_smat_inv_nc5()

```
void fasp_smat_inv_nc5 (  
    REAL * a )
```

Compute the inverse matrix of a 5*5 full matrix A (in place)

Parameters

<i>a</i>	Pointer to the REAL array which stands a 5*5 matrix
----------	---

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 170 of file [BlaSmallMatInv.c](#).

9.67.3.12 fasp_smat_inv_nc7()

```
void fasp_smat_inv_nc7 (  
    REAL * a )
```


Compute the inverse matrix of a 7*7 matrix a.

Parameters

<i>a</i>	Pointer to the REAL array which stands a 7*7 matrix
----------	---

Note

This is NOT implemented yet!

Author

Xiaozhe Hu, Shiquan Zhang

Date

05/01/2010

Definition at line 425 of file [BlaSmallMatInv.c](#).

9.67.3.13 fasp_smat_invp_nc()

```
SHORT fasp_smat_invp_nc (  
    REAL * a,  
    const INT n )
```

Compute the inverse of a matrix using Gauss Elimination with Pivoting.

Parameters

<i>a</i>	Pointer to the REAL array which stands a n*n matrix
<i>n</i>	Dimension of the matrix

Author

Chensong Zhang

Date

04/03/2015

Note

This routine is based on gaussj() from "Numerical Recipies in C"!

Definition at line 508 of file [BlaSmallMatInv.c](#).

9.67.3.14 fasp_smat_Linf()

```
REAL fasp_smat_Linf (  
    const REAL * A,  
    const INT n )
```

Compute the L infinity norm of A.

Parameters

A	Pointer to the $n \times n$ dense matrix
n	the dimension of the dense matrix

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 646 of file [BlaSmallMatInv.c](#).

9.68 BlaSmallMatInv.c

[Go to the documentation of this file.](#)

```

00001
00014 #include "fasp.h"
00015 #include "fasp_funcs.h"
00016
00017 #define SWAP(a,b) {temp=(a); (a)=(b); (b)=temp;}
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00033 void fasp_smat_inv_nc2 (REAL *a)
00034 {
00035     const REAL a0 = a[0], a1 = a[1];
00036     const REAL a2 = a[2], a3 = a[3];
00037
00038     const REAL det = a0*a3 - a1*a2;
00039
00040     if ( ABS(det) < SMALLREAL ) {
00041         printf("### WARNING: Matrix is nearly singular, det = %e! Ignore.\n", det);
00042         printf("###-----\n");
00043         printf("### %12.5e %12.5e \n", a0, a1);
00044         printf("### %12.5e %12.5e \n", a2, a3);
00045         printf("###-----\n");
00046
00047         a[0] = 1.0; a[1] = 0.0;
00048         a[2] = 0.0; a[3] = 1.0;
00049     }
00050     else {
00051         REAL det_inv = 1.0 / det;
00052         a[0] = a3 * det_inv; a[1] = -a1 * det_inv;
00053         a[2] = -a2 * det_inv; a[3] = a0 * det_inv;
00054     }
00055 }
00056
00067 void fasp_smat_inv_nc3 (REAL *a)
00068 {
00069     const REAL a0 = a[0], a1 = a[1], a2 = a[2];
00070     const REAL a3 = a[3], a4 = a[4], a5 = a[5];
00071     const REAL a6 = a[6], a7 = a[7], a8 = a[8];
00072
00073     const REAL M0 = a4*a8-a5*a7, M3 = a2*a7-a1*a8, M6 = a1*a5-a2*a4;
00074     const REAL M1 = a5*a6-a3*a8, M4 = a0*a8-a2*a6, M7 = a2*a3-a0*a5;
00075     const REAL M2 = a3*a7-a4*a6, M5 = a1*a6-a0*a7, M8 = a0*a4-a1*a3;
00076
00077     const REAL det = a0*M0+a3*M3+a6*M6;
00078
00079     if ( ABS(det) < SMALLREAL ) {
00080         printf("### WARNING: Matrix is nearly singular, det = %e! Ignore.\n", det);
00081         printf("###-----\n");
00082         printf("### %12.5e %12.5e %12.5e \n", a0, a1, a2);
00083         printf("### %12.5e %12.5e %12.5e \n", a3, a4, a5);
00084         printf("### %12.5e %12.5e %12.5e \n", a6, a7, a8);
00085         printf("###-----\n");
00086
00087         a[0] = 1.0; a[1] = 0.0; a[2] = 0.0;

```

```

00088         a[3] = 0.0; a[4] = 1.0; a[5] = 0.0;
00089         a[6] = 0.0; a[7] = 0.0; a[8] = 1.0;
00090     }
00091     else {
00092         REAL det_inv = 1.0/det;
00093         a[0] = M0*det_inv; a[1] = M3*det_inv; a[2] = M6*det_inv;
00094         a[3] = M1*det_inv; a[4] = M4*det_inv; a[5] = M7*det_inv;
00095         a[6] = M2*det_inv; a[7] = M5*det_inv; a[8] = M8*det_inv;
00096     }
00097 }
00098
00111 void fasp_smat_inv_nc4 (REAL *a)
00112 {
00113     const REAL a11 = a[0], a12 = a[1], a13 = a[2], a14 = a[3];
00114     const REAL a21 = a[4], a22 = a[5], a23 = a[6], a24 = a[7];
00115     const REAL a31 = a[8], a32 = a[9], a33 = a[10], a34 = a[11];
00116     const REAL a41 = a[12], a42 = a[13], a43 = a[14], a44 = a[15];
00117
00118     const REAL M11 = a22*a33*a44 + a23*a34*a42 + a24*a32*a43 - a22*a34*a43 - a23*a32*a44 - a24*a33*a42;
00119     const REAL M12 = a12*a34*a43 + a13*a32*a44 + a14*a33*a42 - a12*a33*a44 - a13*a34*a42 - a14*a32*a43;
00120     const REAL M13 = a12*a23*a44 + a13*a24*a42 + a14*a22*a43 - a12*a24*a43 - a13*a22*a44 - a14*a23*a42;
00121     const REAL M14 = a12*a24*a33 + a13*a22*a34 + a14*a23*a32 - a12*a23*a34 - a13*a24*a32 - a14*a22*a33;
00122     const REAL M21 = a21*a34*a43 + a23*a31*a44 + a24*a33*a41 - a21*a33*a44 - a23*a34*a41 - a24*a31*a43;
00123     const REAL M22 = a11*a33*a44 + a13*a34*a41 + a14*a31*a43 - a11*a34*a43 - a13*a31*a44 - a14*a33*a41;
00124     const REAL M23 = a11*a24*a43 + a13*a21*a44 + a14*a23*a41 - a11*a23*a44 - a13*a24*a41 - a14*a21*a43;
00125     const REAL M24 = a11*a23*a34 + a13*a24*a31 + a14*a21*a33 - a11*a24*a33 - a13*a21*a34 - a14*a23*a31;
00126     const REAL M31 = a21*a32*a44 + a22*a34*a41 + a24*a31*a42 - a21*a34*a42 - a22*a31*a44 - a24*a32*a41;
00127     const REAL M32 = a11*a34*a42 + a12*a31*a44 + a14*a32*a41 - a11*a32*a44 - a12*a34*a41 - a14*a31*a42;
00128     const REAL M33 = a11*a22*a44 + a12*a24*a41 + a14*a21*a42 - a11*a24*a42 - a12*a21*a44 - a14*a22*a41;
00129     const REAL M34 = a11*a24*a32 + a12*a21*a34 + a14*a22*a31 - a11*a22*a34 - a12*a24*a31 - a14*a21*a32;
00130     const REAL M41 = a21*a33*a42 + a22*a31*a43 + a23*a32*a41 - a21*a32*a43 - a22*a33*a41 - a23*a31*a42;
00131     const REAL M42 = a11*a32*a43 + a12*a33*a41 + a13*a31*a42 - a11*a33*a42 - a12*a31*a43 - a13*a32*a41;
00132     const REAL M43 = a11*a23*a42 + a12*a21*a43 + a13*a22*a41 - a11*a22*a43 - a12*a23*a41 - a13*a21*a42;
00133     const REAL M44 = a11*a22*a33 + a12*a23*a31 + a13*a21*a32 - a11*a23*a32 - a12*a21*a33 - a13*a22*a31;
00134
00135     const REAL det = a11*M11 + a12*M21 + a13*M31 + a14*M41;
00136
00137     if ( ABS(det) < SMALLREAL ) {
00138         printf("### WARNING: Matrix is nearly singular, det = %e! Ignore.\n", det);
00139         printf("###-----\n");
00140         printf("### %12.5e %12.5e %12.5e %12.5e\n", a11, a12, a13, a14);
00141         printf("### %12.5e %12.5e %12.5e %12.5e\n", a21, a22, a23, a24);
00142         printf("### %12.5e %12.5e %12.5e %12.5e\n", a31, a32, a33, a34);
00143         printf("### %12.5e %12.5e %12.5e %12.5e\n", a41, a42, a43, a44);
00144         printf("###-----\n");
00145
00146         a[0] = 1.0; a[1] = 0.0; a[2] = 0.0; a[3] = 0.0;
00147         a[4] = 0.0; a[5] = 1.0; a[6] = 0.0; a[7] = 0.0;
00148         a[8] = 0.0; a[9] = 0.0; a[10] = 1.0; a[11] = 0.0;
00149         a[12] = 0.0; a[13] = 0.0; a[14] = 0.0; a[15] = 1.0;
00150     }
00151     else {
00152         REAL det_inv = 1.0 / det;
00153         a[0] = M11 * det_inv; a[1] = M12 * det_inv; a[2] = M13 * det_inv; a[3] = M14 * det_inv;
00154         a[4] = M21 * det_inv; a[5] = M22 * det_inv; a[6] = M23 * det_inv; a[7] = M24 * det_inv;
00155         a[8] = M31 * det_inv; a[9] = M32 * det_inv; a[10] = M33 * det_inv; a[11] = M34 * det_inv;
00156         a[12] = M41 * det_inv; a[13] = M42 * det_inv; a[14] = M43 * det_inv; a[15] = M44 * det_inv;
00157     }
00158 }
00159
00170 void fasp_smat_inv_nc5 (REAL *a)
00171 {
00172     const REAL a0=a[0], a1=a[1], a2=a[2], a3=a[3], a4=a[4];
00173     const REAL a5=a[5], a6=a[6], a7=a[7], a8=a[8], a9=a[9];
00174     const REAL a10=a[10], a11=a[11], a12=a[12], a13=a[13], a14=a[14];
00175     const REAL a15=a[15], a16=a[16], a17=a[17], a18=a[18], a19=a[19];
00176     const REAL a20=a[20], a21=a[21], a22=a[22], a23=a[23], a24=a[24];
00177
00178     REAL det0, det1, det2, det3, det4, det;
00179
00180     det0 = a6 * ( a12 * (a18*a24-a19*a23) + a17 * (a14*a23-a13*a24) + a22 * (a13*a19 - a14*a18) );
00181     det0 += a11 * ( a7 * (a19*a23-a18*a24) + a17 * (a8*a24 -a9*a23 ) + a22 * (a9*a18 - a8*a19) );
00182     det0 += a16 * ( a7 * (a13*a24-a14*a23) + a12 * (a9*a23 -a8*a24 ) + a22 * (a8*a14 - a9*a13) );
00183     det0 += a21 * ( a17 * (a9*a13 -a8*a14 ) + a7 * (a14*a18-a13*a19) + a12 * (a8*a19 - a9*a18) );
00184
00185     det1 = a1 * ( a22 * (a14*a18-a13*a19) + a12 * (a19*a23-a18*a24) + a17 * (a13*a24 - a14*a23) );
00186     det1 += a11 * ( a17 * (a4*a23 - a3*a24) + a2 * (a18*a24-a19*a23) + a22 * (a3*a19 - a4*a18) );
00187     det1 += a16 * ( a12 * (a3*a24 - a4*a23) + a2 * (a14*a23-a13*a24) + a22 * (a4*a13 - a3*a14) );
00188     det1 += a21 * ( a2 * (a13*a19-a14*a18) + a12 * (a4*a18 -a3*a19 ) + a17 * (a3*a14 - a4*a13) );
00189
00190     det2 = a1 * ( a7 * (a18*a24-a19*a23) + a17 * (a9*a23-a8*a24) + a22 * (a8*a19 - a9*a18) );

```

```

00191     det2 += a6 * ( a2 * (a19*a23-a18*a24) + a17 * (a3*a24-a4*a23) + a22 * (a4*a18 - a3*a19) );
00192     det2 += a16 * ( a2 * (a8*a24 -a9*a23 ) + a7 * (a4*a23-a3*a24) + a22 * (a3*a9 - a4*a8) );
00193     det2 += a21 * ( a7 * (a3*a19 -a4*a18 ) + a2 * (a9*a18-a8*a19) + a17 * (a4*a8 - a3*a9) );
00194
00195     det3 = a1 * ( a12* (a8*a24 -a9*a23) + a7 * (a14*a23-a13*a24) + a22 * (a9*a13 - a8*a14) );
00196     det3 += a6 * ( a2 * (a13*a24-a14*a23) + a12 * (a4*a23 -a3*a24 ) + a22 * (a3*a14 - a4*a13) );
00197     det3 += a11 * ( a7 * (a3*a24 -a4*a23) + a2 * (a9*a23 -a8*a24 ) + a22 * (a4*a8 - a3*a9) );
00198     det3 += a21 * ( a2 * (a8*a14 -a9*a13) + a7 * (a4*a13 -a3*a14 ) + a12 * (a3*a9 - a4*a8) );
00199
00200     det4 = a1 * ( a7 * (a13*a19-a14*a18) + a12 * (a9*a18 -a8*a19 ) + a17 * (a8*a14 - a9*a13) );
00201     det4 += a6 * ( a12* (a3*a19 -a4*a18 ) + a17 * (a4*a13 -a3*a14 ) + a2 * (a14*a18- a13*a19));
00202     det4 += a11 * ( a2 * (a8*a19 -a9*a18 ) + a7 * (a4*a18 -a3*a19 ) + a17 * (a3*a9 - a4*a8) );
00203     det4 += a16 * ( a7 * (a3*a14 -a4*a13 ) + a2 * (a9*a13 -a8*a14 ) + a12 * (a4*a8 - a3*a9) );
00204
00205     det = det0*a0 + det1*a5+ det2*a10 + det3*a15 + det4*a20;
00206
00207     if ( ABS(det) < SMALLREAL ) {
00208         printf("### WARNING: Matrix is nearly singular, det = %e! Ignore.\n", det);
00209         printf("###-----\n");
00210         printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n", a0, a1, a2, a3, a4);
00211         printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n", a5, a6, a7, a8, a9);
00212         printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n", a10, a11, a12, a13, a14);
00213         printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n", a15, a16, a17, a18, a19);
00214         printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n", a20, a21, a22, a23, a24);
00215         printf("###-----\n");
00216
00217         a[0] = 1.0; a[1] = 0.0; a[2] = 0.0; a[3] = 0.0; a[4] = 0.0;
00218         a[5] = 0.0; a[6] = 1.0; a[7] = 0.0; a[8] = 0.0; a[9] = 0.0;
00219         a[10] = 0.0; a[11] = 0.0; a[12] = 1.0; a[13] = 0.0; a[14] = 0.0;
00220         a[15] = 0.0; a[16] = 0.0; a[17] = 0.0; a[18] = 1.0; a[19] = 0.0;
00221         a[20] = 0.0; a[21] = 0.0; a[22] = 0.0; a[23] = 0.0; a[24] = 1.0;
00222     }
00223     else {
00224         REAL det_inv = 1 / det;
00225
00226         a[0] = a6 * (a12 * a18 * a24 - a12 * a19 * a23 - a17 * a13 * a24 + a17 * a14 * a23 + a22 * a13 *
a19 - a22 * a14 * a18);
00227         a[0] += a11 * (a7 * a19 * a23 - a7 * a18 * a24 + a17 * a8 * a24 - a17 * a9 * a23 - a22 * a8 * a19
+ a22 * a9 * a18);
00228         a[0] += a16 * (a7 * a13 * a24 - a7 * a14 * a23 - a12 * a8 * a24 + a12 * a9 * a23 + a22 * a8 * a14
- a22 * a9 * a13);
00229         a[0] += a21 * (a7 * a14 * a18 - a7 * a13 * a19 + a12 * a8 * a19 - a12 * a9 * a18 - a17 * a8 * a14
+ a17 * a9 * a13);
00230         a[0] *= det_inv;
00231
00232         a[1] = a1 * (a12 * a19 * a23 - a12 * a18 * a24 + a22 * a14 * a18 - a17 * a14 * a23 - a22 * a13 *
a19 + a17 * a13 * a24);
00233         a[1] += a11 * (a22 * a3 * a19 + a2 * a18 * a24 - a17 * a3 * a24 - a22 * a4 * a18 - a2 * a19 * a23
+ a17 * a4 * a23);
00234         a[1] += a16 * (a12 * a3 * a24 - a12 * a4 * a23 - a22 * a3 * a14 + a2 * a14 * a23 + a22 * a4 * a13
- a2 * a13 * a24);
00235         a[1] += a21 * (a12 * a4 * a18 - a12 * a3 * a19 - a2 * a14 * a18 - a17 * a4 * a13 + a2 * a13 * a19
+ a17 * a3 * a14);
00236         a[1] *= det_inv;
00237
00238         a[2] = a1 * (a7 * a18 * a24 - a7 * a19 * a23 - a17 * a8 * a24 + a17 * a9 * a23 + a22 * a8 * a19 -
a22 * a9 * a18);
00239         a[2] += a6 * (a2 * a19 * a23 - a2 * a18 * a24 + a17 * a3 * a24 - a17 * a4 * a23 - a22 * a3 * a19 +
a22 * a4 * a18);
00240         a[2] += a16 * (a2 * a8 * a24 - a2 * a9 * a23 - a7 * a3 * a24 + a7 * a4 * a23 + a22 * a3 * a9 - a22
* a4 * a8);
00241         a[2] += a21 * (a2 * a9 * a18 - a2 * a8 * a19 + a7 * a3 * a19 - a7 * a4 * a18 - a17 * a3 * a9 + a17
* a4 * a8);
00242         a[2] *= det_inv;
00243
00244         a[3] = a1 * (a12 * a8 * a24 - a12 * a9 * a23 + a7 * a14 * a23 - a7 * a13 * a24 + a22 * a9 * a13 -
a22 * a8 * a14);
00245         a[3] += a6 * (a12 * a4 * a23 - a12 * a3 * a24 + a22 * a3 * a14 - a22 * a4 * a13 + a2 * a13 * a24 -
a2 * a14 * a23);
00246         a[3] += a11 * (a7 * a3 * a24 - a7 * a4 * a23 + a22 * a4 * a8 - a22 * a3 * a9 + a2 * a9 * a23 - a2
* a8 * a24);
00247         a[3] += a21 * (a12 * a3 * a9 - a12 * a4 * a8 + a2 * a8 * a14 - a2 * a9 * a13 + a7 * a4 * a13 - a7
* a3 * a14);
00248         a[3] *= det_inv;
00249
00250         a[4] = a1 * (a7 * a13 * a19 - a7 * a14 * a18 - a12 * a8 * a19 + a12 * a9 * a18 + a17 * a8 * a14 -
a17 * a9 * a13);
00251         a[4] += a6 * (a2 * a14 * a18 - a2 * a13 * a19 + a12 * a3 * a19 - a12 * a4 * a18 - a17 * a3 * a14 +
a17 * a4 * a13);
00252         a[4] += a11 * (a2 * a8 * a19 - a2 * a9 * a18 - a7 * a3 * a19 + a7 * a4 * a18 + a17 * a3 * a9 - a17
* a4 * a8);

```

```

00253     a[4] += a16 * (a2 * a9 * a13 - a2 * a8 * a14 + a7 * a3 * a14 - a7 * a4 * a13 - a12 * a3 * a9 + a12
    * a4 * a8);
00254     a[4] *= det_inv;
00255
00256     a[5] = a5 * (a12 * a19 * a23 - a12 * a18 * a24 + a22 * a14 * a18 - a22 * a13 * a19 + a17 * a13 *
a24 - a17 * a14 * a23);
00257     a[5] += a20 * (a12 * a9 * a18 - a12 * a8 * a19 + a7 * a13 * a19 - a18 * a7 * a14 + a17 * a8 * a14
- a9 * a17 * a13);
00258     a[5] += a15 * (a22 * a9 * a13 - a12 * a9 * a23 + a12 * a24 * a8 + a7 * a14 * a23 - a24 * a7 * a13
- a22 * a14 * a8);
00259     a[5] += a10 * (a18 * a7 * a24 - a18 * a22 * a9 - a17 * a8 * a24 + a17 * a9 * a23 + a22 * a8 * a19
- a19 * a23 * a7);
00260     a[5] *= det_inv;
00261
00262     a[6] = a2 * (a19 * a23 * a10 - a14 * a23 * a15 - a18 * a24 * a10 + a18 * a14 * a20 - a13 * a19 *
a20 + a24 * a13 * a15);
00263     a[6] += a12 * (a18 * a0 * a24 - a18 * a20 * a4 + a3 * a19 * a20 - a19 * a23 * a0 + a4 * a23 * a15
- a24 * a15 * a3);
00264     a[6] += a17 * (a4 * a13 * a20 - a13 * a24 * a0 + a14 * a23 * a0 - a3 * a14 * a20 + a24 * a3 * a10
- a4 * a23 * a10);
00265     a[6] += a22 * (a14 * a15 * a3 - a18 * a14 * a0 + a18 * a4 * a10 - a4 * a13 * a15 + a13 * a19 * a0
- a3 * a19 * a10);
00266     a[6] *= det_inv;
00267
00268     a[7] = a0 * (a18 * a9 * a22 - a18 * a24 * a7 + a19 * a23 * a7 - a9 * a23 * a17 + a24 * a8 * a17 -
a8 * a19 * a22);
00269     a[7] += a5 * (a2 * a18 * a24 - a2 * a19 * a23 + a17 * a4 * a23 - a17 * a3 * a24 + a22 * a3 * a19 -
a22 * a4 * a18);
00270     a[7] += a15 * (a4 * a8 * a22 - a3 * a9 * a22 - a24 * a8 * a2 + a9 * a23 * a2 - a4 * a23 * a7 + a24
* a3 * a7);
00271     a[7] += a20 * (a18 * a4 * a7 - a18 * a9 * a2 + a9 * a3 * a17 - a4 * a8 * a17 + a8 * a19 * a2 - a3
* a19 * a7);
00272     a[7] *= det_inv;
00273
00274     a[8] = a0 * (a12 * a9 * a23 - a12 * a24 * a8 + a22 * a14 * a8 - a7 * a14 * a23 + a24 * a7 * a13 -
a9 * a22 * a13);
00275     a[8] += a5 * (a12 * a3 * a24 - a12 * a4 * a23 - a22 * a3 * a14 + a2 * a14 * a23 - a2 * a13 * a24 +
a22 * a4 * a13);
00276     a[8] += a10 * (a22 * a9 * a3 - a4 * a22 * a8 + a4 * a7 * a23 - a2 * a9 * a23 + a24 * a2 * a8 - a7
* a24 * a3);
00277     a[8] += a20 * (a7 * a14 * a3 - a4 * a7 * a13 + a9 * a2 * a13 + a12 * a4 * a8 - a12 * a9 * a3 - a2
* a14 * a8);
00278     a[8] *= det_inv;
00279
00280     a[9] = a0 * (a12 * a8 * a19 - a12 * a18 * a9 + a18 * a7 * a14 - a8 * a17 * a14 + a17 * a13 * a9 -
a7 * a13 * a19);
00281     a[9] += a5 * (a2 * a13 * a19 - a2 * a14 * a18 - a12 * a3 * a19 + a12 * a4 * a18 + a17 * a3 * a14 -
a17 * a4 * a13);
00282     a[9] += a10 * (a18 * a2 * a9 - a18 * a7 * a4 + a3 * a7 * a19 - a2 * a8 * a19 + a17 * a8 * a4 - a3
* a17 * a9);
00283     a[9] += a15 * (a8 * a2 * a14 - a12 * a8 * a4 + a12 * a3 * a9 - a3 * a7 * a14 + a7 * a13 * a4 - a2
* a13 * a9);
00284     a[9] *= det_inv;
00285
00286     a[10] = a5 * (a18 * a24 * a11 - a24 * a13 * a16 + a14 * a23 * a16 - a19 * a23 * a11 + a13 * a19 *
a21 - a18 * a14 * a21);
00287     a[10] += a10 * (a19 * a23 * a6 - a9 * a23 * a16 + a24 * a8 * a16 - a8 * a19 * a21 + a18 * a9 * a21
- a18 * a24 * a6);
00288     a[10] += a15 * (a24 * a13 * a6 - a14 * a23 * a6 - a24 * a8 * a11 + a9 * a23 * a11 + a14 * a8 * a21
- a13 * a9 * a21);
00289     a[10] += a20 * (a18 * a14 * a6 - a18 * a9 * a11 + a8 * a19 * a11 - a13 * a19 * a6 + a9 * a13 * a16
- a14 * a8 * a16);
00290     a[10] *= det_inv;
00291
00292     a[11] = a4 * (a21 * a13 * a15 - a11 * a23 * a15 + a16 * a23 * a10 - a13 * a16 * a20 + a18 * a11 *
a20 - a18 * a21 * a10);
00293     a[11] += a14 * (a18 * a0 * a21 - a1 * a18 * a20 + a16 * a3 * a20 - a23 * a0 * a16 + a1 * a23 * a15
- a21 * a3 * a15);
00294     a[11] += a19 * (a1 * a13 * a20 - a1 * a23 * a10 + a23 * a0 * a11 + a21 * a3 * a10 - a11 * a3 * a20
- a13 * a0 * a21);
00295     a[11] += a24 * (a13 * a0 * a16 - a18 * a0 * a11 + a11 * a3 * a15 + a1 * a18 * a10 - a1 * a13 * a15
- a16 * a3 * a10);
00296     a[11] *= det_inv;
00297
00298     a[12] = a4 * (a5 * a21 * a18 - a18 * a20 * a6 + a20 * a16 * a8 - a5 * a16 * a23 + a15 * a6 * a23 -
a21 * a15 * a8);
00299     a[12] += a9 * (a1 * a20 * a18 - a1 * a15 * a23 + a0 * a16 * a23 - a18 * a0 * a21 - a20 * a16 * a3
+ a15 * a21 * a3);
00300     a[12] += a19 * (a20 * a6 * a3 - a5 * a21 * a3 + a0 * a21 * a8 - a23 * a0 * a6 + a1 * a5 * a23 - a1
* a20 * a8);
00301     a[12] += a24 * (a1 * a15 * a8 - a0 * a16 * a8 + a18 * a0 * a6 - a1 * a5 * a18 + a5 * a16 * a3 - a6

```

```

    * a15 * a3);
00302     a[12] = det_inv;
00303
00304     a[13] = a0 * (a24 * a11 * a8 - a6 * a24 * a13 + a21 * a9 * a13 - a11 * a9 * a23 + a14 * a6 * a23 -
a14 * a21 * a8);
00305     a[13] += a1 * (a5 * a13 * a24 - a5 * a14 * a23 + a14 * a20 * a8 + a10 * a9 * a23 - a24 * a10 * a8
- a20 * a9 * a13);
00306     a[13] += a3 * (a6 * a10 * a24 - a10 * a9 * a21 + a5 * a14 * a21 - a5 * a24 * a11 + a20 * a9 * a11
- a14 * a6 * a20);
00307     a[13] += a4 * (a5 * a11 * a23 - a5 * a21 * a13 + a21 * a10 * a8 - a6 * a10 * a23 + a20 * a6 * a13
- a11 * a20 * a8);
00308     a[13] = det_inv;
00309
00310     a[14] = a0 * (a13 * a19 * a6 - a14 * a18 * a6 - a11 * a19 * a8 + a14 * a16 * a8 + a11 * a18 * a9 -
a13 * a16 * a9);
00311     a[14] += a1 * (a14 * a18 * a5 - a13 * a19 * a5 + a10 * a19 * a8 - a14 * a15 * a8 - a10 * a18 * a9
+ a13 * a15 * a9);
00312     a[14] += a3 * (a11 * a19 * a5 - a11 * a15 * a9 + a10 * a16 * a9 - a10 * a19 * a6 + a14 * a15 * a6
- a14 * a16 * a5);
00313     a[14] += a4 * (a11 * a15 * a8 - a11 * a18 * a5 + a13 * a16 * a5 - a13 * a15 * a6 + a10 * a18 * a6
- a10 * a16 * a8);
00314     a[14] = det_inv;
00315
00316     a[15] = a5 * (a19 * a22 * a11 - a24 * a17 * a11 + a12 * a24 * a16 - a22 * a14 * a16 - a12 * a19 *
a21 + a17 * a14 * a21);
00317     a[15] += a10 * (a24 * a17 * a6 - a19 * a22 * a6 - a24 * a7 * a16 + a22 * a9 * a16 + a19 * a7 * a21
- a17 * a9 * a21);
00318     a[15] += a15 * (a22 * a14 * a6 - a9 * a22 * a11 + a24 * a7 * a11 - a12 * a24 * a6 - a7 * a14 * a21
+ a12 * a9 * a21);
00319     a[15] += a20 * (a12 * a19 * a6 - a17 * a14 * a6 - a19 * a7 * a11 + a9 * a17 * a11 + a7 * a14 * a16
- a12 * a9 * a16);
00320     a[15] = det_inv;
00321
00322     a[16] = a0 * (a11 * a17 * a24 - a11 * a19 * a22 - a12 * a16 * a24 + a12 * a19 * a21 + a14 * a16 *
a22 - a14 * a17 * a21);
00323     a[16] += a1 * (a10 * a19 * a22 - a10 * a17 * a24 + a12 * a15 * a24 - a12 * a19 * a20 - a14 * a15 *
a22 + a14 * a17 * a20);
00324     a[16] += a2 * (a10 * a16 * a24 - a10 * a19 * a21 - a11 * a15 * a24 + a11 * a19 * a20 + a14 * a15 *
a21 - a14 * a16 * a20);
00325     a[16] += a4 * (a10 * a17 * a21 + a11 * a15 * a22 - a11 * a17 * a20 - a12 * a15 * a21 + a12 * a16
* a20 - a10 * a16 * a22);
00326     a[16] = det_inv;
00327
00328     a[17] = a0 * (a21 * a9 * a17 - a6 * a24 * a17 + a19 * a6 * a22 - a0 * a16 * a9 * a22 + a24 * a16 *
a7 - a19 * a21 * a7);
00329     a[17] += a1 * (a5 * a24 * a17 - a5 * a19 * a22 + a19 * a20 * a7 - a20 * a9 * a17 + a15 * a9 * a22
- a24 * a15 * a7);
00330     a[17] += a2 * (a5 * a19 * a21 - a19 * a6 * a20 - a5 * a24 * a16 + a24 * a6 * a15 - a15 * a9 * a21
+ a20 * a9 * a16);
00331     a[17] += a4 * (a16 * a5 * a22 - a6 * a15 * a22 + a20 * a6 * a17 - a5 * a21 * a17 - a6 * a15 * a22
+ a21 * a15 * a7 - a16 * a20 * a7);
00332     a[17] = det_inv;
00333
00334     a[18] = a0 * (a12 * a24 * a6 - a14 * a22 * a6 - a11 * a24 * a7 + a14 * a21 * a7 + a11 * a22 * a9 -
a12 * a21 * a9);
00335     a[18] += a1 * (a14 * a22 * a5 - a12 * a24 * a5 + a10 * a24 * a7 - a14 * a20 * a7 - a10 * a22 * a9
+ a12 * a20 * a9);
00336     a[18] += a2 * (a11 * a24 * a5 - a11 * a20 * a9 + a14 * a20 * a6 - a14 * a21 * a5 + a10 * a21 * a9
- a10 * a24 * a6);
00337     a[18] += a4 * (a11 * a20 * a7 - a11 * a22 * a5 + a12 * a21 * a5 + a10 * a22 * a6 - a12 * a20 * a6
- a10 * a21 * a7);
00338     a[18] = det_inv;
00339
00340     a[19] = a0 * (a12 * a16 * a9 - a6 * a12 * a19 + a6 * a17 * a14 - a17 * a11 * a9 + a11 * a7 * a19 -
a16 * a7 * a14);
00341     a[19] += a1 * (a5 * a12 * a19 - a5 * a17 * a14 - a12 * a15 * a9 + a17 * a10 * a9 + a15 * a7 * a14
- a10 * a7 * a19);
00342     a[19] += a2 * (a11 * a15 * a9 - a5 * a11 * a19 + a5 * a16 * a14 - a6 * a15 * a14 + a6 * a10 * a19
- a16 * a10 * a9);
00343     a[19] += a4 * (a5 * a17 * a11 - a5 * a12 * a16 + a12 * a6 * a15 + a10 * a7 * a16 - a17 * a6 * a10
- a15 * a7 * a11);
00344     a[19] = det_inv;
00345
00346     a[20] = a5 * (a12 * a18 * a21 - a12 * a23 * a16 + a22 * a13 * a16 - a18 * a22 * a11 + a23 * a17 *
a11 - a17 * a13 * a21);
00347     a[20] += a15 * (a12 * a23 * a6 - a12 * a8 * a21 + a8 * a22 * a11 - a23 * a7 * a11 + a7 * a13 * a21
- a22 * a13 * a6);
00348     a[20] += a20 * (a12 * a8 * a16 - a12 * a18 * a6 + a18 * a7 * a11 - a8 * a17 * a11 + a17 * a13 * a6
- a7 * a13 * a16);
00349     a[20] += a10 * (a17 * a8 * a21 - a22 * a8 * a16 - a18 * a7 * a21 + a18 * a22 * a6 + a23 * a7 * a16
- a23 * a17 * a6);

```

```

00350     a[20] *= det_inv;
00351
00352     a[21] = a0 * (a12 * a23 * a16 - a12 * a18 * a21 + a17 * a13 * a21 + a18 * a22 * a11 - a23 * a17 *
a11 - a22 * a13 * a16);
00353     a[21] += a1 * (a12 * a18 * a20 - a12 * a23 * a15 + a22 * a13 * a15 + a23 * a17 * a10 - a17 * a13 *
a20 - a18 * a22 * a10);
00354     a[21] += a2 * (a18 * a21 * a10 - a18 * a11 * a20 - a21 * a13 * a15 + a16 * a13 * a20 - a23 * a16 *
a10 + a23 * a11 * a15);
00355     a[21] += a3 * (a17 * a11 * a20 - a12 * a16 * a20 + a12 * a21 * a15 - a21 * a17 * a10 - a22 * a11 *
a15 + a16 * a22 * a10);
00356     a[21] *= det_inv;
00357
00358     a[22] = a0 * (a18 * a21 * a7 - a18 * a6 * a22 + a23 * a6 * a17 + a16 * a8 * a22 - a21 * a8 * a17 -
a23 * a16 * a7);
00359     a[22] += a1 * (a5 * a18 * a22 - a5 * a23 * a17 - a15 * a8 * a22 + a20 * a8 * a17 - a18 * a20 * a7
+ a23 * a15 * a7);
00360     a[22] += a3 * (a16 * a20 * a7 + a6 * a15 * a22 - a6 * a20 * a17 - a5 * a16 * a22 + a5 * a21 * a17
- a21 * a15 * a7);
00361     a[22] += a2 * (a5 * a23 * a16 - a5 * a18 * a21 + a18 * a6 * a20 + a15 * a8 * a21 - a20 * a8 * a16
- a23 * a6 * a15);
00362     a[22] *= det_inv;
00363
00364     a[23] = a0 * (a12 * a21 * a8 - a22 * a11 * a8 + a11 * a7 * a23 - a6 * a12 * a23 - a21 * a7 * a13 +
a6 * a22 * a13);
00365     a[23] += a1 * (a5 * a12 * a23 - a5 * a22 * a13 - a10 * a7 * a23 + a20 * a7 * a13 + a22 * a10 * a8
- a12 * a20 * a8);
00366     a[23] += a2 * (a5 * a21 * a13 + a11 * a20 * a8 + a6 * a10 * a23 - a5 * a11 * a23 - a21 * a10 * a8
- a6 * a20 * a13);
00367     a[23] += a3 * (a5 * a22 * a11 - a5 * a12 * a21 + a10 * a7 * a21 - a22 * a6 * a10 - a20 * a7 * a11
+ a12 * a6 * a20);
00368     a[23] *= det_inv;
00369
00370     a[24] = a0 * (a17 * a11 * a8 - a11 * a7 * a18 + a6 * a12 * a18 - a12 * a16 * a8 + a16 * a7 * a13 -
a6 * a17 * a13);
00371     a[24] += a1 * (a5 * a17 * a13 - a5 * a12 * a18 + a10 * a7 * a18 + a12 * a15 * a8 - a17 * a10 * a8
- a15 * a7 * a13);
00372     a[24] += a2 * (a5 * a11 * a18 - a5 * a16 * a13 + a16 * a10 * a8 + a6 * a15 * a13 - a11 * a15 * a8
- a6 * a10 * a18);
00373     a[24] += a3 * (a5 * a12 * a16 + a17 * a6 * a10 - a5 * a17 * a11 - a12 * a6 * a15 - a10 * a7 * a16
+ a15 * a7 * a11);
00374     a[24] *= det_inv;
00375
00376     }
00377
00378     printf("### DEBUG: Check inverse matrix...\n");
00379     printf("###-----\n");
00380     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
00381     a0 * a[0] + a1 * a[5] + a2 * a[10] + a3 * a[15] + a4 * a[20],
00382     a0 * a[1] + a1 * a[6] + a2 * a[11] + a3 * a[16] + a4 * a[21],
00383     a0 * a[2] + a1 * a[7] + a2 * a[12] + a3 * a[17] + a4 * a[22],
00384     a0 * a[3] + a1 * a[8] + a2 * a[13] + a3 * a[18] + a4 * a[23],
00385     a0 * a[4] + a1 * a[9] + a2 * a[14] + a3 * a[19] + a4 * a[24]);
00386     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
00387     a5 * a[0] + a6 * a[5] + a7 * a[10] + a8 * a[15] + a9 * a[20],
00388     a5 * a[1] + a6 * a[6] + a7 * a[11] + a8 * a[16] + a9 * a[21],
00389     a5 * a[2] + a6 * a[7] + a7 * a[12] + a8 * a[17] + a9 * a[22],
00390     a5 * a[3] + a6 * a[8] + a7 * a[13] + a8 * a[18] + a9 * a[23],
00391     a5 * a[4] + a6 * a[9] + a7 * a[14] + a8 * a[19] + a9 * a[24]);
00392     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
00393     a10 * a[0] + a11 * a[5] + a12 * a[10] + a13 * a[15] + a14 * a[20],
00394     a10 * a[1] + a11 * a[6] + a12 * a[11] + a13 * a[16] + a14 * a[21],
00395     a10 * a[2] + a11 * a[7] + a12 * a[12] + a13 * a[17] + a14 * a[22],
00396     a10 * a[3] + a11 * a[8] + a12 * a[13] + a13 * a[18] + a14 * a[23],
00397     a10 * a[4] + a11 * a[9] + a12 * a[14] + a13 * a[19] + a14 * a[24]);
00398     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
00399     a15 * a[0] + a16 * a[5] + a17 * a[10] + a18 * a[15] + a19 * a[20],
00400     a15 * a[1] + a16 * a[6] + a17 * a[11] + a18 * a[16] + a19 * a[21],
00401     a15 * a[2] + a16 * a[7] + a17 * a[12] + a18 * a[17] + a19 * a[22],
00402     a15 * a[3] + a16 * a[8] + a17 * a[13] + a18 * a[18] + a19 * a[23],
00403     a15 * a[4] + a16 * a[9] + a17 * a[14] + a18 * a[19] + a19 * a[24]);
00404     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
00405     a20 * a[0] + a21 * a[5] + a22 * a[10] + a23 * a[15] + a24 * a[20],
00406     a20 * a[1] + a21 * a[6] + a22 * a[11] + a23 * a[16] + a24 * a[21],
00407     a20 * a[2] + a21 * a[7] + a22 * a[12] + a23 * a[17] + a24 * a[22],
00408     a20 * a[3] + a21 * a[8] + a22 * a[13] + a23 * a[18] + a24 * a[23],
00409     a20 * a[4] + a21 * a[9] + a22 * a[14] + a23 * a[19] + a24 * a[24]);
00410     printf("###-----\n");
00411 }
00412
00425 void fasp_smat_inv_nc7 (REAL *a)
00426 {

```

```

00427     fasp_smat_invp_nc(a,7);
00428 }
00429
00441 void fasp_smat_inv_nc (REAL      *a,
00442                       const INT  n)
00443 {
00444     INT i,j,k,l,u,kn,in;
00445     REAL alinv;
00446
00447     for (k=0; k<n; ++k) {
00448
00449         kn = k*n;
00450         l  = kn+k;
00451
00452         if (ABS(a[l]) < SMALLREAL) {
00453             printf("### ERROR: Diagonal entry is close to zero! ");
00454             printf("diag_%d = %.2e! [%s]\n", k, a[l], __FUNCTION__);
00455             exit(ERROR_SOLVER_EXIT);
00456         }
00457         alinv = 1.0/a[l];
00458         a[l] = alinv;
00459
00460         for (j=0; j<k; ++j) {
00461             u = kn+j; a[u] *= alinv;
00462         }
00463
00464         for (j=k+1; j<n; ++j) {
00465             u = kn+j; a[u] *= alinv;
00466         }
00467
00468         for (i=0; i<k; ++i) {
00469             in = i*n;
00470             for (j=0; j<n; ++j)
00471                 if (j!=k) {
00472                     u = in+j; a[u] -= a[in+k]*a[kn+j];
00473                 } // end if (j!=k)
00474         }
00475
00476         for (i=k+1; i<n; ++i) {
00477             in = i*n;
00478             for (j=0; j<n; ++j)
00479                 if (j!=k) {
00480                     u = in+j; a[u] -= a[in+k]*a[kn+j];
00481                 } // end if (j!=k)
00482         }
00483
00484         for (i=0; i<k; ++i) {
00485             u=i*n+k; a[u] *= -alinv;
00486         }
00487
00488         for (i=k+1; i<n; ++i) {
00489             u=i*n+k; a[u] *= -alinv;
00490         }
00491     } // end for (k=0; k<n; ++k)
00492 }
00493
00494
00508 SHORT fasp_smat_invp_nc (REAL      *a,
00509                          const INT  n)
00510 {
00511     INT i, j, k, l, ll, u;
00512     INT icol = 0, irow = 0;
00513     REAL vmax, dum, pivinv, temp;
00514
00515     INT *work = (INT *)fasp_mem_calloc(3*n,sizeof(INT));
00516     INT *indxc = work, *indxr = work+n, *ipiv = work+2*n;
00517
00518     // ipiv, indxr, and indxc are used for book-keeping on the pivoting.
00519     for ( j=0; j<n; j++ ) ipiv[j] = 0;
00520
00521     #if DEBUG_MODE > 1
00522         printf("### DEBUG: Matrix block\n");
00523         for ( i = 0; i < n; ++i ) {
00524             for ( j = 0; j < n; ++j ) {
00525                 printf(" %10.5e", a[i * n + j]);
00526             }
00527             printf("\n");
00528         }
00529     #endif
00530
00531     // This is the main loop over the columns to be reduced.

```



```

00532     for ( i=0; i<n; i++ ) {
00533
00534         // This is the outer loop of the search for a pivot element.
00535         vmax = 0.0;
00536         for ( j=0; j<n; j++ ) {
00537             if ( ipiv[j] != 1 ) {
00538                 for ( k=0; k<n; k++ ) {
00539                     if ( ipiv[k] == 0 ) {
00540                         u = j*n+k;
00541                         if ( ABS(a[u]) >= vmax ) {
00542                             vmax = ABS(a[u]); irow = j; icol = k;
00543                         }
00544                     }
00545                 } // end for k
00546             }
00547         } // end for j
00548
00549         ++(ipiv[icol]);
00550
00551         // We now have the pivot element, so we interchange rows, if needed, to put
00552         // the pivot element on the diagonal. The columns are not physically
00553         // interchanged, only relabeled: indx[i], the column of the ith pivot
00554         // element, is the ith column that is reduced, while indxr[i] is the row in
00555         // which that pivot element was originally located. If indxr[i] != indx[i]
00556         // there is an implied column interchange. With this form of bookkeeping,
00557         // the inverse matrix will be scrambled by columns.
00558         if ( irow != icol ) {
00559             for ( l=0; l<n; l++ ) SWAP(a[irow*n+l],a[icol*n+l]);
00560         }
00561
00562         indxr[i] = irow; indx[i] = icol;
00563         u = icol*n+icol;
00564         if ( ABS(a[u]) < SMALLREAL ) {
00565             printf("### WARNING: The matrix is nearly singular!\n");
00566             return ERROR_SOLVER_EXIT;
00567         }
00568         pivinv = 1.0/a[u]; a[u]=1.0;
00569         for ( l=0; l<n; l++ ) a[icol*n+l] *= pivinv;
00570
00571         for ( ll=0; ll<n; ll++ ) {
00572             if ( ll != icol ) {
00573                 u = ll*n+icol;
00574                 dum = a[u]; a[u] = 0.0;
00575                 for ( l=0; l<n; l++ ) a[ll*n+l] -= a[icol*n+l]*dum;
00576             }
00577         }
00578     }
00579     // This is the end of the main loop over columns of the reduction.
00580
00581     // It only remains to unscramble the matrix in view of the column interchanges.
00582     for ( l=n-1; l>=0; l-- ) {
00583         if ( indxr[l] != indx[l] )
00584             for ( k=0; k<n; k++ ) SWAP(a[k*n+indxr[l]],a[k*n+indx[l]]);
00585     } // And we are done.
00586
00587     fasp_mem_free(work); work = NULL;
00588
00589     return FASP_SUCCESS;
00590 }
00591
00603 SHORT fasp_smat_inv (REAL      *a,
00604                      const INT  n)
00605 {
00606     SHORT status = FASP_SUCCESS;
00607
00608     switch (n) {
00609
00610         case 2:
00611             fasp_smat_inv_nc2(a);
00612             break;
00613
00614         case 3:
00615             fasp_smat_inv_nc3(a);
00616             break;
00617
00618         case 4:
00619             fasp_smat_inv_nc4(a);
00620             break;
00621
00622         case -5:
00623             fasp_smat_inv_nc5(a);

```

```

00624         break;
00625
00626     default:
00627         status = fasp_smat_invp_nc(a,n);
00628         break;
00629
00630     }
00631
00632     return status;
00633 }
00634
00646 REAL fasp_smat_Linf (const REAL *A,
00647                     const INT n)
00648 {
00649
00650     REAL norm = 0.0, value;
00651
00652     INT i,j;
00653
00654     for ( i = 0; i < n; i++ ) {
00655         for ( value = 0.0, j = 0; j < n; j++ ) {
00656             value = value + ABS(A[i*n+j]);
00657         }
00658         norm = MAX(norm, value);
00659     }
00660
00661     return norm;
00662 }
00663
00674 void fasp_smat_identity_nc2 (REAL *a)
00675 {
00676     memset(a, 0X0, 4*sizeof(REAL));
00677
00678     a[0] = 1.0; a[3] = 1.0;
00679 }
00680
00691 void fasp_smat_identity_nc3 (REAL *a)
00692 {
00693     memset(a, 0X0, 9*sizeof(REAL));
00694
00695     a[0] = 1.0; a[4] = 1.0; a[8] = 1.0;
00696 }
00697
00708 void fasp_smat_identity_nc5 (REAL *a)
00709 {
00710     memset(a, 0X0, 25*sizeof(REAL));
00711
00712     a[0] = 1.0;
00713     a[6] = 1.0;
00714     a[12] = 1.0;
00715     a[18] = 1.0;
00716     a[24] = 1.0;
00717 }
00718
00729 void fasp_smat_identity_nc7 (REAL *a)
00730 {
00731     memset(a, 0X0, 49*sizeof(REAL));
00732
00733     a[0] = 1.0;
00734     a[8] = 1.0;
00735     a[16] = 1.0;
00736     a[24] = 1.0;
00737     a[32] = 1.0;
00738     a[40] = 1.0;
00739     a[48] = 1.0;
00740 }
00741
00754 void fasp_smat_identity (REAL *a,
00755                         const INT n,
00756                         const INT n2)
00757 {
00758     memset(a, 0X0, n2*sizeof(REAL));
00759
00760     switch (n) {
00761
00762     case 2: {
00763         a[0] = 1.0;
00764         a[3] = 1.0;
00765     }
00766
00767         break;

```

```

00768         case 3: {
00769             a[0] = 1.0;
00770             a[4] = 1.0;
00771             a[8] = 1.0;
00772         }
00773         break;
00774
00775         case 4: {
00776             a[0] = 1.0;
00777             a[5] = 1.0;
00778             a[10] = 1.0;
00779             a[15] = 1.0;
00780         }
00781         break;
00782
00783         case 5: {
00784             a[0] = 1.0;
00785             a[6] = 1.0;
00786             a[12] = 1.0;
00787             a[18] = 1.0;
00788             a[24] = 1.0;
00789         }
00790         break;
00791
00792         case 6: {
00793             a[0] = 1.0;
00794             a[7] = 1.0;
00795             a[14] = 1.0;
00796             a[21] = 1.0;
00797             a[28] = 1.0;
00798             a[35] = 1.0;
00799         }
00800         break;
00801
00802         case 7: {
00803             a[0] = 1.0;
00804             a[8] = 1.0;
00805             a[16] = 1.0;
00806             a[24] = 1.0;
00807             a[32] = 1.0;
00808             a[40] = 1.0;
00809             a[48] = 1.0;
00810         }
00811         break;
00812
00813         default: {
00814             INT l;
00815             for (l = 0; l < n; l++) a[l*n+1] = 1.0;
00816         }
00817         break;
00818     }
00819
00820 }
00821
00822 /*-----*/
00823 /*--      End of File      --*/
00824 /*-----*/

```

9.69 BlaSmallMatLU.c File Reference

LU decomposition and direct solver for small dense matrices.

```

#include <math.h>
#include "fasp.h"

```

Functions

- [SHORT fasp_smat_lu_decomp](#) (REAL *A, INT pivot[], const INT n)
LU decomposition of A using Doolittle's method.
- [SHORT fasp_smat_lu_solve](#) (const REAL *A, REAL b[], const INT pivot[], REAL x[], const INT n)
Solving $Ax=b$ using LU decomposition.

9.69.1 Detailed Description

LU decomposition and direct solver for small dense matrices.

Note

This file contains Level-1 (Bla) functions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSmallMatLU.c](#).

9.69.2 Function Documentation

9.69.2.1 fasp_smat_lu_decomp()

```
SHORT fasp_smat_lu_decomp (
    REAL * A,
    INT pivot[],
    const INT n )
```

LU decomposition of A using Doolittle's method.

Parameters

<i>A</i>	Pointer to the full matrix
<i>pivot</i>	Pivoting positions
<i>n</i>	Size of matrix A

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Note

Use Doolittle's method to decompose the $n \times n$ matrix A into a unit lower triangular matrix L and an upper triangular matrix U such that $A = LU$. The matrices L and U replace the matrix A. The diagonal elements of L are 1 and are not stored.

The Doolittle method with partial pivoting is: Determine the pivot row and interchange the current row with the pivot row, then assuming that row k is the current row, $k = 0, \dots, n - 1$ evaluate in order the following pair of expressions $U[k][j] = A[k][j] - (L[k][0]*U[0][j] + \dots + L[k][k-1]*U[k-1][j])$ for $j = k, k+1, \dots, n-1$ $L[i][k] = (A[i][k] - (L[i][0]*U[0][k] + \dots + L[i][k-1]*U[k-1][k])) / U[k][k]$ for $i = k+1, \dots, n-1$.

Author

Xuehai Huang

Date

04/02/2009

Definition at line 52 of file [BlaSmallMatLU.c](#).

9.69.2.2 fasp_smat_lu_solve()

```
SHORT fasp_smat_lu_solve (
    const REAL * A,
    REAL b[],
    const INT pivot[],
    REAL x[],
    const INT n )
```

Solving $Ax=b$ using LU decomposition.

Parameters

<i>A</i>	Pointer to the full matrix
<i>b</i>	Right hand side array (b is used as the working array!!!)
<i>pivot</i>	Pivoting positions
<i>x</i>	Pointer to the solution array
<i>n</i>	Size of matrix A

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Note

This routine uses Doolittle's method to solve the linear equation $Ax = b$. This routine is called after the matrix A has been decomposed into a product of a unit lower triangular matrix L and an upper triangular matrix U with pivoting. The solution proceeds by solving the linear equation $Ly = b$ for y and subsequently solving the linear equation $Ux = y$ for x.

Author

Xuehai Huang

Date

04/02/2009

Definition at line 124 of file BlaSmallMatLU.c.

9.70 BlaSmallMatLU.c

[Go to the documentation of this file.](#)

```
00001
00013 #include <math.h>
00014
00015 #include "fasp.h"
00016
00017 /*-----*/
00018 /*--      Public Functions      --*/
00019 /*-----*/
00020
00052 SHORT fasp_smat_lu_decomp (REAL      *A,
00053                          INT        pivot[],
00054                          const INT  n)
00055 {
00056     INT i, j, k;
00057     REAL *p_k=NULL, *p_row=NULL, *p_col=NULL;
00058     REAL max;
00059
00060     /* For each row and column, k = 0, ..., n-1, */
```

```

00061     for (k = 0, p_k = A; k < n; p_k += n, k++) {
00062
00063         // find the pivot row
00064         pivot[k] = k;
00065         max = fabs( *(p_k + k) );
00066         for (j = k + 1, p_row = p_k + n; j < n; ++j, p_row += n) {
00067             if ( max < fabs(*(p_row + k)) ) {
00068                 max = fabs(*(p_row + k));
00069                 pivot[k] = j;
00070                 p_col = p_row;
00071             }
00072         }
00073
00074         // if the pivot row differs from the current row, interchange the two rows.
00075         if (pivot[k] != k)
00076             for (j = 0; j < n; ++j) {
00077                 max = *(p_k + j);
00078                 *(p_k + j) = *(p_col + j);
00079                 *(p_col + j) = max;
00080             }
00081
00082         // if the matrix is singular, return error
00083         if ( fabs( *(p_k + k) ) < SMALLREAL ) return -1;
00084
00085         // otherwise find the lower triangular matrix elements for column k.
00086         for (i = k+1, p_row = p_k + n; i < n; p_row += n, ++i) {
00087             *(p_row + k) /= *(p_k + k);
00088         }
00089
00090         // update remaining matrix
00091         for (i = k+1, p_row = p_k + n; i < n; p_row += n, ++i)
00092             for (j = k+1; j < n; ++j)
00093                 *(p_row + j) -= *(p_row + k) * *(p_k + j);
00094     }
00095 }
00096
00097 return FASP_SUCCESS;
00098 }
00099
00124 SHORT fasp_smat_lu_solve (const REAL *A,
00125                          REAL b[],
00126                          const INT pivot[],
00127                          REAL x[],
00128                          const INT n)
00129 {
00130     INT i, k;
00131     REAL dum;
00132     const REAL *p_k;
00133
00134     /* solve Ly = b */
00135     for (k = 0, p_k = A; k < n; p_k += n, k++) {
00136         if (pivot[k] != k) {dum = b[k]; b[k] = b[pivot[k]]; b[pivot[k]] = dum; }
00137         x[k] = b[k];
00138         for (i = 0; i < k; ++i) x[k] -= x[i] * *(p_k + i);
00139     }
00140
00141     /* solve Ux = y */
00142     for (k = n-1, p_k = A + n*(n-1); k >= 0; k--, p_k -= n) {
00143         if (pivot[k] != k) {dum = b[k]; b[k] = b[pivot[k]]; b[pivot[k]] = dum; }
00144         for (i = k + 1; i < n; ++i) x[k] -= x[i] * *(p_k + i);
00145         if (*(p_k + k) == 0.0) return -1;
00146         x[k] /= *(p_k + k);
00147     }
00148
00149     return FASP_SUCCESS;
00150 }
00151
00152 /*-----*/
00153 /*--          End of File          --*/
00154 /*-----*/
00155
00156 /*
00157
00158 //A simple test example can be written as the following
00159 INT main (INT argc, const char * argv[])
00160 {
00161     REAL A[3][3] = {{0.0, 1.0, 4.0},
00162 {4.0, 1.0, 0.0},
00163 {1.0, 4.0, 1.0}};
00164
00165     REAL b[3] = {1, 1, 1}, x[3];

```

```

00166
00167 INT pivot[3];
00168
00169 INT ret, i, j;
00170
00171 ret = lu_decomp(&A[0][0], pivot, 3); // LU decomposition
00172
00173 ret = lu_solve(&A[0][0], b, pivot, x, 3); // Solve decomposed Ax=b
00174
00175 return 1;
00176 }
00177
00178 */

```

9.71 BlaSparseBLC.c File Reference

Sparse matrix block operations.

```

#include <time.h>
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_dblc_free](#) (dBLCmat *A)
Free block CSR sparse matrix data memory space.

9.71.1 Detailed Description

Sparse matrix block operations.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#) and [BlaSparseCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseBLC.c](#).

9.71.2 Function Documentation

9.71.2.1 fasp_dblc_free()

```

void fasp_dblc_free (
    dBLCmat * A )

```

Free block CSR sparse matrix data memory space.

Parameters

<i>A</i>	Pointer to the dBLCmat matrix
----------	---

Author

Xiaozhe Hu

Date

04/18/2014

Definition at line 38 of file [BlaSparseBLC.c](#).

9.72 BlaSparseBLC.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <time.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_block.h"
00022 #include "fasp_functs.h"
00023
00024 /*-----*/
00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00038 void fasp_dblc_free (dBLCmat *A)
00039 {
00040     INT i;
00041     INT num_blocks = (A->brow)*(A->bcol);
00042
00043     if (A == NULL) return; // Nothing need to be freed!
00044
00045     for ( i=0; i<num_blocks; i++ ) {
00046         fasp_dcsr_free(A->blocks[i]);
00047         A->blocks[i] = NULL;
00048     }
00049
00050     fasp_mem_free(A->blocks); A->blocks = NULL;
00051 }
00052
00053 /*-----*/
00054 /*--      End of File      --*/
00055 /*-----*/

```

9.73 BlaSparseBSR.c File Reference

Sparse matrix operations for [dBSRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [dBSRmat fasp_dbsr_create](#) (const [INT](#) ROW, const [INT](#) COL, const [INT](#) NNZ, const [INT](#) nb, const [INT](#) storage_manner)

Create BSR sparse matrix data memory space.
- void [fasp_dbsr_alloc](#) (const [INT](#) ROW, const [INT](#) COL, const [INT](#) NNZ, const [INT](#) nb, const [INT](#) storage_manner, [dBSRmat](#) *A)

Allocate memory space for BSR format sparse matrix.
- void [fasp_dbsr_free](#) ([dBSRmat](#) *A)

- Free memory space for BSR format sparse matrix.*
- void [fasp_dbsr_cp](#) (const [dBSRmat](#) *A, [dBSRmat](#) *B)
copy a [dCSRmat](#) to a new one B=A
- [INT fasp_dbsr_trans](#) (const [dBSRmat](#) *A, [dBSRmat](#) *AT)
Find A^T from given [dBSRmat](#) matrix A.
- [SHORT fasp_dbsr_getblk](#) (const [dBSRmat](#) *A, const [INT](#) *Is, const [INT](#) *Js, const [INT](#) m, const [INT](#) n, [dBSRmat](#) *B)
Get a sub BSR matrix of A with specified rows and columns.
- [SHORT fasp_dbsr_diagpref](#) ([dBSRmat](#) *A)
Reorder the column and data arrays of a square BSR matrix, so that the first entry in each row is the diagonal one.
- [dvector fasp_dbsr_getdiainv](#) (const [dBSRmat](#) *A)
Get D^{-1} of matrix A.
- [dBSRmat fasp_dbsr_diainv](#) (const [dBSRmat](#) *A)
*Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.*
- [dBSRmat fasp_dbsr_diainv2](#) (const [dBSRmat](#) *A, [REAL](#) *diainv)
*Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.*
- [dBSRmat fasp_dbsr_diainv3](#) (const [dBSRmat](#) *A, [REAL](#) *diainv)
*Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.*
- [dBSRmat fasp_dbsr_diainv4](#) (const [dBSRmat](#) *A, [REAL](#) *diainv)
*Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.*
- void [fasp_dbsr_getdiag](#) ([INT](#) n, const [dBSRmat](#) *A, [REAL](#) *diag)
Abstract the diagonal blocks of a BSR matrix.
- [dBSRmat fasp_dbsr_diagLU](#) (const [dBSRmat](#) *A, [REAL](#) *DL, [REAL](#) *DU)
*Compute $B := DL * A * DU$. We decompose each diagonal block of A into LDU form and $DL = \text{diag}(L^{-1})$ and $DU = \text{diag}(U^{-1})$.*
- [dBSRmat fasp_dbsr_diagLU2](#) ([dBSRmat](#) *A, [REAL](#) *DL, [REAL](#) *DU)
*Compute $B := DL * A * DU$. We decompose each diagonal block of A into LDU form and $DL = \text{diag}(L^{-1})$ and $DU = \text{diag}(U^{-1})$.*
- [dBSRmat fasp_dbsr_perm](#) (const [dBSRmat](#) *A, const [INT](#) *P)
Apply permutation of A, i.e. $A_{\text{perm}} = PAP'$ by the orders given in P.
- [INT fasp_dbsr_merge_col](#) ([dBSRmat](#) *A)
Check and merge some same col index in one row.

9.73.1 Detailed Description

Sparse matrix operations for [dBSRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaSmallMat.c](#), and [BlaSmallMatInv.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseBSR.c](#).

9.73.2 Function Documentation

9.73.2.1 fasp_dbsr_alloc()

```
void fasp_dbsr_alloc (
    const INT ROW,
    const INT COL,
    const INT NNZ,
    const INT nb,
    const INT storage_manner,
    dBSRmat * A )
```

Allocate memory space for BSR format sparse matrix.

Parameters

<i>ROW</i>	Number of rows of block
<i>COL</i>	Number of columns of block
<i>NNZ</i>	Number of nonzero blocks
<i>nb</i>	Dimension of each block
<i>storage_manner</i>	Storage manner for each sub-block
<i>A</i>	Pointer to new dBSRmat matrix

Author

Xiaozhe Hu

Date

10/26/2010

Definition at line 99 of file [BlaSparseBSR.c](#).

9.73.2.2 fasp_dbsr_cp()

```
void fasp_dbsr_cp (
    const dBSRmat * A,
    dBSRmat * B )
```

copy a [dCSRmat](#) to a new one B=A

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>B</i>	Pointer to the dBSRmat matrix

Author

Xiaozhe Hu

Date

08/07/2011

Definition at line 172 of file [BlaSparseBSR.c](#).

9.73.2.3 fasp_dbsr_create()

```
dBSRmat fasp_dbsr_create (
    const INT ROW,
    const INT COL,
    const INT NNZ,
    const INT nb,
    const INT storage_manner )
```

Create BSR sparse matrix data memory space.

Parameters

<i>ROW</i>	Number of rows of block
<i>COL</i>	Number of columns of block
<i>NNZ</i>	Number of nonzero blocks
<i>nb</i>	Dimension of each block
<i>storage_manner</i>	Storage manner for each sub-block

Returns

A The new [dBSRmat](#) matrix

Author

Xiaozhe Hu

Date

10/26/2010

Definition at line 45 of file [BlaSparseBSR.c](#).

9.73.2.4 fasp_dbsr_diaginv()

```
dBSRmat fasp_dbsr_diaginv (
    const dBSRmat * A )
```

Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
----------	---

Author

Zhiyang Zhou

Date

2010/10/26

Note

Works for general nb (Xiaozhe)

Modified by Chunsheng Feng, Zheng Li on 08/25/2012 Modified by Chensong Zhang on 09/27/2017
Definition at line 591 of file [BlaSparseBSR.c](#).

9.73.2.5 fasp_dbsr_diaginv2()

```
dBSRmat fasp_dbsr_diaginv2 (
    const dBSRmat * A,
    REAL * diaginv )
```

Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>diaginv</i>	Pointer to the inverses of all the diagonal blocks

Author

Zhiyang Zhou

Date

2010/11/07

Note

Works for general nb (Xiaozhe)

Modified by Chunsheng Feng, Zheng Li on 08/25/2012
Definition at line 751 of file [BlaSparseBSR.c](#).

9.73.2.6 fasp_dbsr_diaginv3()

```
dBSRmat fasp_dbsr_diaginv3 (
    const dBSRmat * A,
    REAL * diaginv )
```

Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>diaginv</i>	Pointer to the inverses of all the diagonal blocks

Returns

BSR matrix after diagonal scaling

Author

Xiaozhe Hu

Date

12/25/2010

Note

Works for general nb (Xiaozhe)

Modified by Xiaozhe Hu on 05/26/2012

Definition at line 857 of file [BlaSparseBSR.c](#).**9.73.2.7 fasp_dbsr_diaginv4()**

```
dBSRmat fasp_dbsr_diaginv4 (
    const dBSRmat * A,
    REAL * diaginv )
```

Compute $B := D^{-1} * A$, where 'D' is the block diagonal part of A.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>diaginv</i>	Pointer to the inverses of all the diagonal blocks

Returns

BSR matrix after diagonal scaling

Note

Works for general nb (Xiaozhe)

A is pre-ordered that the first block of each row is the diagonal block!

Author

Xiaozhe Hu

Date

03/12/2011

Modified by Chunsheng Feng, Zheng Li on 08/26/2012

Definition at line 1260 of file [BlaSparseBSR.c](#).**9.73.2.8 fasp_dbsr_diagLU()**

```
dBSRmat fasp_dbsr_diagLU (
    const dBSRmat * A,
    REAL * DL,
    REAL * DU )
```

Compute $B := DL * A * DU$. We decompose each diagonal block of A into LDU form and $DL = \text{diag}(L^{-1})$ and $DU = \text{diag}(U^{-1})$.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>DL</i>	Pointer to the $\text{diag}(L^{-1})$
<i>DU</i>	Pointer to the $\text{diag}(U^{-1})$

Returns

BSR matrix after scaling

Author

Xiaozhe Hu

Date

04/02/2014

Definition at line [1593](#) of file [BlaSparseBSR.c](#).

9.73.2.9 fasp_dbsr_diagLU2()

```
dBSRmat fasp_dbsr_diagLU2 (
    dBSRmat * A,
    REAL * DL,
    REAL * DU )
```

Compute $B := DL * A * DU$. We decompose each diagonal block of A into LDU form and $DL = \text{diag}(L^{-1})$ and $DU = \text{diag}(U^{-1})$.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>DL</i>	Pointer to the $\text{diag}(L^{-1})$
<i>DU</i>	Pointer to the $\text{diag}(U^{-1})$

Returns

BSR matrix after scaling

Author

Zheng Li, Xiaozhe Hu

Date

06/17/2014

Definition at line [1822](#) of file [BlaSparseBSR.c](#).

9.73.2.10 fasp_dbsr_diagpref()

```
SHORT fasp_dbsr_diagpref (
    dBSRmat * A )
```

Reorder the column and data arrays of a square BSR matrix, so that the first entry in each row is the diagonal one.

Parameters

<i>A</i>	Pointer to the BSR matrix
----------	---------------------------

Author

Xiaozhe Hu

Date

03/10/2011

Author

Chunsheng Feng, Zheng Li

Date

09/02/2012

Note

Reordering is done in place.

Definition at line [385](#) of file [BlaSparseBSR.c](#).**9.73.2.11 fasp_dbsr_free()**

```
void fasp_dbsr_free (
    dBSRmat * A )
```

Free memory space for BSR format sparse matrix.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
----------	---

Author

Xiaozhe Hu

Date

10/26/2010

Definition at line [146](#) of file [BlaSparseBSR.c](#).**9.73.2.12 fasp_dbsr_getblk()**

```
SHORT fasp_dbsr_getblk (
    const dBSRmat * A,
    const INT * Is,
    const INT * Js,
    const INT m,
```



```
const INT n,
dBSRmat * B )
```

Get a sub BSR matrix of A with specified rows and columns.

Parameters

<i>A</i>	Pointer to dBSRmat BSR matrix
<i>B</i>	Pointer to dBSRmat BSR matrix
<i>Is</i>	Pointer to selected rows
<i>Js</i>	Pointer to selected columns
<i>m</i>	Number of selected rows
<i>n</i>	Number of selected columns

Returns

FASP_SUCCESS if succeeded, otherwise return error information.

Author

Shiquan Zhang, Xiaozhe Hu

Date

12/25/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line [287](#) of file [BlaSparseBSR.c](#).

9.73.2.13 fasp_dbsr_getdiag()

```
void fasp_dbsr_getdiag (
    INT n,
    const dBSRmat * A,
    REAL * diag )
```

Abstract the diagonal blocks of a BSR matrix.

Parameters

<i>n</i>	Number of blocks to get
<i>A</i>	Pointer to the ' dBSRmat ' type matrix
<i>diag</i>	Pointer to array which stores the diagonal blocks in row by row manner

Author

Zhiyang Zhou

Date

2010/10/26

Note

Works for general nb (Xiaozhe)

Modified by Chunsheng Feng, Zheng Li on 08/25/2012
Definition at line 1555 of file [BlaSparseBSR.c](#).

9.73.2.14 fasp_dbsr_getdiaginv()

```
dvector fasp_dbsr_getdiaginv (
    const dBSRmat * A )
```

Get D^{-1} of matrix A.

Parameters

A	Pointer to the dBSRmat matrix
---	---

Author

Xiaozhe Hu

Date

02/19/2013

Note

Works for general nb (Xiaozhe)

Definition at line 486 of file [BlaSparseBSR.c](#).

9.73.2.15 fasp_dbsr_merge_col()

```
INT fasp_dbsr_merge_col (
    dBSRmat * A )
```

Check and merge some same col index in one row.

Parameters

A	Pointer to the original dBSRmat matrix
---	--

Returns

The new merged [dCSRmat](#) matrix

Author

Chunsheng Feng

Date

30/07/2017

Definition at line 2141 of file [BlaSparseBSR.c](#).

9.73.2.16 fasp_dbsr_perm()

```
dBSRmat fasp_dbsr_perm (
    const dBSRmat * A,
    const INT * P )
```

Apply permutation of A, i.e. Aperm=PAP' by the orders given in P.

Parameters

<i>A</i>	Pointer to the original dBSRmat matrix
<i>P</i>	Pointer to the given ordering

Returns

The new ordered [dBSRmat](#) matrix if succeed, NULL if fail

Author

Zheng Li

Date

24/9/2015

Note

$P[i] = k$ means k-th row and column become i-th row and column!

Definition at line [2023](#) of file [BlaSparseBSR.c](#).

9.73.2.17 fasp_dbsr_trans()

```
INT fasp_dbsr_trans (
    const dBSRmat * A,
    dBSRmat * AT )
```

Find A^T from given [dBSRmat](#) matrix A.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>AT</i>	Pointer to the transpose of dBSRmat matrix A

Author

Chunsheng FENG

Date

2011/06/08

Modified by Xiaozhe Hu (08/06/2011)

Definition at line [199](#) of file [BlaSparseBSR.c](#).

9.74 BlaSparseBSR.c

[Go to the documentation of this file.](#)

```

00001
00015 #include <math.h>
00016
00017 #ifdef _OPENMP
00018 #include <omp.h>
00019 #endif
00020
00021 #include "fasp.h"
00022 #include "fasp_functs.h"
00023
00024 /*-----*/
00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00045 dBSRmat fasp_dbsr_create (const INT ROW,
00046                          const INT COL,
00047                          const INT NNZ,
00048                          const INT nb,
00049                          const INT storage_manner)
00050 {
00051     dBSRmat A;
00052
00053     if ( ROW > 0 ) {
00054         A.IA = (INT*)fasp_mem_calloc(ROW+1, sizeof(INT));
00055     }
00056     else {
00057         A.IA = NULL;
00058     }
00059
00060     if ( NNZ > 0 ) {
00061         A.JA = (INT*)fasp_mem_calloc(NNZ, sizeof(INT));
00062     }
00063     else {
00064         A.JA = NULL;
00065     }
00066
00067     if ( nb > 0 && NNZ > 0 ) {
00068         A.val = (REAL*)fasp_mem_calloc(NNZ*nb*nb, sizeof(REAL));
00069     }
00070     else {
00071         A.val = NULL;
00072     }
00073
00074     A.storage_manner = storage_manner;
00075     A.ROW = ROW;
00076     A.COL = COL;
00077     A.NNZ = NNZ;
00078     A.nb = nb;
00079
00080     return A;
00081 }
00082
00099 void fasp_dbsr_alloc (const INT ROW,
00100                      const INT COL,
00101                      const INT NNZ,
00102                      const INT nb,
00103                      const INT storage_manner,
00104                      dBSRmat *A)
00105 {
00106     if ( ROW > 0 ) {
00107         A->IA = (INT*)fasp_mem_calloc(ROW+1, sizeof(INT));
00108     }
00109     else {
00110         A->IA = NULL;
00111     }
00112
00113     if ( NNZ > 0 ) {
00114         A->JA = (INT*)fasp_mem_calloc(NNZ, sizeof(INT));
00115     }
00116     else {
00117         A->JA = NULL;
00118     }
00119
00120     if ( nb > 0 ) {
00121         A->val = (REAL*)fasp_mem_calloc(NNZ*nb*nb, sizeof(REAL));
00122     }
00123     else {

```

```

00124     A->val = NULL;
00125 }
00126
00127 A->storage_manner = storage_manner;
00128 A->ROW = ROW;
00129 A->COL = COL;
00130 A->NNZ = NNZ;
00131 A->nb = nb;
00132
00133 return;
00134 }
00135
00146 void fasp_dbsr_free (dBSRmat *A)
00147 {
00148     if (A==NULL) return;
00149
00150     fasp_mem_free(A->IA); A->IA = NULL;
00151     fasp_mem_free(A->JA); A->JA = NULL;
00152     fasp_mem_free(A->val); A->val = NULL;
00153
00154     A->ROW = 0;
00155     A->COL = 0;
00156     A->NNZ = 0;
00157     A->nb = 0;
00158     A->storage_manner = 0;
00159 }
00160
00172 void fasp_dbsr_cp (const dBSRmat *A,
00173                   dBSRmat *B)
00174 {
00175     B->ROW = A->ROW;
00176     B->COL = A->COL;
00177     B->NNZ = A->NNZ;
00178     B->nb = A->nb;
00179     B->storage_manner = A->storage_manner;
00180
00181     memcpy(B->IA, A->IA, (A->ROW+1)*sizeof(INT));
00182     memcpy(B->JA, A->JA, (A->NNZ)*sizeof(INT));
00183     memcpy(B->val, A->val, (A->NNZ)*(A->nb)*(A->nb)*sizeof(REAL));
00184 }
00185
00199 INT fasp_dbsr_trans (const dBSRmat *A,
00200                     dBSRmat *AT)
00201 {
00202     const INT n = A->ROW, m = A->COL, nnz = A->NNZ, nb = A->nb;
00203
00204     INT status = FASP_SUCCESS;
00205     INT i, j, k, p, inb, jnb, nb2;
00206
00207     AT->ROW = m;
00208     AT->COL = n;
00209     AT->NNZ = nnz;
00210     AT->nb = nb;
00211     AT->storage_manner = A->storage_manner;
00212
00213     AT->IA = (INT*)fasp_mem_malloc(m+1, sizeof(INT));
00214     AT->JA = (INT*)fasp_mem_malloc(nnz, sizeof(INT));
00215     nb2 = nb*nb;
00216
00217     if (A->val) {
00218         AT->val = (REAL*)fasp_mem_malloc(nnz*nb2, sizeof(REAL));
00219     }
00220     else {
00221         AT->val = NULL;
00222     }
00223
00224     // first pass: find the number of nonzeros in the first m-1 columns of A
00225     // Note: these numbers are stored in the array AT->IA from 1 to m-1
00226     fasp_iarray_set(m+1, AT->IA, 0);
00227
00228     for ( j=0; j<nnz; ++j ) {
00229         i=A->JA[j]; // column number of A = row number of A'
00230         if (i<m-1) AT->IA[i+2]++;
00231     }
00232
00233     for ( i=2; i<=m; ++i ) AT->IA[i]=AT->IA[i-1];
00234
00235     // second pass: form A'
00236     if ( A->val ) {
00237         for ( i=0; i<n; ++i ) {
00238             INT ibegin=A->IA[i], iend1=A->IA[i+1];

```

```

00239         for ( p=ibegin; p<iendl; p++ ) {
00240             j=A->JA[p]+1;
00241             k=AT->IA[j];
00242             AT->JA[k]=i;
00243             for ( inb=0; inb<nb; inb++ )
00244                 for ( jnb=0; jnb<nb; jnb++ )
00245                     AT->val[nb2*k + inb*nb + jnb] = A->val[nb2*p + jnb*nb + inb];
00246             AT->IA[j]=k+1;
00247         } // end for p
00248     } // end for i
00249 }
00250 }
00251 else {
00252     for ( i=0; i<n; ++i ) {
00253         INT ibegin=A->IA[i], iendl=A->IA[i+1];
00254         for ( p=ibegin; p<iendl; p++ ) {
00255             j=A->JA[p]+1;
00256             k=AT->IA[j];
00257             AT->JA[k]=i;
00258             AT->IA[j]=k+1;
00259         } // end for p
00260     } // end of i
00261 } // end if
00262 }
00263 return (status);
00264 }
00265 }
00266
00287 SHORT fasp_dbsr_getblk (const dBSRmat *A,
00288                         const INT *Is,
00289                         const INT *Js,
00290                         const INT m,
00291                         const INT n,
00292                         dBSRmat *B)
00293 {
00294     INT status = FASP_SUCCESS;
00295     INT i,j,k,nnz=0;
00296     INT *col_flag;
00297     SHORT use_omp = FALSE;
00298
00299     const INT nb = A->nb;
00300     const INT nb2=nb*nb;
00301
00302 #ifdef _OPENMP
00303     INT myid, mybegin, stride_i, myend, nthreads;
00304     if ( n > OPENMP_HOLDS ) {
00305         use_omp = TRUE;
00306         nthreads = fasp_get_num_threads();
00307     }
00308 #endif
00309
00310     // create colum flags
00311     col_flag = (INT*)fasp_mem_calloc(A->COL,sizeof(INT));
00312
00313     B->ROW=m; B->COL=n; B->nb=nb; B->storage_manner=A->storage_manner;
00314
00315     B->IA = (INT*)fasp_mem_calloc(m+1,sizeof(INT));
00316
00317     if ( use_omp ) {
00318 #ifdef _OPENMP
00319         stride_i = n/nthreads;
00320 #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00321         {
00322             myid = omp_get_thread_num();
00323             mybegin = myid*stride_i;
00324             if ( myid < nthreads-1 ) myend = mybegin+stride_i;
00325             else myend = n;
00326             for ( i = mybegin; i < myend; ++i ) {
00327                 col_flag[Js[i]]=i+1;
00328             }
00329         }
00330 #endif
00331     }
00332     else {
00333         for ( i=0; i<n; ++i ) col_flag[Js[i]]=i+1;
00334     }
00335
00336     // first pass: count nonzeros for sub matrix
00337     B->IA[0] = 0;
00338     for ( i=0; i<m; ++i ) {
00339         for ( k=A->IA[Is[i]]; k<A->IA[Is[i]+1]; ++k ) {

```

```

00340         j=A->JA[k];
00341         if (col_flag[j]>0) nnz++;
00342     } /* end for k */
00343     B->IA[i+1] = nnz;
00344 } /* end for i */
00345 B->NNZ = nnz;
00346
00347 // allocate
00348 B->JA = (INT*)fasp_mem_calloc(nnz,sizeof(INT));
00349 B->val = (REAL*)fasp_mem_calloc(nnz*nb2,sizeof(REAL));
00350
00351 // second pass: copy data to B
00352 nnz = 0;
00353 for ( i=0; i<m; ++i) {
00354     for ( k=A->IA[Is[i]]; k<A->IA[Is[i]+1]; ++k ) {
00355         j = A->JA[k];
00356         if ( col_flag[j] > 0 ) {
00357             B->JA[nnz]=col_flag[j]-1;
00358             memcpy(B->val+nnz*nb2, A->val+k*nb2, nb2*sizeof(REAL));
00359             nnz++;
00360         }
00361     } /* end for k */
00362 } /* end for i */
00363
00364 fasp_mem_free(col_flag); col_flag = NULL;
00365
00366 return(status);
00367 }
00368
00385 SHORT fasp_dbsr_diagpref (dBSRmat *A)
00386 {
00387     SHORT      status = FASP_SUCCESS;
00388     const INT   num_rowsA = A -> ROW;
00389     const INT   num_colsA = A -> COL;
00390     const INT   nb       = A->nb;
00391     const INT   nb2      = nb*nb;
00392
00393     const INT   *A_i      = A -> IA;
00394     INT         *A_j      = A -> JA;
00395     REAL        *A_data   = A -> val;
00396
00397     INT i, j, tempi, row_size;
00398
00399 #ifdef _OPENMP
00400     // variables for OpenMP
00401     INT myid, mybegin, myend, ibegin, iend;
00402     INT nthreads = fasp_get_num_threads();
00403 #endif
00404
00405     /* the matrix should be square */
00406     if (num_rowsA != num_colsA) return ERROR_INPUT_PAR;
00407
00408 #ifdef _OPENMP
00409     if (num_rowsA > OPENMP_HOLDS) {
00410         REAL *tempd = (REAL*)fasp_mem_calloc(nb2*nthreads, sizeof(REAL));
00411 #pragma omp parallel for private (myid,mybegin,myend,i,j,tempi,ibegin,iend)
00412         for (myid = 0; myid < nthreads; myid++) {
00413             fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
00414             for (i = mybegin; i < myend; i++) {
00415                 ibegin = A_i[i+1]; iend = A_i[i];
00416                 for (j = ibegin; j < iend; j++) {
00417                     if (A_j[j] == i) {
00418                         if (j != ibegin) {
00419                             // swap index
00420                             tempi = A_j[ibegin];
00421                             A_j[ibegin] = A_j[j];
00422                             A_j[j] = tempi;
00423                             // swap block
00424                             memcpy(tempd+myid*nb2, A_data+ibegin*nb2, (nb2)*sizeof(REAL));
00425                             memcpy(A_data+ibegin*nb2, A_data+j*nb2, (nb2)*sizeof(REAL));
00426                             memcpy(A_data+j*nb2, tempd+myid*nb2, (nb2)*sizeof(REAL));
00427                         }
00428                         break;
00429                     }
00430                 } /* diagonal element is missing */
00431                 if (j == iend-1) {
00432                     status = -2;
00433                     break;
00434                 }
00435             }
00436         }
00437     }

```

```

00437     }
00438     fasp_mem_free(tempd); tempd = NULL;
00439 }
00440 else {
00441 #endif
00442     REAL *tempd = (REAL*)fasp_mem_calloc(nb2, sizeof(REAL));
00443     for (i = 0; i < num_rowsA; i++) {
00444         row_size = A_i[i+1] - A_i[i];
00445         for (j = 0; j < row_size; j++) {
00446             if (A_j[j] == i) {
00447                 if (j != 0) {
00448                     // swap index
00449                     tempi = A_j[0];
00450                     A_j[0] = A_j[j];
00451                     A_j[j] = tempi;
00452                     // swap block
00453                     memcpy(tempd, A_data, (nb2)*sizeof(REAL));
00454                     memcpy(A_data, A_data+j*nb2, (nb2)*sizeof(REAL));
00455                     memcpy(A_data+j*nb2, tempd, (nb2)*sizeof(REAL));
00456                 }
00457                 break;
00458             }
00459             /* diagonal element is missing */
00460             if (j == row_size-1) return -2;
00461         }
00462         A_j += row_size;
00463         A_data += row_size*nb2;
00464     }
00465     fasp_mem_free(tempd); tempd = NULL;
00466 #ifdef _OPENMP
00467 }
00468 #endif
00469
00470     if (status < 0) return status;
00471     else return FASP_SUCCESS;
00472 }
00473
00486 dvector fasp_dbsr_getdiaginv (const dBSRmat *A)
00487 {
00488     // members of A
00489     const INT ROW = A->ROW;
00490     const INT nb = A->nb;
00491     const INT nb2 = nb*nb;
00492     const INT size = ROW*nb2;
00493     const INT *IA = A->IA;
00494     const INT *JA = A->JA;
00495     REAL *val = A->val;
00496
00497     dvector diaginv;
00498
00499     INT i,k;
00500
00501     // Variables for OpenMP
00502     SHORT nthreads = 1, use_openmp = FALSE;
00503     INT myid, mybegin, myend;
00504
00505 #ifdef _OPENMP
00506     if ( ROW > OPENMP_HOLDS ) {
00507         use_openmp = TRUE;
00508         nthreads = fasp_get_num_threads();
00509     }
00510 #endif
00511
00512     // allocate memory
00513     diaginv.row = size;
00514     diaginv.val = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00515
00516     // get all the diagonal sub-blocks
00517     if (use_openmp) {
00518 #ifdef _OPENMP
00519 #pragma omp parallel for private(myid, i, mybegin, myend, k)
00520 #endif
00521         for (myid = 0; myid < nthreads; myid++) {
00522             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00523             for (i = mybegin; i < myend; ++i) {
00524                 for (k = IA[i]; k < IA[i+1]; ++k) {
00525                     if (JA[k] == i)
00526                         memcpy(diaginv.val+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00527                 }
00528             }
00529         }
00530     }

```



```

00530     }
00531     else {
00532         for (i = 0; i < ROW; ++i) {
00533             for (k = IA[i]; k < IA[i+1]; ++k) {
00534                 if (JA[k] == i)
00535                     memcpy(diaginv.val+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00536             }
00537         }
00538     }
00539     // compute the inverses of all the diagonal sub-blocks
00540     if (use_openmp) {
00541 #ifdef _OPENMP
00542 #pragma omp parallel for private(myid, i, mybegin, myend)
00543 #endif
00544         for (myid = 0; myid < nthreads; myid++) {
00545             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00546             if (nb > 1) {
00547                 for (i = mybegin; i < myend; ++i) {
00548                     fasp_smat_inv(diaginv.val+i*nb2, nb);
00549                 }
00550             }
00551             else {
00552                 for (i = mybegin; i < myend; ++i) {
00553                     // zero-diagonal should be tested previously
00554                     diaginv.val[i] = 1.0 / diaginv.val[i];
00555                 }
00556             }
00557         }
00558     }
00559     else {
00560         if (nb > 1) {
00561             for (i = 0; i < ROW; ++i) {
00562                 fasp_smat_inv(diaginv.val+i*nb2, nb);
00563             }
00564         }
00565         else {
00566             for (i = 0; i < ROW; ++i) {
00567                 // zero-diagonal should be tested previously
00568                 diaginv.val[i] = 1.0 / diaginv.val[i];
00569             }
00570         }
00571     }
00572     return (diaginv);
00573 }
00574 }
00575
00591 dBSRmat fasp_dbsr_diaginv (const dBSRmat *A)
00592 {
00593     // members of A
00594     const INT    ROW  = A->ROW;
00595     const INT    COL  = A->COL;
00596     const INT    NNZ  = A->NNZ;
00597     const INT    nb   = A->nb;
00598     const INT    nb2  = nb*nb;
00599     const INT    size = ROW*nb2;
00600     const INT    *IA   = A->IA;
00601     const INT    *JA   = A->JA;
00602     REAL         *val  = A->val;
00603
00604     // create a dBSRmat B
00605     dBSRmat B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
00606     INT *IAb = B.IA;
00607     INT *JAb = B.JA;
00608     REAL *valb = B.val;
00609
00610     INT i, j, k, m, l;
00611
00612     // variables for OpenMP
00613     SHORT nthreads = 1, use_openmp = FALSE;
00614     INT myid, mybegin, myend;
00615
00616     // allocate memory
00617     REAL *diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00618
00619     if ( IAb ) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
00620     else goto FINISHED;
00621
00622     if ( JAb ) memcpy(JAb, JA, NNZ*sizeof(INT));
00623     else goto FINISHED;
00624
00625 #ifdef _OPENMP

```

```

00626     if (ROW > OPENMP_HOLDS) {
00627         use_omp = TRUE;
00628         nthreads = fasp_get_num_threads();
00629     }
00630 #endif
00631
00632     // get all the diagonal sub-blocks
00633     if (use_omp) {
00634 #ifdef _OPENMP
00635 #pragma omp parallel for private(myid, i, mybegin, myend, k)
00636 #endif
00637         for (myid = 0; myid < nthreads; myid++) {
00638             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00639             for (i = mybegin; i < myend; ++i) {
00640                 for (k = IA[i]; k < IA[i+1]; ++k) {
00641                     if (JA[k] == i)
00642                         memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00643                 }
00644             }
00645         }
00646     }
00647     else {
00648         for (i = 0; i < ROW; ++i) {
00649             for (k = IA[i]; k < IA[i+1]; ++k) {
00650                 if (JA[k] == i)
00651                     memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00652             }
00653         }
00654     }
00655
00656     // compute the inverses of all the diagonal sub-blocks
00657     if (use_omp) {
00658 #ifdef _OPENMP
00659 #pragma omp parallel for private(myid, i, mybegin, myend)
00660 #endif
00661         for (myid = 0; myid < nthreads; myid++) {
00662             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00663             if (nb > 1) {
00664                 for (i = mybegin; i < myend; ++i) {
00665                     fasp_smat_inv(diaginv+i*nb2, nb);
00666                 }
00667             }
00668             else {
00669                 for (i = mybegin; i < myend; ++i) {
00670                     // zero-diagonal should be tested previously
00671                     diaginv[i] = 1.0 / diaginv[i];
00672                 }
00673             }
00674         }
00675     }
00676     else {
00677         if (nb > 1) {
00678             for (i = 0; i < ROW; ++i) {
00679                 fasp_smat_inv(diaginv+i*nb2, nb);
00680             }
00681         }
00682         else {
00683             for (i = 0; i < ROW; ++i) {
00684                 // zero-diagonal should be tested previously
00685                 diaginv[i] = 1.0 / diaginv[i];
00686             }
00687         }
00688     }
00689
00690     // compute D^{-1}*A
00691     if (use_omp) {
00692 #ifdef _OPENMP
00693 #pragma omp parallel for private(myid, mybegin, myend, i, k, m, j, l)
00694 #endif
00695         for (myid = 0; myid < nthreads; myid++) {
00696             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00697             for (i = mybegin; i < myend; ++i) {
00698                 for (k = IA[i]; k < IA[i+1]; ++k) {
00699                     m = k*nb2;
00700                     j = JA[k];
00701                     if (j == i) {
00702                         // Identity sub-block
00703                         memset(valb+m, 0X0, nb2*sizeof(REAL));
00704                         for (l = 0; l < nb; l++) valb[m+l*nb+1] = 1.0;
00705                     }
00706                     else {

```

```

00707             fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
00708         }
00709     }
00710 }
00711 }
00712 }
00713 else {
00714     for (i = 0; i < ROW; ++i) {
00715         for (k = IA[i]; k < IA[i+1]; ++k) {
00716             m = k*nb2;
00717             j = JA[k];
00718             if (j == i) {
00719                 // Identity sub-block
00720                 memset(valb+m, 0X0, nb2*sizeof(REAL));
00721                 for (l = 0; l < nb; l++) valb[m+l*nb+1] = 1.0;
00722             }
00723             else {
00724                 fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
00725             }
00726         }
00727     }
00728 }
00729
00730 FINISHED:
00731     fasp_mem_free(diaginv); diaginv = NULL;
00732
00733     return (B);
00734 }
00735
00751 dBSRmat fasp_dbsr_diaginv2 (const dBSRmat *A,
00752                             REAL          *diaginv)
00753 {
00754     // members of A
00755     const INT ROW = A->ROW;
00756     const INT COL = A->COL;
00757     const INT NNZ = A->NNZ;
00758     const INT nb  = A->nb, nbp1 = nb+1;
00759     const INT nb2 = nb*nb;
00760
00761     INT *IA = A->IA;
00762     INT *JA = A->JA;
00763     REAL *val = A->val;
00764
00765     dBSRmat B;
00766     INT *IAb = NULL;
00767     INT *JAb = NULL;
00768     REAL *valb = NULL;
00769
00770     INT i,k,m,l,ibegin,iend;
00771
00772     // Variables for OpenMP
00773     SHORT nthreads = 1, use_openmp = FALSE;
00774     INT myid, mybegin, myend;
00775
00776 #ifdef _OPENMP
00777     if (ROW > OPENMP_HOLDS) {
00778         use_openmp = TRUE;
00779         nthreads = fasp_get_num_threads();
00780     }
00781 #endif
00782
00783     // Create a dBSRmat 'B'
00784     B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
00785     IAb = B.IA;
00786     JAb = B.JA;
00787     valb = B.val;
00788
00789     if (IAb) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
00790     else goto FINISHED;
00791
00792     if (JAb) memcpy(JAb, JA, NNZ*sizeof(INT));
00793     else goto FINISHED;
00794
00795     // compute D^{-1}*A
00796     if (use_openmp) {
00797 #ifdef _OPENMP
00798         #pragma omp parallel for private (myid, i, mybegin, myend, ibegin, iend, k, m, l)
00799         #endif
00800         for (myid = 0; myid < nthreads; myid++) {
00801             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00802             for (i = mybegin; i < myend; ++i) {

```

```

00803         ibegin = IA[i]; iend = IA[i+1];
00804         for (k = ibegin; k < iend; ++k) {
00805             m = k*nb2;
00806             if (JA[k] != i) {
00807                 fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
00808             }
00809             else {
00810                 // Identity sub-block
00811                 memset(valb+m, 0X0, nb2*sizeof(REAL));
00812                 for (l = 0; l < nb; l++) valb[m+l*nbp1] = 1.0;
00813             }
00814         }
00815     }
00816 }
00817 }
00818 else {
00819     // compute D^{-1}*A
00820     for (i = 0; i < ROW; ++i) {
00821         ibegin = IA[i]; iend = IA[i+1];
00822         for (k = ibegin; k < iend; ++k) {
00823             m = k*nb2;
00824             if (JA[k] != i) {
00825                 fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
00826             }
00827             else {
00828                 // Identity sub-block
00829                 memset(valb+m, 0X0, nb2*sizeof(REAL));
00830                 for (l = 0; l < nb; l++) valb[m+l*nbp1] = 1.0;
00831             }
00832         }
00833     }
00834 }
00835 }
00836 FINISHED:
00837     return (B);
00838 }
00839
00857 dBSRmat fasp_dbsr_diaginv3 (const dBSRmat *A,
00858                             REAL *diaginv)
00859 {
00860     dBSRmat B;
00861     // members of A
00862     INT ROW = A->ROW;
00863     INT ROW_plus_one = ROW+1;
00864     INT COL = A->COL;
00865     INT NNZ = A->NNZ;
00866     INT nb = A->nb;
00867     INT *IA = A->IA;
00868     INT *JA = A->JA;
00869     REAL *val = A->val;
00870
00871     INT *IAb = NULL;
00872     INT *JAb = NULL;
00873     REAL *valb = NULL;
00874
00875     INT nb2 = nb*nb;
00876     INT i,j,k,m;
00877
00878     SHORT use_openmp = FALSE;
00879
00880 #ifdef _OPENMP
00881     INT myid, mybegin, myend, stride_i, nthreads = 1;
00882     if ( ROW > OPENMP_HOLDS ) {
00883         use_openmp = TRUE;
00884         nthreads = fasp_get_num_threads();
00885     }
00886 #endif
00887
00888     // Create a dBSRmat 'B'
00889     B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
00890
00891     IAb = B.IA;
00892     JAb = B.JA;
00893     valb = B.val;
00894
00895     fasp_iarray_cp(ROW_plus_one, IA, IAb);
00896     fasp_iarray_cp(NNZ, JA, JAb);
00897
00898     switch (nb) {
00899
00900         case 2:

```

```

00901         // main loop
00902         if (use_omp) {
00903 #ifdef _OPENMP
00904             stride_i = ROW/nthreads;
00905 #pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
00906             {
00907                 myid = omp_get_thread_num();
00908                 mybegin = myid*stride_i;
00909                 if (myid < nthreads-1) myend = mybegin+stride_i;
00910                 else myend = ROW;
00911                 for (i=mybegin; i < myend; ++i) {
00912                     // get the diagonal sub-blocks
00913
00914                     k = IA[i];
00915                     m = k*4;
00916                     memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
00917                     fasp_smat_identity_nc2(valb+m);
00918
00919                     // compute the inverses of the diagonal sub-blocks
00920                     fasp_smat_inv_nc2(diaginv+i*4);
00921                     // compute D^{-1}*A
00922                     for (k = IA[i]+1; k < IA[i+1]; ++k)
00923                     {
00924                         m = k*4;
00925                         j = JA[k];
00926                         fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
00927                     }
00928                 } // end of main loop
00929             }
00930 #endif
00931         }
00932         else {
00933             // main loop
00934             for (i = 0; i < ROW; ++i) {
00935                 // get the diagonal sub-blocks
00936                 k = IA[i];
00937                 m = k*4;
00938                 memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
00939                 fasp_smat_identity_nc2(valb+m);
00940
00941                 // compute the inverses of the diagonal sub-blocks
00942                 fasp_smat_inv_nc2(diaginv+i*4);
00943                 // compute D^{-1}*A
00944                 for (k = IA[i]+1; k < IA[i+1]; ++k) {
00945                     m = k*4;
00946                     fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
00947                 }
00948             } // end of main loop
00949         }
00950
00951         break;
00952
00953     case 3:
00954         // main loop
00955         if (use_omp) {
00956 #ifdef _OPENMP
00957             stride_i = ROW/nthreads;
00958 #pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
00959             {
00960                 myid = omp_get_thread_num();
00961                 mybegin = myid*stride_i;
00962                 if (myid < nthreads-1) myend = mybegin+stride_i;
00963                 else myend = ROW;
00964                 for (i=mybegin; i < myend; ++i) {
00965                     // get the diagonal sub-blocks
00966                     for (k = IA[i]; k < IA[i+1]; ++k) {
00967                         if (JA[k] == i) {
00968                             m = k*9;
00969                             memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
00970                             fasp_smat_identity_nc3(valb+m);
00971                         }
00972                     }
00973                     // compute the inverses of the diagonal sub-blocks
00974                     fasp_smat_inv_nc3(diaginv+i*9);
00975                     // compute D^{-1}*A
00976                     for (k = IA[i]; k < IA[i+1]; ++k) {
00977                         m = k*9;
00978                         j = JA[k];
00979                         if (j != i) fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
00980                     }
00981                 } // end of main loop

```

```

00982         }
00983     #endif
00984     }
00985
00986     else {
00987         for (i = 0; i < ROW; ++i) {
00988             // get the diagonal sub-blocks
00989             for (k = IA[i]; k < IA[i+1]; ++k) {
00990                 if (JA[k] == i) {
00991                     m = k*9;
00992                     memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
00993                     fasp_smat_identity_nc3(valb+m);
00994                 }
00995             }
00996             #if DEBUG_MODE > 0
00997                 printf("### DEBUG: row, col = %d\n", i);
00998             #endif
00999             // compute the inverses of the diagonal sub-blocks
01000             fasp_smat_inv_nc3(diaginv+i*9);
01001
01002             // compute D^{-1}*A
01003             for (k = IA[i]; k < IA[i+1]; ++k) {
01004                 m = k*9;
01005                 j = JA[k];
01006                 if (j != i) fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
01007             }
01008             }// end of main loop
01009         }
01010
01011         break;
01012
01013     case -5:
01014         // main loop
01015         if (use_openmp) {
01016             #ifdef _OPENMP
01017                 stride_i = ROW/nthreads;
01018                 #pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
01019                 {
01020                     myid = omp_get_thread_num();
01021                     mybegin = myid*stride_i;
01022                     if (myid < nthreads-1) myend = mybegin+stride_i;
01023                     else myend = ROW;
01024                     for (i=mybegin; i < myend; ++i) {
01025                         // get the diagonal sub-blocks
01026                         for (k = IA[i]; k < IA[i+1]; ++k) {
01027                             if (JA[k] == i) {
01028                                 m = k*25;
01029                                 memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
01030                                 fasp_smat_identity_nc5(valb+m);
01031                             }
01032                         }
01033
01034                         // compute the inverses of the diagonal sub-blocks
01035                         fasp_smat_inv_nc5(diaginv+i*25);
01036
01037                         // compute D^{-1}*A
01038                         for (k = IA[i]; k < IA[i+1]; ++k) {
01039                             m = k*25;
01040                             j = JA[k];
01041                             if (j != i) fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
01042                         }
01043                     }// end of main loop
01044                 }
01045             #endif
01046         }
01047
01048     else {
01049         for (i = 0; i < ROW; ++i) {
01050             // get the diagonal sub-blocks
01051             for (k = IA[i]; k < IA[i+1]; ++k) {
01052                 if (JA[k] == i)
01053                 {
01054                     m = k*25;
01055                     memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
01056                     fasp_smat_identity_nc5(valb+m);
01057                 }
01058             }
01059         }
01060
01061         // compute the inverses of the diagonal sub-blocks
01062         // fasp_smat_inv_nc5(diaginv+i*25); // Not numerically stable!!! --zcs 04/26/2021

```

```

01063         fasp_smat_invp_nc(diaginv + i * 25, 5);
01064
01065 #if 0
01066     REAL aa[25], bb[25]; // for debug inverse of diag
01067     for (k = 0; k < 25; k++) bb[k] = diaginv[i * 25 + k]; // before inversion
01068     for (k = 0; k < 25; k++) aa[k] = diaginv[i * 25 + k]; // after inversion
01069
01070     printf("### DEBUG: Check inverse matrix...\n");
01071     printf("###-----\n");
01072     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
01073         bb[0]*aa[0] + bb[1]*aa[5] + bb[2]*aa[10] + bb[3]*aa[15] + bb[4]*aa[20],
01074         bb[0]*aa[1] + bb[1]*aa[6] + bb[2]*aa[11] + bb[3]*aa[16] + bb[4]*aa[21],
01075         bb[0]*aa[2] + bb[1]*aa[7] + bb[2]*aa[12] + bb[3]*aa[17] + bb[4]*aa[22],
01076         bb[0]*aa[3] + bb[1]*aa[8] + bb[2]*aa[13] + bb[3]*aa[18] + bb[4]*aa[23],
01077         bb[0]*aa[4] + bb[1]*aa[9] + bb[2]*aa[14] + bb[3]*aa[19] + bb[4]*aa[24]);
01078     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
01079         bb[5]*aa[0] + bb[6]*aa[5] + bb[7]*aa[10] + bb[8]*aa[15] + bb[9]*aa[20],
01080         bb[5]*aa[1] + bb[6]*aa[6] + bb[7]*aa[11] + bb[8]*aa[16] + bb[9]*aa[21],
01081         bb[5]*aa[2] + bb[6]*aa[7] + bb[7]*aa[12] + bb[8]*aa[17] + bb[9]*aa[22],
01082         bb[5]*aa[3] + bb[6]*aa[8] + bb[7]*aa[13] + bb[8]*aa[18] + bb[9]*aa[23],
01083         bb[5]*aa[4] + bb[6]*aa[9] + bb[7]*aa[14] + bb[8]*aa[19] + bb[9]*aa[24]);
01084     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
01085         bb[10]*aa[0] + bb[11]*aa[5] + bb[12]*aa[10] + bb[13]*aa[15] + bb[14]*aa[20],
01086         bb[10]*aa[1] + bb[11]*aa[6] + bb[12]*aa[11] + bb[13]*aa[16] + bb[14]*aa[21],
01087         bb[10]*aa[2] + bb[11]*aa[7] + bb[12]*aa[12] + bb[13]*aa[17] + bb[14]*aa[22],
01088         bb[10]*aa[3] + bb[11]*aa[8] + bb[12]*aa[13] + bb[13]*aa[18] + bb[14]*aa[23],
01089         bb[10]*aa[4] + bb[11]*aa[9] + bb[12]*aa[14] + bb[13]*aa[19] + bb[14]*aa[24]);
01090
01091     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
01092         bb[15]*aa[0] + bb[16]*aa[5] + bb[17]*aa[10] + bb[18]*aa[15] + bb[19]*aa[20],
01093         bb[15]*aa[1] + bb[16]*aa[6] + bb[17]*aa[11] + bb[18]*aa[16] + bb[19]*aa[21],
01094         bb[15]*aa[2] + bb[16]*aa[7] + bb[17]*aa[12] + bb[18]*aa[17] + bb[19]*aa[22],
01095         bb[15]*aa[3] + bb[16]*aa[8] + bb[17]*aa[13] + bb[18]*aa[18] + bb[19]*aa[23],
01096         bb[15]*aa[4] + bb[16]*aa[9] + bb[17]*aa[14] + bb[18]*aa[19] + bb[19]*aa[24]);
01097
01098     printf("### %12.5e %12.5e %12.5e %12.5e %12.5e\n",
01099         bb[20]*aa[0] + bb[21]*aa[5] + bb[22]*aa[10] + bb[23]*aa[15] + bb[24]*aa[20],
01100         bb[20]*aa[1] + bb[21]*aa[6] + bb[22]*aa[11] + bb[23]*aa[16] + bb[24]*aa[21],
01101         bb[20]*aa[2] + bb[21]*aa[7] + bb[22]*aa[12] + bb[23]*aa[17] + bb[24]*aa[22],
01102         bb[20]*aa[3] + bb[21]*aa[8] + bb[22]*aa[13] + bb[23]*aa[18] + bb[24]*aa[23],
01103         bb[20]*aa[4] + bb[21]*aa[9] + bb[22]*aa[14] + bb[23]*aa[19] + bb[24]*aa[24]);
01104     printf("###-----\n");
01105 #endif
01106
01107     // compute D^{-1}*A
01108     for (k = IA[i]; k < IA[i+1]; ++k) {
01109         m = k*25;
01110         j = JA[k];
01111         if (j != i) fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
01112     } // end of main loop
01113 }
01114
01115 break;
01116
01117 case -7:
01118     // main loop
01119     if (use_openmp) {
01120 #ifdef _OPENMP
01121         stride_i = ROW/nthreads;
01122 #pragma omp parallel private(myid, mybegin, myend, i, k, m, j) num_threads(nthreads)
01123         {
01124             myid = omp_get_thread_num();
01125             mybegin = myid*stride_i;
01126             if (myid < nthreads-1) myend = mybegin+stride_i;
01127             else myend = ROW;
01128             for (i=mybegin; i < myend; ++i) {
01129                 // get the diagonal sub-blocks

```

```

01129         for (k = IA[i]; k < IA[i+1]; ++k) {
01130             if (JA[k] == i) {
01131                 m = k*49;
01132                 memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
01133                 fasp_smat_identity_nc7(valb+m);
01134             }
01135         }
01136
01137         // compute the inverses of the diagonal sub-blocks
01138         fasp_smat_inv_nc7(diaginv+i*49);
01139
01140         // compute D^{-1}*A
01141         for (k = IA[i]; k < IA[i+1]; ++k) {
01142             m = k*49;
01143             j = JA[k];
01144             if (j != i) fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
01145         }
01146     } // end of main loop
01147 }
01148 #endif
01149
01150     }
01151     else {
01152         for (i = 0; i < ROW; ++i) {
01153             // get the diagonal sub-blocks
01154             for (k = IA[i]; k < IA[i+1]; ++k) {
01155                 if (JA[k] == i) {
01156                     m = k*49;
01157                     memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
01158                     fasp_smat_identity_nc7(valb+m);
01159                 }
01160             }
01161
01162             // compute the inverses of the diagonal sub-blocks
01163             // fasp_smat_inv_nc7(diaginv+i*49); // Not numerically stable!!! --zcs 04/26/2021
01164             fasp_smat_invp_nc(diaginv + i * 49, 7);
01165
01166             // compute D^{-1}*A
01167             for (k = IA[i]; k < IA[i+1]; ++k) {
01168                 m = k*49;
01169                 j = JA[k];
01170                 if (j != i) fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
01171             }
01172         } // end of main loop
01173     }
01174
01175     break;
01176
01177     default:
01178         // main loop
01179         if (use_omp) {
01180             #ifdef _OPENMP
01181                 stride_i = ROW/nthreads;
01182                 #pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
01183                 {
01184                     myid = omp_get_thread_num();
01185                     mybegin = myid*stride_i;
01186                     if (myid < nthreads-1) myend = mybegin+stride_i;
01187                     else myend = ROW;
01188                     for (i=mybegin; i < myend; ++i) {
01189                         // get the diagonal sub-blocks
01190                         for (k = IA[i]; k < IA[i+1]; ++k) {
01191                             if (JA[k] == i) {
01192                                 m = k*nb2;
01193                                 memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
01194                                 fasp_smat_identity(valb+m, nb, nb2);
01195                             }
01196                         }
01197
01198                         // compute the inverses of the diagonal sub-blocks
01199                         fasp_smat_inv(diaginv+i*nb2, nb);
01200
01201                         // compute D^{-1}*A
01202                         for (k = IA[i]; k < IA[i+1]; ++k) {
01203                             m = k*nb2;
01204                             j = JA[k];
01205                             if (j != i) fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
01206                         }
01207                     } // end of main loop
01208                 }
01209             #endif

```



```

01210     }
01211     else {
01212         for (i = 0; i < ROW; ++i) {
01213             // get the diagonal sub-blocks
01214             for (k = IA[i]; k < IA[i+1]; ++k) {
01215                 if (JA[k] == i) {
01216                     m = k*nb2;
01217                     memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
01218                     fasp_smat_identity(valb+m, nb, nb2);
01219                 }
01220             }
01221
01222             // compute the inverses of the diagonal sub-blocks
01223             // fasp_smat_inv(diaginv+i*nb2, nb); // Not numerically stable!!! --zcs 04/26/2021
01224             fasp_smat_inv_nc(diaginv + i * nb2, nb);
01225
01226             // compute D^{-1}*A
01227             for (k = IA[i]; k < IA[i+1]; ++k) {
01228                 m = k*nb2;
01229                 j = JA[k];
01230                 if (j != i) fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
01231             }
01232         } // end of main loop
01233     }
01234     break;
01235 }
01236
01237 return (B);
01238 }
01239
01240 dBSRmat fasp_dbsr_diaginv4 (const dBSRmat *A,
01241                             REAL *diaginv)
01242 {
01243     // members of A
01244     const INT ROW = A->ROW;
01245     const INT COL = A->COL;
01246     const INT NNZ = A->NNZ;
01247     const INT nb = A->nb;
01248     const INT nb2 = nb*nb;
01249     const INT *IA = A->IA;
01250     const INT *JA = A->JA;
01251     REAL *val = A->val;
01252
01253     dBSRmat B;
01254     INT *IAb = NULL;
01255     INT *JAb = NULL;
01256     REAL *valb = NULL;
01257
01258     INT i,k,m;
01259     INT ibegin, iend;
01260
01261     // Variables for OpenMP
01262     SHORT nthreads = 1, use_openmp = FALSE;
01263     INT myid, mybegin, myend;
01264
01265     #ifdef _OPENMP
01266     if (ROW > OPENMP_HOLDS) {
01267         use_openmp = TRUE;
01268         nthreads = fasp_get_num_threads();
01269     }
01270     #endif
01271
01272     // Create a dBSRmat 'B'
01273     B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
01274
01275     IAb = B.IA;
01276     JAb = B.JA;
01277     valb = B.val;
01278
01279     if (IAb) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
01280     else goto FINISHED;
01281
01282     if (JAb) memcpy(JAb, JA, NNZ*sizeof(INT));
01283     else goto FINISHED;
01284
01285     switch (nb) {
01286     case 2:
01287         if (use_openmp) {
01288             #ifdef _openmp

```

```

01310 #pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
01311 #endif
01312         for (myid = 0; myid < nthreads; myid++) {
01313             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01314             for (i = mybegin; i < myend; ++i) {
01315                 ibegin = IA[i]; iend = IA[i+1];
01316                 // get the diagonal sub-blocks (It is the first block of each row)
01317                 m = ibegin*4;
01318                 memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
01319                 fasp_smat_identity_nc2(valb+m);
01320
01321                 // compute the inverses of the diagonal sub-blocks
01322                 fasp_smat_inv_nc2(diaginv+i*4); // fixed by Zheng Li
01323
01324                 // compute  $D^{-1} \cdot A$ 
01325                 for (k = ibegin+1; k < iend; ++k) {
01326                     m = k*4;
01327                     fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
01328                 }
01329             }
01330         } // end of main loop
01331     }
01332     else {
01333         for (i = 0; i < ROW; ++i) {
01334             ibegin = IA[i]; iend = IA[i+1];
01335             // get the diagonal sub-blocks (It is the first block of each row)
01336             m = ibegin*4;
01337             memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
01338             fasp_smat_identity_nc2(valb+m);
01339
01340             // compute the inverses of the diagonal sub-blocks
01341             fasp_smat_inv_nc2(diaginv+i*4); // fixed by Zheng Li
01342
01343             // compute  $D^{-1} \cdot A$ 
01344             for (k = ibegin+1; k < iend; ++k) {
01345                 m = k*4;
01346                 fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
01347             }
01348         } // end of main loop
01349     }
01350
01351     break;
01352
01353     case 3:
01354         if (use_omp) {
01355 #ifdef _openmp
01356 #pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
01357 #endif
01358             for (myid = 0; myid < nthreads; myid++) {
01359                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01360                 for (i = mybegin; i < myend; ++i) {
01361                     ibegin = IA[i]; iend = IA[i+1];
01362                     // get the diagonal sub-blocks (It is the first block of each row)
01363                     m = ibegin*9;
01364                     memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
01365                     fasp_smat_identity_nc3(valb+m);
01366                     // compute the inverses of the diagonal sub-blocks
01367                     fasp_smat_inv_nc3(diaginv+i*9);
01368                     // compute  $D^{-1} \cdot A$ 
01369                     for (k = ibegin+1; k < iend; ++k) {
01370                         m = k*9;
01371                         fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
01372                     }
01373                 }
01374             } // end of main loop
01375         }
01376     else {
01377         for (i = 0; i < ROW; ++i) {
01378             ibegin = IA[i]; iend = IA[i+1];
01379             // get the diagonal sub-blocks (It is the first block of each row)
01380             m = ibegin*9;
01381             memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
01382             fasp_smat_identity_nc3(valb+m);
01383
01384             // compute the inverses of the diagonal sub-blocks
01385             fasp_smat_inv_nc3(diaginv+i*9);
01386
01387             // compute  $D^{-1} \cdot A$ 
01388             for (k = ibegin+1; k < iend; ++k) {
01389                 m = k*9;
01390                 fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);

```

```

01391     }
01392     } // end of main loop
01393 }
01394
01395     break;
01396
01397     case 5:
01398         if (use_omp) {
01399             #ifdef _OPENMP
01400             #pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
01401             #endif
01402                 for (myid = 0; myid < nthreads; myid++) {
01403                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01404                     for (i = mybegin; i < myend; ++i) {
01405                         // get the diagonal sub-blocks
01406                         ibegin = IA[i]; iend = IA[i+1];
01407                         m = ibegin*25;
01408                         memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
01409                         fasp_smat_identity_nc5(valb+m);
01410
01411                         // compute the inverses of the diagonal sub-blocks
01412                         fasp_smat_inv_nc5(diaginv+i*25);
01413
01414                         // compute D^{-1}*A
01415                         for (k = ibegin+1; k < iend; ++k) {
01416                             m = k*25;
01417                             fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
01418                         }
01419                     }
01420                 }
01421         }
01422         else {
01423             for (i = 0; i < ROW; ++i) {
01424                 // get the diagonal sub-blocks
01425                 ibegin = IA[i]; iend = IA[i+1];
01426                 m = ibegin*25;
01427                 memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
01428                 fasp_smat_identity_nc5(valb+m);
01429
01430                 // compute the inverses of the diagonal sub-blocks
01431                 fasp_smat_inv_nc5(diaginv+i*25);
01432
01433                 // compute D^{-1}*A
01434                 for (k = ibegin+1; k < iend; ++k) {
01435                     m = k*25;
01436                     fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
01437                 }
01438             } // end of main loop
01439         }
01440         break;
01441
01442     case 7:
01443         if (use_omp) {
01444             #ifdef _OPENMP
01445             #pragma omp parallel for private(myid, i, mybegin, myend, ibegin, iend, m, k)
01446             #endif
01447                 for (myid = 0; myid < nthreads; myid++) {
01448                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01449                     for (i = mybegin; i < myend; ++i) {
01450                         // get the diagonal sub-blocks
01451                         ibegin = IA[i]; iend = IA[i+1];
01452                         m = ibegin*49;
01453                         memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
01454                         fasp_smat_identity_nc7(valb+m);
01455
01456                         // compute the inverses of the diagonal sub-blocks
01457                         fasp_smat_inv_nc7(diaginv+i*49);
01458
01459                         // compute D^{-1}*A
01460                         for (k = ibegin+1; k < iend; ++k) {
01461                             m = k*49;
01462                             fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
01463                         }
01464                     }
01465                 } // end of main loop
01466         }
01467         else {
01468             for (i = 0; i < ROW; ++i) {
01469                 // get the diagonal sub-blocks
01470                 ibegin = IA[i]; iend = IA[i+1];
01471                 m = ibegin*49;

```

```

01472         memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
01473         fasp_smat_identity_nc7(valb+m);
01474
01475         // compute the inverses of the diagonal sub-blocks
01476         fasp_smat_inv_nc7(diaginv+i*49);
01477
01478         // compute D^{-1}*A
01479         for (k = ibegin+1; k < iend; ++k) {
01480             m = k*49;
01481             fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
01482         }
01483     } // end of main loop
01484 }
01485
01486     break;
01487
01488     default:
01489         if (use_omp) {
01490 #ifdef _OPENMP
01491 #pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
01492 #endif
01493             for (myid = 0; myid < nthreads; myid++) {
01494                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01495                 for (i = mybegin; i < myend; ++i) {
01496                     // get the diagonal sub-blocks
01497                     ibegin = IA[i]; iend = IA[i+1];
01498                     m = ibegin*nb2;
01499                     memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
01500                     fasp_smat_identity(valb+m, nb, nb2);
01501
01502                     // compute the inverses of the diagonal sub-blocks
01503                     fasp_smat_inv(diaginv+i*nb2, nb);
01504
01505                     // compute D^{-1}*A
01506                     for (k = ibegin+1; k < iend; ++k) {
01507                         m = k*nb2;
01508                         fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
01509                     }
01510                 } // end of main loop
01511             }
01512         }
01513     else {
01514         for (i = 0; i < ROW; ++i) {
01515             // get the diagonal sub-blocks
01516             ibegin = IA[i]; iend = IA[i+1];
01517             m = ibegin*nb2;
01518             memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
01519             fasp_smat_identity(valb+m, nb, nb2);
01520
01521             // compute the inverses of the diagonal sub-blocks
01522             fasp_smat_inv(diaginv+i*nb2, nb);
01523
01524             // compute D^{-1}*A
01525             for (k = ibegin+1; k < iend; ++k) {
01526                 m = k*nb2;
01527                 fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
01528             }
01529         } // end of main loop
01530     }
01531 }
01532
01533     break;
01534 }
01535
01536     return (B);
01537 }
01538
01555 void fasp_dbsr_getdiag (INT          n,
01556                        const dBSRmat *A,
01557                        REAL          *diag )
01558 {
01559     const INT nb2 = A->nb*A->nb;
01560
01561     INT i,k;
01562
01563     if ( n==0 || n>A->ROW || n>A->COL ) n = MIN(A->ROW,A->COL);
01564
01565 #ifdef _OPENMP
01566 #pragma omp parallel for private(i,k) if(n>OPENMP_HOLDS)
01567 #endif
01568     for (i = 0; i < n; ++i) {

```

```

01569         for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
01570             if (A->JA[k] == i) {
01571                 memcpy(diag+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
01572                 break;
01573             }
01574         }
01575     }
01576 }
01577
01593 dBSRmat fasp_dbsr_diagLU (const dBSRmat *A,
01594                          REAL          *DL,
01595                          REAL          *DU)
01596 {
01597     // members of A
01598     const INT ROW = A->ROW;
01599     const INT ROW_plus_one = ROW+1;
01600     const INT COL = A->COL;
01601     const INT NNZ = A->NNZ;
01602     const INT nb = A->nb;
01603
01604     const INT *IA = A->IA;
01605     const INT *JA = A->JA;
01606     const REAL *val = A->val;
01607
01608     INT *IAb = NULL;
01609     INT *JAb = NULL;
01610     REAL *valb = NULL;
01611
01612     INT nb2 = nb*nb;
01613     INT i, j, k;
01614
01615     // Create a dBSRmat 'B'
01616     dBSRmat B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
01617
01618     IAb = B.IA;
01619     JAb = B.JA;
01620     valb = B.val;
01621
01622     fasp_iarray_cp(ROW_plus_one, IA, IAb);
01623     fasp_iarray_cp(NNZ, JA, JAb);
01624
01625     // work array
01626     REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));
01627
01628     // get DL and DU
01629     switch (nb) {
01630     case 2:
01631         for (i=0; i<ROW; i++){
01632             for (j=IA[i]; j<IA[i+1]; j++){
01633                 if (JA[j] == i){
01634                     temp[0] = val[j*nb2];
01635                     temp[1] = val[j*nb2+1];
01636                     temp[2] = val[j*nb2+2];
01637                     temp[3] = val[j*nb2+3];
01638
01639                     // form DL
01640                     DL[i*nb2] = 1.0;
01641                     DL[i*nb2+1] = 0.0;
01642                     DL[i*nb2+2] = -temp[2]/temp[0];
01643                     DL[i*nb2+3] = 1.0;
01644                     //DL[i*nb2+2] = -temp[2]/(temp[0]*s);
01645                     //DL[i*nb2+3] = 1.0/s;
01646
01647                     // form DU
01648                     DU[i*nb2] = 1.0;
01649                     DU[i*nb2+1] = -temp[1]/temp[0];
01650                     DU[i*nb2+2] = 0.0;
01651                     DU[i*nb2+3] = 1.0;
01652
01653                     break;
01654                 } // end of if (JA[j] == i)
01655             } // end of for (j=IA[i]; j<IA[i+1]; j++)
01656         } // end of for (i=0; i<ROW; i++)
01657     }
01658 }

```

```

01665         } // end of for (i=0; i<ROW; i++)
01666
01667         break;
01668
01669     case 3:
01670
01671         for (i=0; i<ROW; i++){
01672
01673             for (j=IA[i]; j<IA[i+1]; j++){
01674
01675                 if (JA[j] == i){
01676
01677                     temp[0] = val[j*nb2];
01678                     temp[1] = val[j*nb2+1];
01679                     temp[2] = val[j*nb2+2];
01680                     temp[3] = val[j*nb2+3];
01681                     temp[4] = val[j*nb2+4];
01682                     temp[5] = val[j*nb2+5];
01683                     temp[6] = val[j*nb2+6];
01684                     temp[7] = val[j*nb2+7];
01685                     temp[8] = val[j*nb2+8];
01686
01687                     // some auxiliary variables
01688                     REAL s22 = temp[4] - ((temp[1]*temp[3])/temp[0]);
01689                     REAL s23 = temp[5] - ((temp[2]*temp[3])/temp[0]);
01690                     REAL s32 = temp[7] - ((temp[1]*temp[6])/temp[0]);
01691
01692                     // form DL
01693                     DL[i*nb2] = 1.0;
01694                     DL[i*nb2+1] = 0.0;
01695                     DL[i*nb2+2] = 0.0;
01696                     DL[i*nb2+3] = -temp[3]/temp[0];
01697                     DL[i*nb2+4] = 1.0;
01698                     DL[i*nb2+5] = 0.0;
01699                     DL[i*nb2+6] = -temp[6]/temp[0] + (temp[3]/temp[0])* (s32/s22);
01700                     DL[i*nb2+7] = -s32/s22;
01701                     DL[i*nb2+8] = 1.0;
01702
01703                     // form DU
01704                     DU[i*nb2] = 1.0;
01705                     DU[i*nb2+1] = -temp[1]/temp[0];
01706                     DU[i*nb2+2] = -temp[2]/temp[0] + (temp[1]/temp[0])* (s23/s22);
01707                     DU[i*nb2+3] = 0.0;
01708                     DU[i*nb2+4] = 1.0;
01709                     DU[i*nb2+5] = -s23/s22;
01710                     DU[i*nb2+6] = 0.0;
01711                     DU[i*nb2+7] = 0.0;
01712                     DU[i*nb2+8] = 1.0;
01713
01714                     break;
01715                 } // end of if (JA[j] == i)
01716
01717             } // end of for (j=IA[i]; j<IA[i+1]; j++)
01718
01719         } // end of for (i=0; i<ROW; i++)
01720
01721         break;
01722
01723     default:
01724         printf("### ERROR: Only works for nb = 2, 3! [%s]\n", __FUNCTION__);
01725         break;
01726
01727 } // end of switch
01728
01729 // compute B = DL*A*DU
01730 switch (nb) {
01731
01732     case 2:
01733
01734         for (i=0; i<ROW; i++){
01735
01736             for (j=IA[i]; j<IA[i+1]; j++){
01737
01738                 k = JA[j];
01739
01740                 // left multiply DL
01741                 fasp_blas_smat_mul_nc2(DL+i*nb2, val+j*nb2, temp);
01742
01743                 // right multiply DU
01744                 fasp_blas_smat_mul_nc2(temp, DU+k*nb2, valb+j*nb2);
01745

```

```

01746
01747         // for diagonal block, set it to be diagonal matrix
01748         if (JA[j] == i){
01749
01750             valb[j*nb2+1] = 0.0;
01751             valb[j*nb2+2] = 0.0;
01752
01753         } // end if (JA[j] == i)
01754
01755
01756         } // end for (j=IA[i]; j<IA[i+1]; j++)
01757
01758     } // end of for (i=0; i<ROW; i++)
01759
01760     break;
01761
01762     case 3:
01763
01764         for (i=0; i<ROW; i++){
01765
01766             for (j=IA[i]; j<IA[i+1]; j++){
01767
01768                 k = JA[j];
01769
01770                 // left multiply DL
01771                 fasp_blas_smat_mul_nc3(DL+i*nb2, val+j*nb2, temp);
01772
01773                 // right multiply DU
01774                 fasp_blas_smat_mul_nc3(temp, DU+k*nb2, valb+j*nb2);
01775
01776                 // for diagonal block, set it to be diagonal matrix
01777                 if (JA[j] == i){
01778
01779                     valb[j*nb2+1] = 0.0;
01780                     valb[j*nb2+2] = 0.0;
01781                     valb[j*nb2+3] = 0.0;
01782                     valb[j*nb2+5] = 0.0;
01783                     valb[j*nb2+6] = 0.0;
01784                     valb[j*nb2+7] = 0.0;
01785                     if (ABS(valb[j*nb2+4]) < SMALLREAL) valb[j*nb2+4] = SMALLREAL;
01786                     if (ABS(valb[j*nb2+8]) < SMALLREAL) valb[j*nb2+8] = SMALLREAL;
01787
01788                 } // end if (JA[j] == i)
01789
01790             } // end for (j=IA[i]; j<IA[i+1]; j++)
01791
01792         } // end of for (i=0; i<ROW; i++)
01793
01794         break;
01795
01796     default:
01797         printf("### ERROR: Only works for nb = 2, 3! [%s]\n", __FUNCTION__);
01798         break;
01799
01800     }
01801
01802     // return
01803     return B;
01804
01805 }
01806
01822 dBSRmat fasp_dbsr_diagLU2 (dBSRmat *A,
01823                             REAL *DL,
01824                             REAL *DU)
01825 {
01826
01827     // members of A
01828     INT ROW = A->ROW;
01829     INT ROW_plus_one = ROW+1;
01830     INT COL = A->COL;
01831     INT NNZ = A->NNZ;
01832     INT nb = A->nb;
01833     INT *IA = A->IA;
01834     INT *JA = A->JA;
01835     REAL *val = A->val;
01836
01837     INT *IAb = NULL;
01838     INT *JAb = NULL;
01839     REAL *valb = NULL;
01840
01841     INT nb2 = nb*nb;

```

```

01842     INT i,j,k;
01843
01844     REAL sqrt3, sqrt4, sqrt8;
01845
01846     // Create a dBSRmat 'B'
01847     dBSRmat B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
01848
01849     REAL *temp = (REAL *)fasp_mem_calloc(nb*nb, sizeof(REAL));
01850
01851     IAb = B.IA;
01852     JAb = B.JA;
01853     valb = B.val;
01854
01855     fasp_iarray_cp(ROW_plus_one, IA, IAb);
01856     fasp_iarray_cp(NNZ, JA, JAb);
01857
01858     // get DL and DU
01859     switch (nb) {
01860     case 2:
01861         for (i=0; i<ROW; i++){
01862             for (j=IA[i]; j<IA[i+1]; j++){
01863                 if (JA[j] == i){
01864                     REAL temp0 = val[j*nb2];
01865                     REAL temp1 = val[j*nb2+1];
01866                     REAL temp2 = val[j*nb2+2];
01867                     REAL temp3 = val[j*nb2+3];
01868
01869                     if (ABS(temp3) < SMALLREAL) temp3 = SMALLREAL;
01870
01871                     sqrt3 = sqrt(ABS(temp3));
01872
01873                     // form DL
01874                     DL[i*nb2] = 1.0;
01875                     DL[i*nb2+1] = 0.0;
01876                     DL[i*nb2+2] = -temp2/temp0/sqrt3;
01877                     DL[i*nb2+3] = 1.0/sqrt3;
01878
01879                     // form DU
01880                     DU[i*nb2] = 1.0;
01881                     DU[i*nb2+1] = -temp1/temp0/sqrt3;
01882                     DU[i*nb2+2] = 0.0;
01883                     DU[i*nb2+3] = 1.0/sqrt3;
01884                     break;
01885                 }
01886             }
01887         }
01888     }
01889
01890     break;
01891
01892     case 3:
01893         for (i=0; i<ROW; i++){
01894             for (j=IA[i]; j<IA[i+1]; j++){
01895                 if (JA[j] == i){
01896                     REAL temp0 = val[j*nb2];
01897                     REAL temp1 = val[j*nb2+1];
01898                     REAL temp2 = val[j*nb2+2];
01899                     REAL temp3 = val[j*nb2+3];
01900                     REAL temp4 = val[j*nb2+4];
01901                     REAL temp5 = val[j*nb2+5];
01902                     REAL temp6 = val[j*nb2+6];
01903                     REAL temp7 = val[j*nb2+7];
01904                     REAL temp8 = val[j*nb2+8];
01905
01906                     if (ABS(temp4) < SMALLREAL) temp4 = SMALLREAL;
01907                     if (ABS(temp8) < SMALLREAL) temp8 = SMALLREAL;
01908
01909                     sqrt4 = sqrt(ABS(temp4));
01910                     sqrt8 = sqrt(ABS(temp8));
01911
01912                     // some auxiliary variables
01913                     REAL s22 = temp4 - ((temp1*temp3)/temp0);
01914                     REAL s23 = temp5 - ((temp2*temp3)/temp0);
01915                     REAL s32 = temp7 - ((temp1*temp6)/temp0);
01916
01917                     // form DL
01918                     DL[i*nb2] = 1.0;
01919                     DL[i*nb2+1] = 0.0;
01920                     DL[i*nb2+2] = 0.0;
01921                     DL[i*nb2+3] = -temp3/temp0/sqrt4;
01922                     DL[i*nb2+4] = 1.0/sqrt4;

```



```

01923         DL[i*nb2+5] = 0.0;
01924         DL[i*nb2+6] = (-temp6/temp0 + (temp3/temp0)*(s32/s22))/sqrt8;
01925         DL[i*nb2+7] = -s32/s22/sqrt8;
01926         DL[i*nb2+8] = 1.0/sqrt8;
01927
01928         // form DU
01929         DU[i*nb2] = 1.0;
01930         DU[i*nb2+1] = -temp1/temp0/sqrt4;
01931         DU[i*nb2+2] = (-temp2/temp0 + (temp1/temp0)*(s23/s22))/sqrt8;
01932         DU[i*nb2+3] = 0.0;
01933         DU[i*nb2+4] = 1.0/sqrt4;
01934         DU[i*nb2+5] = -s23/s22/sqrt8;
01935         DU[i*nb2+6] = 0.0;
01936         DU[i*nb2+7] = 0.0;
01937         DU[i*nb2+8] = 1.0/sqrt8;
01938
01939         break;
01940
01941     }
01942 }
01943 }
01944
01945     break;
01946
01947     default:
01948         printf("### ERROR: Only works for nb = 2, 3! [%s]\n", __FUNCTION__);
01949         break;
01950 } // end of switch
01951
01952 // compute B = DL*A*DU
01953 switch (nb) {
01954     case 2:
01955         for (i=0; i<ROW; i++){
01956             for (j=IA[i]; j<IA[i+1]; j++){
01957                 k = JA[j];
01958                 // left multiply DL
01959                 fasp_blas_smat_mul_nc2(DL+i*nb2, val+j*nb2, temp);
01960                 // right multiply DU
01961                 fasp_blas_smat_mul_nc2(temp, DU+k*nb2, valb[j*nb2]);
01962                 // for diagonal block, set it to be diagonal matrix
01963                 if (JA[j] == i){
01964                     valb[j*nb2+1] = 0.0;
01965                     valb[j*nb2+2] = 0.0;
01966                     if (ABS(valb[j*nb2+3]) < SMALLREAL) valb[j*nb2+3] = SMALLREAL;
01967                 }
01968             }
01969         }
01970     }
01971     break;
01972
01973     case 3:
01974         for (i=0; i<ROW; i++){
01975             for (j=IA[i]; j<IA[i+1]; j++){
01976                 k = JA[j];
01977                 // left multiply DL
01978                 fasp_blas_smat_mul_nc3(DL+i*nb2, val+j*nb2, temp);
01979                 // right multiply DU
01980                 fasp_blas_smat_mul_nc3(temp, DU+k*nb2, valb[j*nb2]);
01981                 // for diagonal block, set it to be diagonal matrix
01982                 if (JA[j] == i){
01983                     valb[j*nb2+1] = 0.0;
01984                     valb[j*nb2+2] = 0.0;
01985                     valb[j*nb2+3] = 0.0;
01986                     valb[j*nb2+5] = 0.0;
01987                     valb[j*nb2+6] = 0.0;
01988                     valb[j*nb2+7] = 0.0;
01989                     if (ABS(valb[j*nb2+4]) < SMALLREAL) valb[j*nb2+4] = SMALLREAL;
01990                     if (ABS(valb[j*nb2+8]) < SMALLREAL) valb[j*nb2+8] = SMALLREAL;
01991                 }
01992             }
01993         }
01994     }
01995     break;
01996
01997     default:
01998         printf("### ERROR: Only works for nb = 2, 3! [%s]\n", __FUNCTION__);
01999         break;
02000 }
02001 // return
02002
02003

```

```

02004     return B;
02005
02006 }
02007
02023 dBSRmat fasp_dbsr_perm (const dBSRmat *A,
02024                          const INT *P)
02025 {
02026     const INT n = A->ROW, nnz = A->NNZ;
02027     const INT *ia = A->IA, *ja = A->JA;
02028     const REAL *Aval = A->val;
02029     const INT nb = A->nb, nb2 = nb*nb;
02030     const INT manner = A->storage_manner;
02031     SHORT nthreads = 1, use_omp = FALSE;
02032
02033     INT i,j,k, jaj,i1,i2,start,jj;
02034
02035 #ifdef _OPENMP
02036     if ( MIN(n, nnz) > OPENMP_HOLDS ) {
02037         use_omp = 0; //TRUE;
02038         nthreads = fasp_get_num_threads();
02039     }
02040 #endif
02041
02042     dBSRmat Aperm = fasp_dbsr_create(n,n,nnz,nb,manner);
02043
02044     // form the transpose of P
02045     INT *Pt = (INT*)fasp_mem_calloc(n,sizeof(INT));
02046
02047     if (use_omp) {
02048         INT myid, mybegin, myend;
02049 #ifdef _OPENMP
02050 #pragma omp parallel for private(myid, mybegin, myend, i)
02051 #endif
02052         for (myid=0; myid<nthreads; ++myid) {
02053             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
02054             for (i=mybegin; i<myend; ++i) Pt[P[i]] = i;
02055         }
02056     }
02057     else {
02058         for (i=0; i<n; ++i) Pt[P[i]] = i;
02059     }
02060
02061     // compute IA of P*A (row permutation)
02062     Aperm.IA[0] = 0;
02063     for (i=0; i<n; ++i) {
02064         k = P[i];
02065         Aperm.IA[i+1] = Aperm.IA[i] + (ia[k+1] - ia[k]);
02066     }
02067
02068     // perform actual P*A
02069     if (use_omp) {
02070         INT myid, mybegin, myend;
02071 #ifdef _OPENMP
02072 #pragma omp parallel for private(myid, mybegin, myend, i, i1, i2, k, start, j, jaj, jj)
02073 #endif
02074         for (myid=0; myid<nthreads; ++myid) {
02075             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
02076             for (i=mybegin; i<myend; ++i) {
02077                 i1 = Aperm.IA[i]; i2 = Aperm.IA[i+1]-1;
02078                 k = P[i];
02079                 start = ia[k];
02080                 for (j=i1; j<=i2; ++j) {
02081                     jaj = start+j-i1;
02082                     Aperm.JA[j] = ja[jaj];
02083                     for (jj=0; jj<nb2; ++jj)
02084                         Aperm.val[j*nb2+jj] = Aval[jaj*nb2+jj];
02085                 }
02086             }
02087         }
02088     }
02089     else {
02090         for (i=0; i<n; ++i) {
02091             i1 = Aperm.IA[i]; i2 = Aperm.IA[i+1]-1;
02092             k = P[i];
02093             start = ia[k];
02094             for (j=i1; j<=i2; ++j) {
02095                 jaj = start+j-i1;
02096                 Aperm.JA[j] = ja[jaj];
02097                 for (jj=0; jj<nb2; ++jj)
02098                     Aperm.val[j*nb2+jj] = Aval[jaj*nb2+jj];
02099             }
02099         }
02099     }

```

```

02100     }
02101 }
02102
02103 // perform P*A*P' (column permutation)
02104 if (use_openmp) {
02105     INT myid, mybegin, myend;
02106 #ifdef _OPENMP
02107 #pragma omp parallel for private(myid, mybegin, myend, k, j)
02108 #endif
02109     for (myid=0; myid<nthreads; ++myid) {
02110         fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
02111         for (k=mybegin; k<myend; ++k) {
02112             j = Aperm.JA[k];
02113             Aperm.JA[k] = Pt[j];
02114         }
02115     }
02116 }
02117 else {
02118     for (k=0; k<nnz; ++k) {
02119         j = Aperm.JA[k];
02120         Aperm.JA[k] = Pt[j];
02121     }
02122 }
02123
02124 fasp_mem_free(Pt); Pt = NULL;
02125
02126 return(Aperm);
02127 }
02128
02141 INT fasp_dbsr_merge_col (dBSRmat *A)
02142 {
02143     INT count = 0;
02144     const INT num_rowsA = A -> ROW;
02145     const INT nb = A->nb;
02146     const INT nb2 = nb*nb;
02147     INT *A_i = A -> IA;
02148     INT *A_j = A -> JA;
02149     REAL *A_data = A -> val;
02150
02151     INT i, ii, j, jj, ibegin, iend, iend1;
02152
02153 #ifdef _OPENMP
02154 // variables for OpenMP
02155     INT myid, mybegin, myend;
02156     INT nthreads = fasp_get_num_threads();
02157 #endif
02158
02159 #ifdef _OPENMP
02160     if (num_rowsA > OPENMP_HOLDS) {
02161 #pragma omp parallel for private (myid,mybegin,myend,i,ii,j,jj,ibegin,iend,iend1)
02162         for (myid = 0; myid < nthreads; myid++) {
02163             fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
02164             for (i = mybegin; i < myend; i++) {
02165                 ibegin = A_i[i]; iend = A_i[i+1]; iend1 = iend-1;
02166                 for (j = ibegin; j < iend1; j++) {
02167                     if (A_j[j] > -1) {
02168                         for (jj = j+1; jj < iend; jj++) {
02169                             if (A_j[jj] == A_j[j]) {
02170                                 // add jj col to j
02171                                 for (ii=0; ii < nb2; ii++)
02172                                     A_data[j*nb2 + ii] += A_data[ jj*nb2+ii];
02173                                 A_j[jj] = -1;
02174                                 count++;
02175                             }
02176                         }
02177                     }
02178                 }
02179             }
02180         }
02181     }
02182 }
02183 #endif
02184 else {
02185     for (i = 0; i < num_rowsA; i++) {
02186         ibegin = A_i[i]; iend = A_i[i+1]; iend1 = iend-1;
02187         for (j = ibegin; j < iend1; j++) {
02188             if (A_j[j] > -1) {
02189                 for (jj = j+1; jj < iend; jj++) {
02190                     if (A_j[jj] == A_j[j]) {
02191                         // add jj col to j
02192                         for (ii=0; ii < nb2; ii++)
02193                             A_data[j*nb2 + ii] += A_data[ jj*nb2+ii];

```

```

02193                                     printf("### WARNING: Same col indices at %d, col %d (%d %d)!\n",
02194                                     i, A_j[j], j, jj );
02195                                     A_j[jj] = -1;
02196                                     count ++;
02197                                     }
02198                                     }
02199                                     }
02200                                     }
02201                                     }
02202 #ifdef __OPENMP
02203 }
02204 #endif
02205
02206     if ( count > 0 ) {
02207         INT *tempA_i = (INT*)fasp_mem_calloc(num_rowsA+1, sizeof(INT));
02208         memcpy(tempA_i, A_i, (num_rowsA+1)*sizeof(INT));
02209         jj = 0; A_i[0] = jj;
02210         for (i = 0; i < num_rowsA; i++) {
02211             ibegin = tempA_i[i]; iend = tempA_i[i+1];
02212             for (j = ibegin; j < iend; j++) {
02213                 if (A_j[j] > -1) {
02214                     memcpy(A_data + jj*nb2, A_data+j*nb2, (nb2)*sizeof(REAL));
02215                     A_j[jj] = A_j[j];
02216                     jj++;
02217                 }
02218             }
02219             A_i[i+1] = jj;
02220         }
02221         A->NNZ = jj;
02222         fasp_mem_free(tempA_i); tempA_i = NULL;
02223
02224         printf("### WARNING: %d col indices have been merged!\n", count);
02225     }
02226
02227     return count;
02228 }
02229
02230 /*-----*/
02231 /*--      End of File      --*/
02232 /*-----*/

```

9.75 BlaSparseCheck.c File Reference

Check properties of sparse matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- [INT fasp_check_diagpos](#) (const [dCSRmat](#) *A)
Check positivity of diagonal entries of a CSR sparse matrix.
- [SHORT fasp_check_diagzero](#) (const [dCSRmat](#) *A)
Check if a CSR sparse matrix has diagonal entries that are very close to zero.
- [INT fasp_check_diagdom](#) (const [dCSRmat](#) *A)
Check whether a matrix is diagonally dominant.
- [INT fasp_check_symm](#) (const [dCSRmat](#) *A)
Check symmetry of a sparse matrix of CSR format.
- void [fasp_check_dCSRmat](#) (const [dCSRmat](#) *A)
Check whether an [dCSRmat](#) matrix is supported or not.
- [SHORT fasp_check_iCSRmat](#) (const [iCSRmat](#) *A)
Check whether an [iCSRmat](#) matrix is valid or not.
- void [fasp_check_ordering](#) ([dCSRmat](#) *A)
Check whether each row of A is in ascending order w.r.t. column indices.

9.75.1 Detailed Description

Check properties of sparse matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), and [BlaSparseCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseCheck.c](#).

9.75.2 Function Documentation

9.75.2.1 fasp_check_dCSRmat()

```
void fasp_check_dCSRmat (
    const dCSRmat * A )
```

Check whether an [dCSRmat](#) matrix is supported or not.

Parameters

A	Pointer to the matrix in dCSRmat format
----------	---

Author

Chensong Zhang

Date

03/29/2009

Definition at line [281](#) of file [BlaSparseCheck.c](#).

9.75.2.2 fasp_check_diagdom()

```
INT fasp_check_diagdom (
    const dCSRmat * A )
```

Check whether a matrix is diagonally dominant.

INT fasp_check_diagdom (const [dCSRmat](#) *A)

Parameters

A	Pointer to the dCSRmat matrix
----------	---

Returns

Number of the rows which are not diagonally dominant

Note

The routine checks whether the sparse matrix is diagonally dominant each row. It will print out the percentage of the rows which are diagonally dominant.

Author

Shuo Zhang

Date

03/29/2009

Definition at line 114 of file [BlaSparseCheck.c](#).

9.75.2.3 fasp_check_diagpos()

```
INT fasp_check_diagpos (
    const dCSRmat * A )
```

Check positivity of diagonal entries of a CSR sparse matrix.

Parameters

<i>A</i>	Pointer to dCSRmat matrix
----------	---

Returns

Number of negative diagonal entries

Author

Shuo Zhang

Date

03/29/2009

Definition at line 35 of file [BlaSparseCheck.c](#).

9.75.2.4 fasp_check_diagzero()

```
SHORT fasp_check_diagzero (
    const dCSRmat * A )
```

Check if a CSR sparse matrix has diagonal entries that are very close to zero.

Parameters

<i>A</i>	pointer to the dCSRmat matrix
----------	---

Returns

FASP_SUCCESS if no diagonal entry is close to zero, else ERROR

Author

Shuo Zhang

Date

03/29/2009

Definition at line 72 of file [BlaSparseCheck.c](#).**9.75.2.5 fasp_check_iCSRmat()**

```
SHORT fasp_check_iCSRmat (
    const iCSRmat * A )
```

Check whether an [iCSRmat](#) matrix is valid or not.

Parameters

<i>A</i>	Pointer to the matrix in iCSRmat format
----------	---

Author

Shuo Zhang

Date

03/29/2009

Definition at line 318 of file [BlaSparseCheck.c](#).**9.75.2.6 fasp_check_ordering()**

```
void fasp_check_ordering (
    dCSRmat * A )
```

Check whether each row of *A* is in ascending order w.r.t. column indices.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
----------	---

Author

Chensong Zhang

Date

02/26/2019

Definition at line 357 of file [BlaSparseCheck.c](#).**9.75.2.7 fasp_check_symm()**

```
INT fasp_check_symm (
    const dCSRmat * A )
```

Check symmetry of a sparse matrix of CSR format.

Parameters

A	Pointer to the dCSRmat matrix
---	---

Returns

1 and 2 if the structure of the matrix is not symmetric; 0 if the structure of the matrix is symmetric,

Note

Print the maximal relative difference between matrix and its transpose.

Author

Shuo Zhang

Date

03/29/2009

Definition at line 159 of file [BlaSparseCheck.c](#).

9.76 BlaSparseCheck.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <math.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_functs.h"
00018
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00035 INT fasp_check_diagpos (const dCSRmat *A)
00036 {
00037     const INT m = A->row;
00038     INT i, num_neg;
00039
00040 #if DEBUG_MODE > 1
00041     printf("### DEBUG: nr = %d, nc = %d, nnz = %d\n", A->row, A->col, A->nnz);
00042 #endif
00043
00044     // store diagonal of A
00045     dvector      diag;
00046     fasp_dcsr_getdiag(m,A,&diag);
00047
00048     // check positiveness of entries of diag
00049     for (num_neg=i=0;i<m;++i) {
00050         if (diag.val[i]<0) num_neg++;
00051     }
00052
00053     printf("Number of negative diagonal entries = %d\n", num_neg);
00054
00055     fasp_dvec_free(&diag);
00056
00057     return num_neg;
00058 }
00059
00072 SHORT fasp_check_diagzero (const dCSRmat *A)
00073 {
00074     const INT    m = A->row;
00075     const INT    *ia = A->IA, *ja = A->JA;
00076     const REAL   *aj = A->val;
00077
00078     SHORT        status = FASP_SUCCESS;
00079     INT          i,j,k,begin_row,end_row;
00080

```

```

00081     for ( i = 0; i < m; ++i ) {
00082         begin_row = ia[i]; end_row = ia[i+1];
00083         for ( k = begin_row; k < end_row; ++k ) {
00084             j = ja[k];
00085             if ( i == j ) {
00086                 if ( ABS(aj[k]) < SMALLREAL ) {
00087                     printf("### ERROR: diag[%d] = %e, close to zero!\n", i, aj[k]);
00088                     status = ERROR_DATA_ZERODIAG;
00089                     goto FINISHED;
00090                 }
00091             }
00092         } // end for k
00093     } // end for i
00094
00095     FINISHED:
00096     return status;
00097 }
00098
00114 INT fasp_check_diagdom (const dCSRmat *A)
00115 {
00116     const INT    nn  = A->row;
00117     const INT    nnz = A->IA[nn]-A->IA[0];
00118     INT          i, j, k;
00119     REAL         sum;
00120
00121     INT *rowp = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00122
00123     for ( i=0; i<nn; ++i ) {
00124         for ( j=A->IA[i]; j<A->IA[i+1]; ++j ) rowp[j]=i;
00125     }
00126
00127     for ( k=0, i=0; i<nn; ++i ) {
00128         sum = 0.0;
00129         for ( j=A->IA[i]; j<A->IA[i+1]; ++j ) {
00130             if ( A->JA[j]==i ) sum += A->val[j];
00131             if ( A->JA[j]!=i ) sum -= fabs(A->val[j]);
00132         }
00133         if ( sum<-SMALLREAL ) ++k;
00134     }
00135
00136     printf("Percentage of the diagonal-dominant rows is %3.2lf%%\n",
00137           100.0*(REAL) (nn-k)/(REAL)nn, "%");
00138
00139     fasp_mem_free(rowp); rowp = NULL;
00140
00141     return k;
00142 }
00143
00159 INT fasp_check_symm (const dCSRmat *A)
00160 {
00161     const REAL symmetry_tol = 1.0e-12;
00162
00163     INT *rowp, *rows[2], *cols[2];
00164     INT i, j, mdi, mdj;
00165     INT nns[2], tnizs[2];
00166     INT type=0;
00167
00168     REAL maxdif, dif;
00169     REAL *vals[2];
00170
00171     const INT nn  = A->row;
00172     const INT nnz = A->IA[nn]-A->IA[0];
00173
00174     if (nnz!=A->nnz) {
00175         printf("### ERROR: nnz=%d, ia[n]-ia[0]=%d, mismatch!\n", A->nnz, nnz);
00176         fasp_chkerz(ERROR_DATA_STRUCTURE, __FUNCTION__);
00177     }
00178
00179     rowp=(INT *)fasp_mem_calloc(nnz, sizeof(INT));
00180
00181     for (i=0; i<nn; ++i) {
00182         for (j=A->IA[i]; j<A->IA[i+1]; ++j) rowp[j]=i;
00183     }
00184
00185     rows[0]=(INT *)fasp_mem_calloc(nnz, sizeof(INT));
00186     cols[0]=(INT *)fasp_mem_calloc(nnz, sizeof(INT));
00187     vals[0]=(REAL *)fasp_mem_calloc(nnz, sizeof(REAL));
00188
00189     memcpy(rows[0], rowp, nnz*sizeof(INT));
00190     memcpy(cols[0], A->JA, nnz*sizeof(INT));
00191     memcpy(vals[0], A->val, nnz*sizeof(REAL));

```

```

00192
00193     nns[0]=nn;
00194     nns[1]=A->col;
00195     tnizs[0]=nnz;
00196
00197     rows[1]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
00198     cols[1]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
00199     vals[1]=(REAL *)fasp_mem_calloc(nnz,sizeof(REAL));
00200
00201     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00202
00203     memcpy(rows[0],rows[1],nnz*sizeof(INT));
00204     memcpy(cols[0],cols[1],nnz*sizeof(INT));
00205     memcpy(vals[0],vals[1],nnz*sizeof(REAL));
00206     nns[0]=A->col;
00207     nns[1]=nn;
00208
00209     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00210
00211     maxdif=0.;
00212     mdi=0;
00213     mdj=0;
00214     for (i=0;i<nnz;++i) {
00215         rows[0][i]=rows[1][i]-rows[0][i];
00216         if (rows[0][i]!=0) {
00217             type=-1;
00218             mdi=rows[0][i];
00219             break;
00220         }
00221
00222         cols[0][i]=cols[1][i]-cols[0][i];
00223         if (cols[0][i]!=0) {
00224             type=-2;
00225             mdj=cols[0][i];
00226             break;
00227         }
00228
00229         if (fabs(vals[0][i])>SMALLREAL||fabs(vals[1][i])>SMALLREAL) {
00230             dif=fabs(vals[1][i]-vals[0][i])/(fabs(vals[0][i])+fabs(vals[1][i]));
00231             if (dif>maxdif) {
00232                 maxdif=dif;
00233                 mdi=rows[0][i];
00234                 mdj=cols[0][i];
00235             }
00236         }
00237     }
00238
00239     if (maxdif>symmetry_tol) type=-3;
00240
00241     switch (type) {
00242     case 0:
00243         printf("Matrix is symmetric with max relative difference is %1.3le\n",maxdif);
00244         break;
00245     case -1:
00246         printf("Matrix has nonsymmetric pattern, check the %d-th, %d-th and %d-th rows and cols\n",
00247             mdi-1,mdi,mdi+1);
00248         break;
00249     case -2:
00250         printf("Matrix has nonsymmetric pattern, check the %d-th, %d-th and %d-th cols and rows\n",
00251             mdj-1,mdj,mdj+1);
00252         break;
00253     case -3:
00254         printf("Matrix is nonsymmetric with max relative difference is %1.3le\n",maxdif);
00255         break;
00256     default:
00257         break;
00258     }
00259
00260     fasp_mem_free(rowp);    rowp    = NULL;
00261     fasp_mem_free(rows[0]); rows[0] = NULL;
00262     fasp_mem_free(rows[1]); rows[1] = NULL;
00263     fasp_mem_free(cols[0]); cols[0] = NULL;
00264     fasp_mem_free(cols[1]); cols[1] = NULL;
00265     fasp_mem_free(vals[0]); vals[0] = NULL;
00266     fasp_mem_free(vals[1]); vals[1] = NULL;
00267
00268     return type;
00269 }
00270
00281 void fasp_check_dCSRmat (const dCSRmat *A)
00282 {

```

```

00283     INT i;
00284
00285     if ( (A->IA == NULL) || (A->JA == NULL) || (A->val == NULL) ) {
00286         printf("### ERROR: Something is wrong with the matrix!\n");
00287         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
00288     }
00289
00290     if ( A->row != A->col ) {
00291         printf("### ERROR: Non-square CSR matrix!\n");
00292         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
00293     }
00294
00295     if ( ( A->nnz <= 0 ) || ( A->row == 0 ) || ( A->col == 0 ) ) {
00296         printf("### ERROR: Empty CSR matrix!\n");
00297         fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00298     }
00299
00300     for ( i = 0; i < A->nnz; ++i ) {
00301         if ( ( A->JA[i] < 0 ) || ( A->JA[i] >= A->col ) ) {
00302             printf("### ERROR: Wrong CSR matrix format!\n");
00303             fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00304         }
00305     }
00306 }
00307
00318 SHORT fasp_check_iCSRmat (const iCSRmat *A)
00319 {
00320     INT i;
00321
00322     if ( (A->IA == NULL) || (A->JA == NULL) || (A->val == NULL) ) {
00323         printf("### ERROR: Something is wrong with the matrix!\n");
00324         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
00325     }
00326
00327     if (A->row != A->col) {
00328         printf("### ERROR: Non-square CSR matrix!\n");
00329         fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00330     }
00331
00332     if ( (A->nnz==0) || (A->row==0) || (A->col==0) ) {
00333         printf("### ERROR: Empty CSR matrix!\n");
00334         fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00335     }
00336
00337     for (i=0;i<A->nnz;++i) {
00338         if ( (A->JA[i]<0) || (A->JA[i]-A->col>=0) ) {
00339             printf("### ERROR: Wrong CSR matrix format!\n");
00340             fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00341         }
00342     }
00343
00344     return FASP_SUCCESS;
00345 }
00346
00357 void fasp_check_ordering (dCSRmat *A)
00358 {
00359     const INT n = A->col;
00360     INT i, j, j1, j2, start, end;
00361
00362     for ( i=0; i<n; ++i ) {
00363
00364         start = A->IA[i];
00365         end = A->IA[i+1] - 1;
00366
00367         for ( j=start; j<end-1; ++j ) {
00368             j1 = A->JA[j]; j2 = A->JA[j + 1];
00369             if ( j1 >= j2 ) {
00370                 printf("### ERROR: Order in row %10d is wrong! %10d, %10d\n", i, j1, j2);
00371                 fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00372             }
00373         }
00374     }
00375 }
00376
00377 }
00378
00379 /*-----*/
00380 /*--      End of File      --*/
00381 /*-----*/

```

9.77 BlaSparseCOO.c File Reference

Sparse matrix operations for [dCOOmat](#) matrices.

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- [dCOOmat fasp_dcoo_create](#) (const [INT](#) m, const [INT](#) n, const [INT](#) nnz)
Create IJ sparse matrix data memory space.
- void [fasp_dcoo_alloc](#) (const [INT](#) m, const [INT](#) n, const [INT](#) nnz, [dCOOmat](#) *A)
Allocate COO sparse matrix memory space.
- void [fasp_dcoo_free](#) ([dCOOmat](#) *A)
Free IJ sparse matrix data memory space.
- void [fasp_dcoo_shift](#) ([dCOOmat](#) *A, const [INT](#) offset)
Re-index a REAL matrix in IJ format to make the index starting from 0 or 1.

9.77.1 Detailed Description

Sparse matrix operations for [dCOOmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#) and [AuxThreads.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseCOO.c](#).

9.77.2 Function Documentation

9.77.2.1 fasp_dcoo_alloc()

```
void fasp_dcoo_alloc (
    const INT m,
    const INT n,
    const INT nnz,
    dCOOmat * A )
```

Allocate COO sparse matrix memory space.

Parameters

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>nnz</i>	Number of nonzeros
<i>A</i>	Pointer to the dCSRmat matrix

Author

Xiaozhe Hu

Date

03/25/2013

Definition at line 70 of file [BlaSparseCOO.c](#).**9.77.2.2 fasp_dcoo_create()**

```
dCOOmat fasp_dcoo_create (
    const INT m,
    const INT n,
    const INT nnz )
```

Create IJ sparse matrix data memory space.

Parameters

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>nnz</i>	Number of nonzeros

ReturnsA The new [dCOOmat](#) matrix**Author**

Chensong Zhang

Date

2010/04/06

Definition at line 42 of file [BlaSparseCOO.c](#).**9.77.2.3 fasp_dcoo_free()**

```
void fasp_dcoo_free (
    dCOOmat * A )
```

Free IJ sparse matrix data memory space.

Parameters

<i>A</i>	Pointer to the dCOOmat matrix
----------	---

Author

Chensong Zhang

Date

2010/04/03

Definition at line 102 of file BlaSparseCOO.c.

9.77.2.4 fasp_dcoo_shift()

```
void fasp_dcoo_shift (
    dCOOmat * A,
    const INT offset )
```

Re-index a REAL matrix in IJ format to make the index starting from 0 or 1.

Parameters

<i>A</i>	Pointer to IJ matrix
<i>offset</i>	Size of offset (1 or -1)

Author

Chensong Zhang

Date

2010/04/06

Modified by Chunsheng Feng, Zheng Li on 08/25/2012

Definition at line 124 of file BlaSparseCOO.c.

9.78 BlaSparseCOO.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <math.h>
00015 #include <time.h>
00016
00017 #ifdef _OPENMP
00018 #include <omp.h>
00019 #endif
00020
00021 #include "fasp.h"
00022 #include "fasp_funcs.h"
00023
00024 /*-----*/
00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00042 dCOOmat fasp_dcoo_create (const INT m,
00043                          const INT n,
00044                          const INT nnz)
00045 {
00046     dCOOmat A;
00047
00048     A.rowind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00049     A.colind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00050     A.val     = (REAL *)fasp_mem_calloc(nnz, sizeof(REAL));
00051
00052     A.row = m; A.col = n; A.nnz = nnz;
00053
00054     return A;
00055 }
00056
00070 void fasp_dcoo_alloc (const INT m,
00071                     const INT n,
00072                     const INT nnz,
```

```

00073             dCOOmat    *A)
00074 {
00075
00076     if ( nnz > 0 ) {
00077         A->rowind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00078         A->colind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00079         A->val     = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
00080     }
00081     else {
00082         A->rowind = NULL;
00083         A->colind = NULL;
00084         A->val     = NULL;
00085     }
00086
00087     A->row = m; A->col = n; A->nnz = nnz;
00088
00089     return;
00090 }
00091
00102 void fasp_dcoo_free (dCOOmat *A)
00103 {
00104     if (A==NULL) return;
00105
00106     fasp_mem_free(A->rowind); A->rowind = NULL;
00107     fasp_mem_free(A->colind); A->colind = NULL;
00108     fasp_mem_free(A->val);    A->val     = NULL;
00109 }
00110
00124 void fasp_dcoo_shift (dCOOmat    *A,
00125                      const INT  offset)
00126 {
00127     const INT nnz = A->nnz;
00128     INT      i, *ai = A->rowind, *aj = A->colind;
00129
00130     // Variables for OpenMP
00131     SHORT nthreads = 1, use_openmp = FALSE;
00132     INT myid, mybegin, myend;
00133
00134     #ifdef _OPENMP
00135     if (nnz > OPENMP_HOLDS) {
00136         use_openmp = TRUE;
00137         nthreads = fasp_get_num_threads();
00138     }
00139     #endif
00140
00141     if (use_openmp) {
00142     #ifdef _OPENMP
00143     #pragma omp parallel for private(myid, i, mybegin, myend)
00144     #endif
00145         for (myid=0; myid<nthreads; myid++) {
00146             fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
00147             for (i=mybegin; i<myend; ++i) {
00148                 ai[i]+=offset; aj[i]+=offset;
00149             }
00150         }
00151     }
00152     else {
00153         for (i=0; i<nnz; ++i) {
00154             ai[i]+=offset; aj[i]+=offset;
00155         }
00156     }
00157 }
00158
00159 /*-----*/
00160 /*--      End of File      --*/
00161 /*-----*/

```

9.79 BlaSparseCSR.c File Reference

Sparse matrix operations for **dCSRmat** matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```


Functions

- `dCSRmat fasp_dcsr_create` (const `INT` m, const `INT` n, const `INT` nnz)
Create CSR sparse matrix data memory space.
- `iCSRmat fasp_icsr_create` (const `INT` m, const `INT` n, const `INT` nnz)
Create CSR sparse matrix data memory space.
- void `fasp_dcsr_alloc` (const `INT` m, const `INT` n, const `INT` nnz, `dCSRmat` *A)
Allocate CSR sparse matrix memory space.
- void `fasp_dcsr_free` (`dCSRmat` *A)
Free CSR sparse matrix data memory space.
- void `fasp_icsr_free` (`iCSRmat` *A)
Free CSR sparse matrix data memory space.
- `INT fasp_dcsr_bandwidth` (const `dCSRmat` *A)
Get bandwidth of matrix.
- `dCSRmat fasp_dcsr_perm` (`dCSRmat` *A, `INT` *P)
Apply permutation of A, i.e. $A_{perm} = PAP'$ by the orders given in P.
- void `fasp_dcsr_sort` (`dCSRmat` *A)
Sort each row of A in ascending order w.r.t. column indices.
- `SHORT fasp_dcsr_getblk` (const `dCSRmat` *A, const `INT` *Is, const `INT` *Js, const `INT` m, const `INT` n, `dCSRmat` *B)
Get a sub CSR matrix of A with specified rows and columns.
- void `fasp_dcsr_getdiag` (`INT` n, const `dCSRmat` *A, `dvector` *diag)
Get first n diagonal entries of a CSR matrix A.
- void `fasp_dcsr_getcol` (const `INT` n, const `dCSRmat` *A, `REAL` *col)
Get the n-th column of a CSR matrix A.
- void `fasp_dcsr_diagpref` (`dCSRmat` *A)
Re-order the column and data arrays of a CSR matrix, so that the first entry in each row is the diagonal.
- `SHORT fasp_dcsr_regdiag` (`dCSRmat` *A, const `REAL` value)
Regularize diagonal entries of a CSR sparse matrix.
- void `fasp_icsr_cp` (const `iCSRmat` *A, `iCSRmat` *B)
Copy a `iCSRmat` to a new one $B=A$.
- void `fasp_dcsr_cp` (const `dCSRmat` *A, `dCSRmat` *B)
copy a `dCSRmat` to a new one $B=A$
- void `fasp_icsr_trans` (const `iCSRmat` *A, `iCSRmat` *AT)
Find transpose of `iCSRmat` matrix A.
- `INT fasp_dcsr_trans` (const `dCSRmat` *A, `dCSRmat` *AT)
Find transpose of `dCSRmat` matrix A.
- void `fasp_dcsr_transpose` (`INT` *row[2], `INT` *col[2], `REAL` *val[2], `INT` *nn, `INT` *tniz)
Transpose of a `dCSRmat` matrix.
- void `fasp_dcsr_compress` (const `dCSRmat` *A, `dCSRmat` *B, const `REAL` dtol)
Compress a CSR matrix A and store in CSR matrix B by dropping small entries $abs(a_{ij}) \leq dtol$.
- `SHORT fasp_dcsr_compress_inplace` (`dCSRmat` *A, const `REAL` dtol)
Compress a CSR matrix A IN PLACE by dropping small entries $abs(a_{ij}) \leq dtol$.
- void `fasp_dcsr_shift` (`dCSRmat` *A, const `INT` offset)
Re-index a `REAL` matrix in CSR format to make the index starting from 0 or 1.
- void `fasp_dcsr_symdiagscale` (`dCSRmat` *A, const `dvector` *diag)
Symmetric diagonal scaling $D^{-1/2} A D^{-1/2}$.
- `dCSRmat fasp_dcsr_sympart` (`dCSRmat` *A)

Get symmetric part of a *dCSRmat* matrix.

- void [fasp_dcsr_transz](#) (*dCSRmat* *A, *INT* *p, *dCSRmat* *AT)

Generalized transpose of A: ($n \times m$) matrix given in *dCSRmat* format.

- *dCSRmat* [fasp_dcsr_permz](#) (*dCSRmat* *A, *INT* *p)

Permute rows and cols of A, i.e. $A=PAP'$ by the ordering in p.

- void [fasp_dcsr_sortz](#) (*dCSRmat* *A, const *SHORT* isym)

Sort each row of A in ascending order w.r.t. column indices.

- void [fasp_dcsr_multicoloring](#) (*dCSRmat* *A, *INT* *flags, *INT* *groups)

Use the greedy multi-coloring to get color groups of the adjacency graph of A.

- void [dCSRmat_Multicoloring](#) (*dCSRmat* *A, *INT* *rowmax, *INT* *groups)

Use the greedy multicoloring algorithm to get color groups for for the adjacency graph of A.

- void [dCSRmat_Multicoloring_Strong_Coupled](#) (*dCSRmat* *A, *iCSRmat* *S, *INT* *flags, *INT* *groups)

Use the greedy multicoloring algorithm to get color groups for the adjacency graph of A.

- void [dCSRmat_Multicoloring_Theta](#) (*dCSRmat* *A, *REAL* theta, *INT* *rowmax, *INT* *groups)

Use the greedy multicoloring algorithm to get color groups for for the adjacency graph of A.

- void [fasp_smoother_dcsr_gs_multicolor](#) (*dvector* *u, *dCSRmat* *A, *dvector* *b, *INT* L, const *INT* order)

9.79.1 Detailed Description

Sparse matrix operations for *dCSRmat* matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxSort.c](#), [AuxThreads.c](#), [AuxVector.c](#), and [BlaSpmvCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseCSR.c](#).

9.79.2 Function Documentation

9.79.2.1 *dCSRmat_Multicoloring*()

```
void dCSRmat_Multicoloring (
    dCSRmat * A,
    INT * rowmax,
    INT * groups )
```

Use the greedy multicoloring algorithm to get color groups for for the adjacency graph of A.

Parameters

<i>A</i>	Input <i>dCSRmat</i>
<i>rowmax</i>	Max row nonzeros of A
<i>groups</i>	Return group numbers

Author

Chunsheng Feng

Date

09/15/2012

Definition at line 1687 of file [BlaSparseCSR.c](#).**9.79.2.2 dCSRmat_Multicoloring_Strong_Coupled()**

```
void dCSRmat_Multicoloring_Strong_Coupled (  
    dCSRmat * A,  
    iCSRmat * S,  
    INT * flags,  
    INT * groups )
```

Use the greedy multicoloring algorithm to get color groups for the adjacency graph of A.

Parameters

<i>A</i>	Input dCSRmat
<i>S</i>	Input iCSRmat Strong Coupled Matrix of A.
<i>flags</i>	Flags for the independent group
<i>groups</i>	Return group numbers

Author

Chunsheng Feng

Date

09/15/2012

Definition at line 1867 of file [BlaSparseCSR.c](#).**9.79.2.3 dCSRmat_Multicoloring_Theta()**

```
void dCSRmat_Multicoloring_Theta (  
    dCSRmat * A,  
    REAL theta,  
    INT * rowmax,  
    INT * groups )
```

Use the greedy multicoloring algorithm to get color groups for for the adjacency graph of A.

Parameters

<i>A</i>	Input dCSRmat
<i>theta</i>	Strength threshold parameter
<i>rowmax</i>	Max row nonzeros of A
<i>groups</i>	Return group numbers

Author

Li Zhao

Date

04/15/2022

Definition at line 1984 of file [BlaSparseCSR.c](#).**9.79.2.4 fasp_dcsr_alloc()**

```
void fasp_dcsr_alloc (
    const INT m,
    const INT n,
    const INT nnz,
    dCSRmat * A )
```

Allocate CSR sparse matrix memory space.

Parameters

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>nnz</i>	Number of nonzeros
<i>A</i>	Pointer to the dCSRmat matrix

Author

Chensong Zhang

Date

2010/04/06

Definition at line 138 of file [BlaSparseCSR.c](#).**9.79.2.5 fasp_dcsr_bandwidth()**

```
INT fasp_dcsr_bandwidth (
    const dCSRmat * A )
```

Get bandwith of matrix.

Parameters

<i>A</i>	pointer to the dCSRmat matrix
----------	---

Author

Zheng Li

Date

03/22/2015

Definition at line 245 of file [BlaSparseCSR.c](#).**9.79.2.6 fasp_dcsr_compress()**

```
void fasp_dcsr_compress (
    const dCSRmat * A,
    dCSRmat * B,
    const REAL dtol )
```

Compress a CSR matrix A and store in CSR matrix B by dropping small entries $\text{abs}(a_{ij}) \leq \text{dtol}$.

Parameters

<i>A</i>	Pointer to dCSRmat CSR matrix
<i>B</i>	Pointer to dCSRmat CSR matrix
<i>dtol</i>	Drop tolerance

Author

Shiquan Zhang

Date

03/10/2010

Modified by Chunsheng Feng, Zheng Li on 08/25/2012

Definition at line 1086 of file [BlaSparseCSR.c](#).**9.79.2.7 fasp_dcsr_compress_inplace()**

```
SHORT fasp_dcsr_compress_inplace (
    dCSRmat * A,
    const REAL dtol )
```

Compress a CSR matrix A IN PLACE by dropping small entries $\text{abs}(a_{ij}) \leq \text{dtol}$.

Parameters

<i>A</i>	Pointer to dCSRmat CSR matrix
<i>dtol</i>	Drop tolerance

Author

Xiaozhe Hu

Date

12/25/2010

Modified by Chensong Zhang on 02/21/2013 Modified by Chunsheng Feng on 10/16/2020: Avoid filtering diagonal entries.

Note

This routine can be modified for filtering.

Definition at line 1166 of file [BlaSparseCSR.c](#).

9.79.2.8 fasp_dcsr_cp()

```
void fasp_dcsr_cp (
    const dCSRmat * A,
    dCSRmat * B )
```

copy a [dCSRmat](#) to a new one B=A

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>B</i>	Pointer to the dCSRmat matrix

Author

Chensong Zhang

Date

04/06/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 851 of file [BlaSparseCSR.c](#).

9.79.2.9 fasp_dcsr_create()

```
dCSRmat fasp_dcsr_create (
    const INT m,
    const INT n,
    const INT nnz )
```

Create CSR sparse matrix data memory space.

Parameters

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>nnz</i>	Number of nonzeros

Returns

A the new [dCSRmat](#) matrix

Author

Chensong Zhang

Date

2010/04/06

Definition at line 47 of file [BlaSparseCSR.c](#).**9.79.2.10 fasp_dcsr_diagpref()**

```
void fasp_dcsr_diagpref (  
    dCSRmat * A )
```

Re-order the column and data arrays of a CSR matrix, so that the first entry in each row is the diagonal.

Parameters

<i>A</i>	Pointer to the matrix to be re-ordered
----------	--

Author

Zhiyang Zhou

Date

09/09/2010

Author

Chunsheng Feng, Zheng Li

Date

09/02/2012

Note

Reordering is done in place.

Modified by Chensong Zhang on Dec/21/2012

Definition at line 680 of file [BlaSparseCSR.c](#).**9.79.2.11 fasp_dcsr_free()**

```
void fasp_dcsr_free (  
    dCSRmat * A )
```

Free CSR sparse matrix data memory space.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
----------	---

Author

Chensong Zhang

Date

2010/04/06 Modified by Chunsheng Feng on 08/11/2017: init A to NULL

Definition at line 184 of file [BlaSparseCSR.c](#).

9.79.2.12 fasp_dcsr_getblk()

```
SHORT fasp_dcsr_getblk (
    const dCSRmat * A,
    const INT * Is,
    const INT * Js,
    const INT m,
    const INT n,
    dCSRmat * B )
```

Get a sub CSR matrix of A with specified rows and columns.

Parameters

<i>A</i>	Pointer to dCSRmat matrix
<i>B</i>	Pointer to dCSRmat matrix
<i>Is</i>	Pointer to selected rows
<i>Js</i>	Pointer to selected columns
<i>m</i>	Number of selected rows
<i>n</i>	Number of selected columns

Returns

FASP_SUCCESS if succeeded, otherwise return error information.

Author

Shiquan Zhang, Xiaozhe Hu

Date

12/25/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 446 of file [BlaSparseCSR.c](#).

9.79.2.13 fasp_dcsr_getcol()

```
void fasp_dcsr_getcol (
    const INT n,
    const dCSRmat * A,
    REAL * col )
```

Get the n-th column of a CSR matrix A.

Parameters

<i>n</i>	Index of a column of A ($0 \leq n \leq A.col-1$)
<i>A</i>	Pointer to dCSRmat CSR matrix
<i>col</i>	Pointer to the column

Author

Xiaozhe Hu

Date

11/07/2009

Modified by Chunsheng Feng, Zheng Li on 07/08/2012

Definition at line 602 of file [BlaSparseCSR.c](#).**9.79.2.14 fasp_dcsr_getdiag()**

```
void fasp_dcsr_getdiag (
    INT n,
    const dCSRmat * A,
    dvector * diag )
```

Get first n diagonal entries of a CSR matrix A.

Parameters

<i>n</i>	Number of diagonal entries to get (if n=0, then get all diagonal entries)
<i>A</i>	Pointer to dCSRmat CSR matrix
<i>diag</i>	Pointer to the diagonal as a dvector

Author

Chensong Zhang

Date

05/20/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 537 of file [BlaSparseCSR.c](#).**9.79.2.15 fasp_dcsr_multicoloring()**

```
void fasp_dcsr_multicoloring (
    dCSRmat * A,
    INT * flags,
    INT * groups )
```

Use the greedy multi-coloring to get color groups of the adjacency graph of A.

Parameters

<i>A</i>	Input dCSRmat
<i>flags</i>	flags for the independent group
<i>groups</i>	Return group numbers

Author

Chunsheng Feng

Date

09/15/2012

Definition at line 1602 of file [BlaSparseCSR.c](#).**9.79.2.16 fasp_dcsr_perm()**

```
dCSRmat fasp_dcsr_perm (
    dCSRmat * A,
    INT * P )
```

Apply permutation of A, i.e. $A_{perm}=PAP'$ by the orders given in P.**Parameters**

<i>A</i>	Pointer to the original dCSRmat matrix
<i>P</i>	Pointer to orders

ReturnsThe new ordered [dCSRmat](#) matrix if succeed, NULL if fail**Author**

Shiquan Zhang

Date

03/10/2010

Note $P[i] = k$ means k-th row and column become i-th row and column!Deprecated! Will be replaced by `fasp_dcsr_permz` later. –Chensong

Modified by Chunsheng Feng, Zheng Li on 07/12/2012

Definition at line 275 of file [BlaSparseCSR.c](#).**9.79.2.17 fasp_dcsr_permz()**

```
dCSRmat fasp_dcsr_permz (
    dCSRmat * A,
    INT * p )
```

Permute rows and cols of A, i.e. $A=PAP'$ by the ordering in p.**Parameters**

<i>A</i>	Pointer to the original dCSRmat matrix
<i>p</i>	Pointer to ordering

Note

This is just applying twice `fasp_dcsr_transz(&A,p,At)`.

In matlab notation: `Aperm=A(p,p)`;

Returns

The new ordered [dCSRmat](#) matrix if succeed, NULL if fail

Author

Ludmil Zikatanov

Date

19951219 (Fortran), 20150912 (C)

Definition at line [1540](#) of file [BlaSparseCSR.c](#).

9.79.2.18 fasp_dcsr_regdiag()

```
SHORT fasp_dcsr_regdiag (  
    dCSRmat * A,  
    const REAL value )
```

Regularize diagonal entries of a CSR sparse matrix.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>value</i>	Set a value on <code>diag(A)</code> which is too close to zero to "value"

Returns

FASP_SUCCESS if no diagonal entry is close to zero, else ERROR

Author

Shiquan Zhang

Date

11/07/2009

Definition at line [786](#) of file [BlaSparseCSR.c](#).

9.79.2.19 fasp_dcsr_shift()

```
void fasp_dcsr_shift (  
    dCSRmat * A,  
    const INT offset )
```

Re-index a REAL matrix in CSR format to make the index starting from 0 or 1.

Parameters

<i>A</i>	Pointer to CSR matrix
<i>offset</i>	Size of offset (1 or -1)

Author

Chensong Zhang

Date

04/06/2010

Modified by Chunsheng Feng, Zheng Li on 07/11/2012

Definition at line [1212](#) of file [BlaSparseCSR.c](#).

9.79.2.20 fasp_dcsr_sort()

```
void fasp_dcsr_sort (
    dCSRmat * A )
```

Sort each row of A in ascending order w.r.t. column indices.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
----------	---

Author

Shiquan Zhang

Date

06/10/2010

Definition at line [385](#) of file [BlaSparseCSR.c](#).

9.79.2.21 fasp_dcsr_sortz()

```
void fasp_dcsr_sortz (
    dCSRmat * A,
    const SHORT isym )
```

Sort each row of A in ascending order w.r.t. column indices.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>isym</i>	Flag for symmetry, =[0/nonzero]=[general/symmetric] matrix

Note

Applying twice [fasp_dcsr_transz\(\)](#), if A is symmetric, then the transpose is applied only once and then AT copied on A.

Author

Ludmil Zikatanov

Date

19951219 (Fortran), 20150912 (C)

Definition at line 1571 of file [BlaSparseCSR.c](#).**9.79.2.22 fasp_dcsr_symdiagscale()**

```
void fasp_dcsr_symdiagscale (
    dCSRmat * A,
    const dvector * diag )
```

Symmetric diagonal scaling $D^{-1/2}AD^{-1/2}$.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>diag</i>	Pointer to the diagonal entries

Author

Xiaozhe Hu

Date

01/31/2011

Modified by Chunsheng Feng, Zheng Li on 07/11/2012

Definition at line 1270 of file [BlaSparseCSR.c](#).**9.79.2.23 fasp_dcsr_sympart()**

```
dCSRmat fasp_dcsr_sympart (
    dCSRmat * A )
```

Get symmetric part of a [dCSRmat](#) matrix.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
----------	---

Returns

Symmetrized the [dCSRmat](#) matrix

Author

Xiaozhe Hu

Date

03/21/2011

Definition at line 1357 of file [BlaSparseCSR.c](#).**9.79.2.24 fasp_dcsr_trans()**

```
void fasp_dcsr_trans (
    const dCSRmat * A,
    dCSRmat * AT )
```

Find transpose of [dCSRmat](#) matrix A.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix
<i>AT</i>	Pointer to the transpose of dCSRmat matrix A (output)

Author

Chensong Zhang

Date

04/06/2010

Modified by Chunsheng Feng, Zheng Li on 06/20/2012

Definition at line 952 of file [BlaSparseCSR.c](#).**9.79.2.25 fasp_dcsr_transpose()**

```
void fasp_dcsr_transpose (
    INT * row[2],
    INT * col[2],
    REAL * val[2],
    INT * nn,
    INT * tniz )
```

Transpose of a [dCSRmat](#) matrix.

Note

This subroutine transpose in CSR format IN ORDER

Parameters

<i>row</i>	Pointers of the rows of the matrix and its transpose
<i>col</i>	Pointers of the columns of the matrix and its transpose
<i>val</i>	Pointers to the values of the matrix and its transpose
<i>nn</i>	Pointer to the number of rows/columns of A and A'
<i>tniz</i>	Pointer to the number of nonzeros A and A'

Author

Shuo Zhang

Date

07/06/2009

Definition at line 1037 of file [BlaSparseCSR.c](#).**9.79.2.26 fasp_dcsr_transz()**

```
void fasp_dcsr_transz (
    dCSRmat * A,
    INT * p,
    dCSRmat * AT )
```

Generalized transpose of A: (n x m) matrix given in [dCSRmat](#) format.

Parameters

<i>A</i>	Pointer to matrix in dCSRmat for transpose, INPUT
<i>p</i>	Permutation, INPUT
<i>AT</i>	Pointer to matrix AT = transpose(A) if p = NULL, OR AT = transpose(A)p if p is not NULL

Note

The storage for all pointers in AT should already be allocated, i.e. AT->IA, AT->JA and AT->val should be allocated before calling this function. If A.val=NULL, then AT->val[] is not changed.

performs $AT = \text{transpose}(A)p$, where p is a permutation. If p=NULL then p=I is assumed. Applying twice this procedure one gets $At = \text{transpose}(\text{transpose}(A)p)p = \text{transpose}(p)Ap$, which is the same A with rows and columns permuted according to p.

If A=NULL, then only transposes/permutes the structure of A.

For p=NULL, applying this two times $A \rightarrow AT \rightarrow A$ orders all the row indices in A in increasing order.

Reference: Fred G. Gustavson. Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Software, 4(3):250-269, 1978.

Author

Ludmil Zikatanov

Date

19951219 (Fortran), 20150912 (C)

Definition at line 1416 of file [BlaSparseCSR.c](#).**9.79.2.27 fasp_icsr_cp()**

```
void fasp_icsr_cp (
    const iCSRmat * A,
    iCSRmat * B )
```

Copy a [iCSRmat](#) to a new one B=A.

Parameters

<i>A</i>	Pointer to the iCSRmat matrix
<i>B</i>	Pointer to the iCSRmat matrix

Author

Chensong Zhang

Date

05/16/2013

Definition at line [827](#) of file [BlaSparseCSR.c](#).

9.79.2.28 fasp_icsr_create()

```
iCSRmat fasp_icsr_create (
    const INT m,
    const INT n,
    const INT nnz )
```

Create CSR sparse matrix data memory space.

Parameters

<i>m</i>	Number of rows
<i>n</i>	Number of columns
<i>nnz</i>	Number of nonzeros

Returns

A the new [iCSRmat](#) matrix

Author

Chensong Zhang

Date

2010/04/06

Definition at line [96](#) of file [BlaSparseCSR.c](#).

9.79.2.29 fasp_icsr_free()

```
void fasp_icsr_free (
    iCSRmat * A )
```

Free CSR sparse matrix data memory space.

Parameters

<i>A</i>	Pointer to the iCSRmat matrix
----------	---

Author

Chensong Zhang

Date

2010/04/06 Modified by Chunsheng Feng on 08/11/2017: init A to NULL

Definition at line 219 of file [BlaSparseCSR.c](#).**9.79.2.30 fasp_icsr_trans()**

```
void fasp_icsr_trans (
    const iCSRmat * A,
    iCSRmat * AT )
```

Find transpose of iCSRmat matrix A.

Parameters

<i>A</i>	Pointer to the iCSRmat matrix A
<i>AT</i>	Pointer to the iCSRmat matrix A'

Author

Chensong Zhang

Date

04/06/2010

Modified by Chunsheng Feng, Zheng Li on 06/20/2012

Definition at line 875 of file [BlaSparseCSR.c](#).**9.79.2.31 fasp_smoother_dcsr_gs_multicolor()**

```
void fasp_smoother_dcsr_gs_multicolor (
    dvector * u,
    dCSRmat * A,
    dvector * b,
    INT L,
    const INT order )
```

Definition at line 2123 of file [BlaSparseCSR.c](#).**9.80 BlaSparseCSR.c**[Go to the documentation of this file.](#)

```
00001
00015 #include <math.h>
00016 #include <time.h>
00017
00018 #ifdef _OPENMP
00019 #include <omp.h>
00020 #endif
00021
00022 #include "fasp.h"
00023 #include "fasp_functs.h"
```

```

00024
00025 #if MULTI_COLOR_ORDER
00026 static void generate_S_theta(dCSRmat*, iCSRmat*, REAL);
00027 #endif
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00047 dCSRmat fasp_dcsr_create(const INT m, const INT n, const INT nnz)
00048 {
00049     dCSRmat A;
00050
00051     if (m > 0) {
00052         A.IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00053     } else {
00054         A.IA = NULL;
00055     }
00056
00057     if (n > 0) {
00058         A.JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00059     } else {
00060         A.JA = NULL;
00061     }
00062
00063     if (nnz > 0) {
00064         A.val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
00065     } else {
00066         A.val = NULL;
00067     }
00068
00069     A.row = m;
00070     A.col = n;
00071     A.nnz = nnz;
00072
00073 #if MULTI_COLOR_ORDER
00074     A.color = 0;
00075     A.IC = NULL;
00076     A.ICMAP = NULL;
00077 #endif
00078
00079     return A;
00080 }
00081
00096 iCSRmat fasp_icsr_create(const INT m, const INT n, const INT nnz)
00097 {
00098     iCSRmat A;
00099
00100     if (m > 0) {
00101         A.IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00102     } else {
00103         A.IA = NULL;
00104     }
00105
00106     if (n > 0) {
00107         A.JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00108     } else {
00109         A.JA = NULL;
00110     }
00111
00112     if (nnz > 0) {
00113         A.val = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00114     } else {
00115         A.val = NULL;
00116     }
00117
00118     A.row = m;
00119     A.col = n;
00120     A.nnz = nnz;
00121
00122     return A;
00123 }
00124
00138 void fasp_dcsr_alloc(const INT m, const INT n, const INT nnz, dCSRmat* A)
00139 {
00140     if (m <= 0 || n <= 0) {
00141         printf("### ERROR: Matrix dim %d, %d must be positive! [%s]\n", m, n,
00142             __FUNCTION__);
00143         return;
00144     }
00145

```

```

00146     if (m > 0) {
00147         A->IA = (INT*) fasp_mem_calloc(m + 1, sizeof(INT));
00148     } else {
00149         A->IA = NULL;
00150     }
00151
00152     if (nnz > 0) {
00153         A->JA = (INT*) fasp_mem_calloc(nnz, sizeof(INT));
00154         A->val = (REAL*) fasp_mem_calloc(nnz, sizeof(REAL));
00155     } else {
00156         A->JA = NULL;
00157         A->val = NULL;
00158     }
00159
00160     A->row = m;
00161     A->col = n;
00162     A->nnz = nnz;
00163
00164     #if MULTI_COLOR_ORDER
00165     A->color = 0;
00166     A->IC = NULL;
00167     A->ICMAP = NULL;
00168     #endif
00169
00170     return;
00171 }
00172
00184 void fasp_dcsr_free(dCSRmat* A)
00185 {
00186     if (A == NULL) return;
00187
00188     fasp_mem_free(A->IA);
00189     A->IA = NULL;
00190     fasp_mem_free(A->JA);
00191     A->JA = NULL;
00192     fasp_mem_free(A->val);
00193     A->val = NULL;
00194
00195     #if MULTI_COLOR_ORDER
00196     fasp_mem_free(A->IC);
00197     A->IC = NULL;
00198     fasp_mem_free(A->ICMAP);
00199     A->ICMAP = NULL;
00200     #endif
00201
00202     A->col = 0;
00203     A->row = 0;
00204     A->nnz = 0;
00205     A = NULL;
00206 }
00207
00219 void fasp_icsr_free(iCSRmat* A)
00220 {
00221     if (A == NULL) return;
00222
00223     fasp_mem_free(A->IA);
00224     A->IA = NULL;
00225     fasp_mem_free(A->JA);
00226     A->JA = NULL;
00227     fasp_mem_free(A->val);
00228     A->val = NULL;
00229     A->col = 0;
00230     A->row = 0;
00231     A->nnz = 0;
00232     A = NULL;
00233 }
00234
00245 INT fasp_dcsr_bandwidth(const dCSRmat* A)
00246 {
00247     const INT row = A->row;
00248     const INT* ia = A->IA;
00249     INT i, max;
00250
00251     for (max = i = 0; i < row; ++i) max = MAX(max, ia[i + 1] - ia[i]);
00252
00253     return (max);
00254 }
00255
00275 dCSRmat fasp_dcsr_perm(dCSRmat* A, INT* P)
00276 {
00277     const INT n = A->row, nnz = A->nnz;

```

```

00278     const INT * ia = A->IA, *ja = A->JA;
00279     const REAL* Aval = A->val;
00280     INT      i, j, k, jaj, i1, i2, start;
00281     SHORT    nthreads = 1, use_omp = FALSE;
00282
00283 #ifdef _OPENMP
00284     if (MIN(n, nnz) > OPENMP_HOLDS) {
00285         use_omp = TRUE;
00286         nthreads = fasp_get_num_threads();
00287     }
00288 #endif
00289
00290     dCSRmat Aperm = fasp_dcsr_create(n, n, nnz);
00291
00292     // form the transpose of P
00293     INT* Pt = (INT*)fasp_mem_calloc(n, sizeof(INT));
00294
00295     if (use_omp) {
00296         INT myid, mybegin, myend;
00297 #ifdef _OPENMP
00298         #pragma omp parallel for private(myid, mybegin, myend, i)
00299 #endif
00300         for (myid = 0; myid < nthreads; ++myid) {
00301             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00302             for (i = mybegin; i < myend; ++i) Pt[P[i]] = i;
00303         }
00304     } else {
00305         for (i = 0; i < n; ++i) Pt[P[i]] = i;
00306     }
00307
00308     // compute IA of P*A (row permutation)
00309     Aperm.IA[0] = 0;
00310     for (i = 0; i < n; ++i) {
00311         k = P[i];
00312         Aperm.IA[i + 1] = Aperm.IA[i] + (ia[k + 1] - ia[k]);
00313     }
00314
00315     // perform actual P*A
00316     if (use_omp) {
00317         INT myid, mybegin, myend;
00318 #ifdef _OPENMP
00319         #pragma omp parallel for private(myid, mybegin, myend, i1, i2, k, start, j, jaj)
00320 #endif
00321         for (myid = 0; myid < nthreads; ++myid) {
00322             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00323             for (i = mybegin; i < myend; ++i) {
00324                 i1 = Aperm.IA[i];
00325                 i2 = Aperm.IA[i + 1] - 1;
00326                 k = P[i];
00327                 start = ia[k];
00328                 for (j = i1; j <= i2; ++j) {
00329                     jaj = start + j - i1;
00330                     Aperm.JA[j] = ja[jaj];
00331                     Aperm.val[j] = Aval[jaj];
00332                 }
00333             }
00334         }
00335     } else {
00336         for (i = 0; i < n; ++i) {
00337             i1 = Aperm.IA[i];
00338             i2 = Aperm.IA[i + 1] - 1;
00339             k = P[i];
00340             start = ia[k];
00341             for (j = i1; j <= i2; ++j) {
00342                 jaj = start + j - i1;
00343                 Aperm.JA[j] = ja[jaj];
00344                 Aperm.val[j] = Aval[jaj];
00345             }
00346         }
00347     }
00348
00349     // perform P*A*P' (column permutation)
00350     if (use_omp) {
00351         INT myid, mybegin, myend;
00352 #ifdef _OPENMP
00353         #pragma omp parallel for private(myid, mybegin, myend, k, j)
00354 #endif
00355         for (myid = 0; myid < nthreads; ++myid) {
00356             fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
00357             for (k = mybegin; k < myend; ++k) {
00358                 j = Aperm.JA[k];

```

```

00359         Aperm.JA[k] = Pt[j];
00360     }
00361 }
00362 } else {
00363     for (k = 0; k < nnz; ++k) {
00364         j = Aperm.JA[k];
00365         Aperm.JA[k] = Pt[j];
00366     }
00367 }
00368
00369 fasp_mem_free(Pt);
00370 Pt = NULL;
00371
00372 return (Aperm);
00373 }
00374
00385 void fasp_dcsr_sort(dCSRmat* A)
00386 {
00387     const INT n = A->col;
00388     INT i, j, start, row_length;
00389
00390     // temp memory for sorting rows of A
00391     INT * index, *ja;
00392     REAL* a;
00393
00394     index = (INT*)fasp_mem_malloc(n, sizeof(INT));
00395     ja = (INT*)fasp_mem_malloc(n, sizeof(INT));
00396     a = (REAL*)fasp_mem_malloc(n, sizeof(REAL));
00397
00398     for (i = 0; i < n; ++i) {
00399         start = A->IA[i];
00400         row_length = A->IA[i + 1] - start;
00401
00402         for (j = 0; j < row_length; ++j) index[j] = j;
00403
00404         fasp_aux_iQuickSortIndex(&(A->JA[start]), 0, row_length - 1, index);
00405
00406         for (j = 0; j < row_length; ++j) {
00407             ja[j] = A->JA[start + index[j]];
00408             a[j] = A->val[start + index[j]];
00409         }
00410
00411         for (j = 0; j < row_length; ++j) {
00412             A->JA[start + j] = ja[j];
00413             A->val[start + j] = a[j];
00414         }
00415     }
00416
00417     // clean up memory
00418     fasp_mem_free(index);
00419     index = NULL;
00420     fasp_mem_free(ja);
00421     ja = NULL;
00422     fasp_mem_free(a);
00423     a = NULL;
00424 }
00425
00446 SHORT fasp_dcsr_getblk(const dCSRmat* A, const INT* Is, const INT* Js, const INT m,
00447                       const INT n, dCSRmat* B)
00448 {
00449     SHORT use_omp = FALSE;
00450     SHORT status = FASP_SUCCESS;
00451     INT i, j, k, nnz = 0;
00452     INT* col_flag;
00453
00454     #ifdef _OPENMP
00455     INT stride_i, mybegin, myend, myid, nthreads;
00456     if (n > OPENMP_HOLDS) {
00457         use_omp = TRUE;
00458         nthreads = fasp_get_num_threads();
00459     }
00460     #endif
00461
00462     // create column flags
00463     col_flag = (INT*)fasp_mem_malloc(A->col, sizeof(INT));
00464
00465     B->row = m;
00466     B->col = n;
00467
00468     B->IA = (INT*)fasp_mem_malloc(m + 1, sizeof(INT));
00469     B->JA = (INT*)fasp_mem_malloc(A->nnz, sizeof(INT));

```

```

00470     B->val = (REAL*) fasp_mem_calloc(A->nnz, sizeof(REAL));
00471
00472 #if MULTI_COLOR_ORDER
00473     B->color = 0;
00474     B->IC = NULL;
00475     B->ICMAP = NULL;
00476 #endif
00477
00478     if (use_omp) {
00479 #ifdef _OPENMP
00480         stride_i = n / nthreads;
00481 #pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
00482         {
00483             myid = omp_get_thread_num();
00484             mybegin = myid * stride_i;
00485             if (myid < nthreads - 1)
00486                 myend = mybegin + stride_i;
00487             else
00488                 myend = n;
00489             for (i = mybegin; i < myend; ++i) {
00490                 col_flag[Js[i]] = i + 1;
00491             }
00492         }
00493 #endif
00494     } else {
00495         for (i = 0; i < n; ++i) col_flag[Js[i]] = i + 1;
00496     }
00497
00498     // Count nonzeros for sub matrix and fill in
00499     B->IA[0] = 0;
00500     for (i = 0; i < m; ++i) {
00501         for (k = A->IA[Is[i]]; k < A->IA[Is[i] + 1]; ++k) {
00502             j = A->JA[k];
00503             if (col_flag[j] > 0) {
00504                 B->JA[nnz] = col_flag[j] - 1;
00505                 B->val[nnz] = A->val[k];
00506                 nnz++;
00507             }
00508         } /* end for k */
00509         B->IA[i + 1] = nnz;
00510     } /* end for i */
00511     B->nnz = nnz;
00512
00513     // re-allocate memory space
00514     B->JA = (INT*) fasp_mem_realloc(B->JA, sizeof(INT) * nnz);
00515     B->val = (REAL*) fasp_mem_realloc(B->val, sizeof(REAL) * nnz);
00516
00517     fasp_mem_free(col_flag);
00518     col_flag = NULL;
00519
00520     return (status);
00521 }
00522
00537 void fasp_dcsr_getdiag(INT n, const dCSRmat* A, dvector* diag)
00538 {
00539     INT i, k, j, ibegin, iend;
00540
00541     SHORT nthreads = 1, use_omp = FALSE;
00542
00543     if (n == 0 || n > A->row || n > A->col) n = MIN(A->row, A->col);
00544
00545 #ifdef _OPENMP
00546     if (n > OPENMP_HOLDS) {
00547         use_omp = TRUE;
00548         nthreads = fasp_get_num_threads();
00549     }
00550 #endif
00551
00552     fasp_dvec_alloc(n, diag);
00553
00554     if (use_omp) {
00555         INT mybegin, myend, myid;
00556 #ifdef _OPENMP
00557 #pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, k, j)
00558 #endif
00559         for (myid = 0; myid < nthreads; myid++) {
00560             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00561             for (i = mybegin; i < myend; i++) {
00562                 ibegin = A->IA[i];
00563                 iend = A->IA[i + 1];
00564                 for (k = ibegin; k < iend; ++k) {

```

```

00565         j = A->JA[k];
00566         if ((j - i) == 0) {
00567             diag->val[i] = A->val[k];
00568             break;
00569         } // end if
00570     } // end for k
00571 } // end for i
00572 }
00573 } else {
00574     for (i = 0; i < n; ++i) {
00575         ibegin = A->IA[i];
00576         iend = A->IA[i + 1];
00577         for (k = ibegin; k < iend; ++k) {
00578             j = A->JA[k];
00579             if ((j - i) == 0) {
00580                 diag->val[i] = A->val[k];
00581                 break;
00582             } // end if
00583         } // end for k
00584     } // end for i
00585 }
00586 }
00587
00602 void fasp_dcsr_getcol(const INT n, const dCSRmat* A, REAL* col)
00603 {
00604     INT i, j, row_begin, row_end;
00605     INT nrow = A->row, ncol = A->col;
00606     INT status = FASP_SUCCESS;
00607
00608     SHORT nthreads = 1, use_omp = FALSE;
00609
00610 #ifdef _OPENMP
00611     if (nrow > OPENMP_HOLDS) {
00612         use_omp = TRUE;
00613         nthreads = fasp_get_num_threads();
00614     }
00615 #endif
00616
00617     // check the column index n
00618     if (n < 0 || n >= ncol) {
00619         printf("### ERROR: Illegal column index %d! [%s]\n", n, __FUNCTION__);
00620         status = ERROR_DUMMY_VAR;
00621         goto FINISHED;
00622     }
00623
00624     // get the column
00625     if (use_omp) {
00626         INT mybegin, myend, myid;
00627
00628 #ifdef _OPENMP
00629 #pragma omp parallel for private(myid, mybegin, myend, i, j, row_begin, row_end)
00630 #endif
00631         for (myid = 0; myid < nthreads; myid++) {
00632             fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00633             for (i = mybegin; i < myend; i++) {
00634                 col[i] = 0.0;
00635                 row_begin = A->IA[i];
00636                 row_end = A->IA[i + 1];
00637                 for (j = row_begin; j < row_end; ++j) {
00638                     if (A->JA[j] == n) {
00639                         col[i] = A->val[j];
00640                     }
00641                 } // end for j
00642             } // end for i
00643         }
00644     } else {
00645         for (i = 0; i < nrow; ++i) {
00646             // set the entry to zero
00647             col[i] = 0.0;
00648             row_begin = A->IA[i];
00649             row_end = A->IA[i + 1];
00650             for (j = row_begin; j < row_end; ++j) {
00651                 if (A->JA[j] == n) {
00652                     col[i] = A->val[j];
00653                 }
00654             } // end for j
00655         } // end for i
00656     }
00657
00658 FINISHED:
00659     fasp_chkerr(status, __FUNCTION__);

```

```

00660 }
00661
00680 void fasp_dcsr_diagpref(dCSRmat* A)
00681 {
00682     const INT num_rowsA = A->row;
00683     REAL* A_data = A->val;
00684     INT* A_i = A->IA;
00685     INT* A_j = A->JA;
00686
00687     // Local variable
00688     INT i, j;
00689     INT tempi, row_size;
00690     REAL tempd;
00691
00692 #ifdef _OPENMP
00693     // variables for OpenMP
00694     INT myid, mybegin, myend, ibegin, iend;
00695     INT nthreads = fasp_get_num_threads();
00696 #endif
00697
00698 #if DEBUG_MODE > 0
00699     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00700 #endif
00701
00702 #ifdef _OPENMP
00703     if (num_rowsA > OPENMP_HOLDS) {
00704 #pragma omp parallel for private(myid, i, j, ibegin, iend, tempi, tempd, mybegin, myend)
00705         for (myid = 0; myid < nthreads; myid++) {
00706             fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
00707             for (i = mybegin; i < myend; i++) {
00708                 ibegin = A_i[i];
00709                 iend = A_i[i + 1];
00710                 // check whether the first entry is already diagonal
00711                 if (A_j[ibegin] != i) {
00712                     for (j = ibegin + 1; j < iend; j++) {
00713                         if (A_j[j] == i) {
00714 #if DEBUG_MODE > 2
00715                             printf("### DEBUG: Switch entry_%d with entry_0\n", j);
00716 #endif
00717                             tempi = A_j[ibegin];
00718                             A_j[ibegin] = A_j[j];
00719                             A_j[j] = tempi;
00720
00721                             tempd = A_data[ibegin];
00722                             A_data[ibegin] = A_data[j];
00723                             A_data[j] = tempd;
00724                             break;
00725                         }
00726                     }
00727                     if (j == iend) {
00728                         printf("### ERROR: Diagonal entry %d is zero!\n", i);
00729                         fasp_chkerr(ERROR_MISC, __FUNCTION__);
00730                     }
00731                 }
00732             }
00733         }
00734     } else {
00735 #endif
00736         for (i = 0; i < num_rowsA; i++) {
00737             row_size = A_i[i + 1] - A_i[i];
00738             // check whether the first entry is already diagonal
00739             if (A_j[0] != i) {
00740                 for (j = 1; j < row_size; j++) {
00741                     if (A_j[j] == i) {
00742 #if DEBUG_MODE > 2
00743                         printf("### DEBUG: Switch entry_%d with entry_0\n", j);
00744 #endif
00745                         tempi = A_j[0];
00746                         A_j[0] = A_j[j];
00747                         A_j[j] = tempi;
00748
00749                         tempd = A_data[0];
00750                         A_data[0] = A_data[j];
00751                         A_data[j] = tempd;
00752
00753                         break;
00754                     }
00755                 }
00756                 if (j == row_size) {
00757                     printf("### ERROR: Diagonal entry %d is zero!\n", i);
00758                     fasp_chkerr(ERROR_MISC, __FUNCTION__);

```



```

00759         }
00760     }
00761     A_j += row_size;
00762     A_data += row_size;
00763 }
00764 #ifdef _OPENMP
00765 }
00766 #endif
00767
00768 #if DEBUG_MODE > 0
00769     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00770 #endif
00771 }
00772
00786 SHORT fasp_dcsr_regdiag(dCSRmat* A, const REAL value)
00787 {
00788     const INT m = A->row;
00789     const INT *ia = A->IA, *ja = A->JA;
00790     REAL* aj = A->val;
00791
00792     // Local variables
00793     INT i, j, k, begin_row, end_row;
00794     SHORT status = ERROR_UNKNOWN;
00795
00796     for (i = 0; i < m; ++i) {
00797         begin_row = ia[i];
00798         end_row = ia[i + 1];
00799         for (k = begin_row; k < end_row; ++k) {
00800             j = ja[k];
00801             if (i == j) {
00802                 if (aj[k] < 0.0)
00803                     goto FINISHED;
00804                 else if (aj[k] < SMALLREAL)
00805                     aj[k] = value;
00806             }
00807         } // end for k
00808     } // end for i
00809
00810     status = FASP_SUCCESS;
00811
00812 FINISHED:
00813     return status;
00814 }
00815
00827 void fasp_icsr_cp(const iCSRmat* A, iCSRmat* B)
00828 {
00829     B->row = A->row;
00830     B->col = A->col;
00831     B->nnz = A->nnz;
00832
00833     fasp_iarray_cp(A->row + 1, A->IA, B->IA);
00834     fasp_iarray_cp(A->nnz, A->JA, B->JA);
00835     fasp_iarray_cp(A->nnz, A->val, B->val);
00836 }
00837
00851 void fasp_dcsr_cp(const dCSRmat* A, dCSRmat* B)
00852 {
00853     B->row = A->row;
00854     B->col = A->col;
00855     B->nnz = A->nnz;
00856
00857     fasp_iarray_cp(A->row + 1, A->IA, B->IA);
00858     fasp_iarray_cp(A->nnz, A->JA, B->JA);
00859     fasp_darray_cp(A->nnz, A->val, B->val);
00860 }
00861
00875 void fasp_icsr_trans(const iCSRmat* A, iCSRmat* AT)
00876 {
00877     const INT n = A->row, m = A->col, nnz = A->nnz, ml = m - 1;
00878
00879     // Local variables
00880     INT i, j, k, p;
00881     INT ibegin, iend;
00882
00883     #if DEBUG_MODE > 1
00884         printf("### DEBUG: m=%d, n=%d, nnz=%d\n", m, n, nnz);
00885     #endif
00886
00887     AT->row = m;
00888     AT->col = n;
00889     AT->nnz = nnz;

```

```

00890
00891     AT->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00892
00893     AT->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00894
00895     if (A->val) {
00896         AT->val = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00897     } else {
00898         AT->val = NULL;
00899     }
00900
00901     // first pass: find the Number of nonzeros in the first m-1 columns of A
00902     // Note: these Numbers are stored in the array AT.IA from 1 to m-1
00903     fasp_iarray_set(m + 1, AT->IA, 0);
00904
00905     for (j = 0; j < nnz; ++j) {
00906         i = A->JA[j]; // column Number of A = row Number of A'
00907         if (i < m1) AT->IA[i + 2]++;
00908     }
00909
00910     for (i = 2; i <= m; ++i) AT->IA[i] += AT->IA[i - 1];
00911
00912     // second pass: form A'
00913     if (A->val != NULL) {
00914         for (i = 0; i < n; ++i) {
00915             ibegin = A->IA[i];
00916             iend = A->IA[i + 1];
00917             for (p = ibegin; p < iend; p++) {
00918                 j = A->JA[p] + 1;
00919                 k = AT->IA[j];
00920                 AT->JA[k] = i;
00921                 AT->val[k] = A->val[p];
00922                 AT->IA[j] = k + 1;
00923             } // end for p
00924         } // end for i
00925     } else {
00926         for (i = 0; i < n; ++i) {
00927             ibegin = A->IA[i];
00928             iend = A->IA[i + 1];
00929             for (p = ibegin; p < iend; p++) {
00930                 j = A->JA[p] + 1;
00931                 k = AT->IA[j];
00932                 AT->JA[k] = i;
00933                 AT->IA[j] = k + 1;
00934             } // end for p
00935         } // end for i
00936     } // end if
00937 }
00938
00952 INT fasp_dcsr_trans(const dCSRmat* A, dCSRmat* AT)
00953 {
00954     const INT n = A->row, m = A->col, nnz = A->nnz;
00955
00956     // Local variables
00957     INT i, j, k, p;
00958
00959     AT->row = m;
00960     AT->col = n;
00961     AT->nnz = nnz;
00962
00963     AT->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
00964
00965     AT->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
00966
00967     if (A->val) {
00968         AT->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
00969     } else {
00970         AT->val = NULL;
00971     }
00972 }
00973
00974 #if MULTI_COLOR_ORDER
00975     AT->color = 0;
00976     AT->IC = NULL;
00977     AT->ICMAP = NULL;
00978 #endif
00979
00980     // first pass: find the Number of nonzeros in the first m-1 columns of A
00981     // Note: these Numbers are stored in the array AT.IA from 1 to m-1
00982
00983     // fasp_iarray_set(m+1, AT->IA, 0);

```

```

00984     memset(AT->IA, 0, sizeof(INT) * (m + 1));
00985
00986     for (j = 0; j < nnz; ++j) {
00987         i = A->JA[j]; // column Number of A = row Number of A'
00988         if (i < m - 1) AT->IA[i + 2]++;
00989     }
00990
00991     for (i = 2; i <= m; ++i) AT->IA[i] += AT->IA[i - 1];
00992
00993     // second pass: form A'
00994     if (A->val) {
00995         for (i = 0; i < n; ++i) {
00996             INT ibegin = A->IA[i], iend = A->IA[i + 1];
00997             for (p = ibegin; p < iend; p++) {
00998                 j = A->JA[p] + 1;
00999                 k = AT->IA[j];
01000                 AT->JA[k] = i;
01001                 AT->val[k] = A->val[p];
01002                 AT->IA[j] = k + 1;
01003             } // end for p
01004         } // end for i
01005     } else {
01006         for (i = 0; i < n; ++i) {
01007             INT ibegin = A->IA[i], iendl = A->IA[i + 1];
01008             for (p = ibegin; p < iendl; p++) {
01009                 j = A->JA[p] + 1;
01010                 k = AT->IA[j];
01011                 AT->JA[k] = i;
01012                 AT->IA[j] = k + 1;
01013             } // end for p
01014         } // end of i
01015     } // end if
01016
01017     return FASP_SUCCESS;
01018 }
01019
01037 void fasp_dcsr_transpose(INT* row[2], INT* col[2], REAL* val[2], INT* nn, INT* tniz)
01038 {
01039     const INT nca = nn[1]; // Number of columns
01040
01041     INT* izc = (INT*)fasp_mem_calloc(nn[1], sizeof(INT));
01042     INT* izcaux = (INT*)fasp_mem_calloc(nn[1], sizeof(INT));
01043
01044     // Local variables
01045     INT i, m, itmp;
01046
01047     // first pass: to set order right
01048     for (i = 0; i < tniz[0]; ++i) izc[col[0][i]]++;
01049
01050     izcaux[0] = 0;
01051     for (i = 1; i < nca; ++i) izcaux[i] = izcaux[i - 1] + izc[i - 1];
01052
01053     // second pass: form transpose
01054     memset(izc, 0, nca * sizeof(INT));
01055
01056     for (i = 0; i < tniz[0]; ++i) {
01057         m = col[0][i];
01058         itmp = izcaux[m] + izc[m];
01059         row[1][itmp] = m;
01060         col[1][itmp] = row[0][i];
01061         val[1][itmp] = val[0][i];
01062         izc[m]++;
01063     }
01064
01065     fasp_mem_free(izc);
01066     izc = NULL;
01067     fasp_mem_free(izcaux);
01068     izcaux = NULL;
01069 }
01070
01086 void fasp_dcsr_compress(const dCSRmat* A, dCSRmat* B, const REAL dtol)
01087 {
01088     INT i, j, k;
01089     INT ibegin, iendl;
01090
01091     SHORT nthreads = 1, use_omp = FALSE;
01092
01093     #ifdef _OPENMP
01094     if (B->nnz > OPENMP_HOLDS) {
01095         use_omp = TRUE;
01096         nthreads = fasp_get_num_threads();
01097     }
01098     #endif

```

```

01097     }
01098 #endif
01099
01100     INT* index = (INT*)fasp_mem_calloc(A->nnz, sizeof(INT));
01101
01102     B->row = A->row;
01103     B->col = A->col;
01104
01105     B->IA = (INT*)fasp_mem_calloc(A->row + 1, sizeof(INT));
01106
01107     B->IA[0] = A->IA[0];
01108
01109     // first pass: determine the size of B
01110     k = 0;
01111     for (i = 0; i < A->row; ++i) {
01112         ibegin = A->IA[i];
01113         iend1 = A->IA[i + 1];
01114         for (j = ibegin; j < iend1; ++j)
01115             if (ABS(A->val[j]) > dtol) {
01116                 index[k] = j;
01117                 ++k;
01118             } /* end of j */
01119         B->IA[i + 1] = k;
01120     } /* end of i */
01121     B->nnz = k;
01122     B->JA = (INT*)fasp_mem_calloc(B->nnz, sizeof(INT));
01123     B->val = (REAL*)fasp_mem_calloc(B->nnz, sizeof(REAL));
01124
01125     // second pass: generate the index and element to B
01126     if (use_omp) {
01127         INT myid, mybegin, myend;
01128 #ifdef _OPENMP
01129 #pragma omp parallel for private(myid, i, mybegin, myend)
01130 #endif
01131         for (myid = 0; myid < nthreads; myid++) {
01132             fasp_get_start_end(myid, nthreads, B->nnz, &mybegin, &myend);
01133             for (i = mybegin; i < myend; ++i) {
01134                 B->JA[i] = A->JA[index[i]];
01135                 B->val[i] = A->val[index[i]];
01136             }
01137         }
01138     } else {
01139         for (i = 0; i < B->nnz; ++i) {
01140             B->JA[i] = A->JA[index[i]];
01141             B->val[i] = A->val[index[i]];
01142         }
01143     }
01144
01145     fasp_mem_free(index);
01146     index = NULL;
01147 }
01148
01166 SHORT fasp_dcsr_compress_inplace(dCSRmat* A, const REAL dtol)
01167 {
01168     const INT row = A->row;
01169     const INT nnz = A->nnz;
01170
01171     INT i, j, k;
01172     INT ibegin, iend = A->IA[0];
01173     SHORT status = FASP_SUCCESS;
01174     k = 0;
01175     for (i = 0; i < row; ++i) {
01176         ibegin = iend;
01177         iend = A->IA[i + 1];
01178         for (j = ibegin; j < iend; ++j)
01179             if (ABS(A->val[j]) > dtol || i == A->JA[j]) {
01180                 A->JA[k] = A->JA[j];
01181                 A->val[k] = A->val[j];
01182                 ++k;
01183             } /* end of j */
01184         A->IA[i + 1] = k;
01185     } /* end of i */
01186
01187     if (k <= nnz) {
01188         A->nnz = k;
01189         A->JA = (INT*)fasp_mem_realloc(A->JA, k * sizeof(INT));
01190         A->val = (REAL*)fasp_mem_realloc(A->val, k * sizeof(REAL));
01191     } else {
01192         printf("### WARNING: Size of compressed matrix is bigger than original!\n");
01193         status = ERROR_UNKNOWN;
01194     }

```

```

01195
01196     return (status);
01197 }
01198
01212 void fasp_dcsr_shift(dCSRmat* A, const INT offset)
01213 {
01214     const INT nnz = A->nnz;
01215     const INT n   = A->row + 1;
01216     INT      i, *ai = A->IA, *aj = A->JA;
01217     SHORT    nthreads = 1, use_openmp = FALSE;
01218
01219 #ifdef _OPENMP
01220     if (MIN(n, nnz) > OPENMP_HOLDS) {
01221         use_openmp = TRUE;
01222         nthreads   = fasp_get_num_threads();
01223     }
01224 #endif
01225
01226     if (use_openmp) {
01227         INT myid, mybegin, myend;
01228 #ifdef _OPENMP
01229 #pragma omp parallel for private(myid, mybegin, myend, i)
01230 #endif
01231         for (myid = 0; myid < nthreads; myid++) {
01232             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
01233             for (i = mybegin; i < myend; i++) {
01234                 ai[i] += offset;
01235             }
01236         }
01237     } else {
01238         for (i = 0; i < n; ++i) ai[i] += offset;
01239     }
01240
01241     if (use_openmp) {
01242         INT myid, mybegin, myend;
01243 #ifdef _OPENMP
01244 #pragma omp parallel for private(myid, mybegin, myend, i)
01245 #endif
01246         for (myid = 0; myid < nthreads; myid++) {
01247             fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
01248             for (i = mybegin; i < myend; i++) {
01249                 aj[i] += offset;
01250             }
01251         }
01252     } else {
01253         for (i = 0; i < nnz; ++i) aj[i] += offset;
01254     }
01255 }
01256
01270 void fasp_dcsr_symdiagscale(dCSRmat* A, const dvector* diag)
01271 {
01272     // information about matrix A
01273     const INT n   = A->row;
01274     const INT* IA  = A->IA;
01275     const INT* JA  = A->JA;
01276     REAL*     val  = A->val;
01277     REAL*     work;
01278
01279     SHORT nthreads = 1, use_openmp = FALSE;
01280
01281     // local variables
01282     INT i, j, k, row_start, row_end;
01283
01284 #ifdef _OPENMP
01285     if (n > OPENMP_HOLDS) {
01286         use_openmp = TRUE;
01287         nthreads   = fasp_get_num_threads();
01288     }
01289 #endif
01290
01291     if (diag->row != n) {
01292         printf("### ERROR: Size of diag = %d != size of matrix = %d!", diag->row, n);
01293         fasp_chkerr(ERROR_MISC, __FUNCTION__);
01294     }
01295
01296     // work space
01297     work = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
01298
01299     if (use_openmp) {
01300         INT myid, mybegin, myend;
01301 #ifdef _OPENMP

```

```

01302 #pragma omp parallel for private(myid, mybegin, myend, i)
01303 #endif
01304     for (myid = 0; myid < nthreads; myid++) {
01305         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
01306         for (i = mybegin; i < myend; i++) work[i] = sqrt(diag->val[i]);
01307     }
01308 } else {
01309     // square root of diagonal entries
01310     for (i = 0; i < n; i++) work[i] = sqrt(diag->val[i]);
01311 }
01312
01313 if (use_openmp) {
01314     INT myid, mybegin, myend;
01315 #ifdef _OPENMP
01316 #pragma omp parallel for private(myid, mybegin, myend, row_start, row_end, i, j, k)
01317 #endif
01318     for (myid = 0; myid < nthreads; myid++) {
01319         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
01320         for (i = mybegin; i < myend; i++) {
01321             row_start = IA[i];
01322             row_end = IA[i + 1];
01323             for (j = row_start; j < row_end; j++) {
01324                 k = JA[j];
01325                 val[j] = val[j] / (work[i] * work[k]);
01326             }
01327         }
01328     }
01329 } else {
01330     // main loop
01331     for (i = 0; i < n; i++) {
01332         row_start = IA[i];
01333         row_end = IA[i + 1];
01334         for (j = row_start; j < row_end; j++) {
01335             k = JA[j];
01336             val[j] = val[j] / (work[i] * work[k]);
01337         }
01338     }
01339 }
01340
01341 // free work space
01342 fasp_mem_free(work);
01343 work = NULL;
01344 }
01345
01357 dCSRmat fasp_dcsr_sympart(dCSRmat* A)
01358 {
01359     // local variable
01360     dCSRmat AT;
01361
01362     // return variable
01363     dCSRmat SA;
01364
01365 #if MULTI_COLOR_ORDER
01366     AT.IC = NULL;
01367     SA.IC = NULL;
01368     AT.ICMAP = NULL;
01369     SA.ICMAP = NULL;
01370 #endif
01371
01372     // get the transpose of A
01373     fasp_dcsr_trans(A, &AT);
01374
01375     // get symmetrized A
01376     fasp_blas_dcsr_add(A, 1.0, &AT, 0.0, &SA);
01377
01378     // clean
01379     fasp_dcsr_free(&AT);
01380
01381     // return
01382     return SA;
01383 }
01384
01416 void fasp_dcsr_transz(dCSRmat* A, INT* p, dCSRmat* AT)
01417 {
01418     /* tested for permutation and transposition */
01419     /* transpose or permute; if A.val is null ==> transpose the
01420 structure only */
01421     const INT n = A->row, m = A->col, nnz = A->nnz;
01422     const INT * ia = NULL, *ja = NULL;
01423     const REAL* a = NULL;
01424     INT ml = m + 1;

```

```

01425     ia             = A->IA;
01426     ja             = A->JA;
01427     a              = A->val;
01428     /* introducing few extra pointers should not hurt too much the speed */
01429     INT * iat = NULL, *jat = NULL;
01430     REAL* at = NULL;
01431
01432     /* loop variables */
01433     INT i, j, jp, pi, iabeg, iaend, k;
01434
01435     /* initialize */
01436     AT->row = m;
01437     AT->col = n;
01438     AT->nnz = nnz;
01439
01440     /* all these should be allocated or change this to allocate them here */
01441     iat = AT->IA;
01442     jat = AT->JA;
01443     at  = AT->val;
01444     for (i = 0; i < m1; ++i) iat[i] = 0;
01445     iaend = ia[n];
01446     for (i = 0; i < iaend; ++i) {
01447         j = ja[i] + 2;
01448         if (j < m1) iat[j]++;
01449     }
01450     iat[0] = 0;
01451     iat[1] = 0;
01452     if (m != 1) {
01453         for (i = 2; i < m1; ++i) {
01454             iat[i] += iat[i - 1];
01455         }
01456     }
01457
01458     if (p && a) {
01459         /* so we permute and also use matrix entries */
01460         for (i = 0; i < n; ++i) {
01461             pi = p[i];
01462             iabeg = ia[pi];
01463             iaend = ia[pi + 1];
01464             if (iaend > iabeg) {
01465                 for (jp = iabeg; jp < iaend; ++jp) {
01466                     j = ja[jp] + 1;
01467                     k = iat[j];
01468                     jat[k] = i;
01469                     at[k] = a[jp];
01470                     iat[j] = k + 1;
01471                 }
01472             }
01473         }
01474     } else if (a && !p) {
01475         /* transpose values, no permutation */
01476         for (i = 0; i < n; ++i) {
01477             iabeg = ia[i];
01478             iaend = ia[i + 1];
01479             if (iaend > iabeg) {
01480                 for (jp = iabeg; jp < iaend; ++jp) {
01481                     j = ja[jp] + 1;
01482                     k = iat[j];
01483                     jat[k] = i;
01484                     at[k] = a[jp];
01485                     iat[j] = k + 1;
01486                 }
01487             }
01488         }
01489     } else if (!a && p) {
01490         /* Only integers and permutation (only a is null) */
01491         for (i = 0; i < n; ++i) {
01492             pi = p[i];
01493             iabeg = ia[pi];
01494             iaend = ia[pi + 1];
01495             if (iaend > iabeg) {
01496                 for (jp = iabeg; jp < iaend; ++jp) {
01497                     j = ja[jp] + 1;
01498                     k = iat[j];
01499                     jat[k] = i;
01500                     iat[j] = k + 1;
01501                 }
01502             }
01503         }
01504     } else {
01505         /* Only integers and no permutation (both a and p are null) */

```

```

01506         for (i = 0; i < n; ++i) {
01507             iabeg = ia[i];
01508             iaend = ia[i + 1];
01509             if (iaend > iabeg) {
01510                 for (jp = iabeg; jp < iaend; ++jp) {
01511                     j = ja[jp] + 1;
01512                     k = iat[j];
01513                     jat[k] = i;
01514                     iat[j] = k + 1;
01515                 }
01516             }
01517         }
01518     }
01519     return;
01520 }
01521 }
01522
01540 dCSRmat fasp_dcsr_permz(dCSRmat* A, INT* p)
01541 {
01542     const INT n = A->row, nnz = A->nnz;
01543     dCSRmat Aperml, Aperm;
01544
01545     Aperml = fasp_dcsr_create(n, n, nnz);
01546     Aperm = fasp_dcsr_create(n, n, nnz);
01547
01548     fasp_dcsr_transz(A, p, &Aperml);
01549     fasp_dcsr_transz(&Aperml, p, &Aperm);
01550
01551     // clean up
01552     fasp_dcsr_free(&Aperml);
01553
01554     return (Aperm);
01555 }
01556
01571 void fasp_dcsr_sortz(dCSRmat* A, const SHORT isym)
01572 {
01573     const INT n = A->row, m = A->col, nnz = A->nnz;
01574     dCSRmat AT = fasp_dcsr_create(m, n, nnz);
01575
01576     /* watch carefully who is a pointer and who is not in fasp_dcsr_transz() */
01577     fasp_dcsr_transz(A, NULL, &AT);
01578
01579     /* if the matrix is symmetric, then only one transpose is needed
01580 and now we just copy */
01581     if ((m == n) && (isym))
01582         fasp_dcsr_cp(&AT, A);
01583     else
01584         fasp_dcsr_transz(&AT, NULL, A);
01585
01586     // clean up
01587     fasp_dcsr_free(&AT);
01588 }
01589
01602 void fasp_dcsr_multicoloring(dCSRmat* A, INT* flags, INT* groups)
01603 {
01604     #if MULTI_COLOR_ORDER
01605         INT k, i, j, pre, group;
01606         INT iend;
01607         INT icount;
01608         INT front, rear;
01609         INT n = A->row;
01610         INT* IA = A->IA;
01611         INT* JA = A->JA;
01612         INT* cq = (INT*)malloc(sizeof(INT) * (n + 1));
01613         INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));
01614
01615         #ifdef _OPENMP
01616         #pragma omp parallel for private(k)
01617         #endif
01618         for (k = 0; k < n; k++) cq[k] = k;
01619
01620         group = 0;
01621         for (k = 0; k < n; k++) {
01622             if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
01623         }
01624
01625         A->IC = (INT*)malloc(sizeof(INT) * (group + 2));
01626         A->ICMAP = (INT*)malloc(sizeof(INT) * (n));
01627
01628         front = n - 1;
01629         rear = n - 1;

```



```

01630
01631     memset(newr, -1, sizeof(INT) * (n + 1));
01632     memset(A->ICMAP, 0, sizeof(INT) * n);
01633
01634     group    = 0;
01635     icount    = 0;
01636     A->IC[0] = 0;
01637     pre      = 0;
01638
01639     do {
01640         front++;
01641         if (front == n) front = 0;
01642         i = cq[front];
01643         if (i <= pre) {
01644             A->IC[group] = icount;
01645             A->ICMAP[icount] = i;
01646             group++;
01647             icount++;
01648             iend = IA[i + 1];
01649             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01650         } else if (newr[i] == group) {
01651             rear++;
01652             if (rear == n) rear = 0;
01653             cq[rear] = i;
01654         } else {
01655             A->ICMAP[icount] = i;
01656             icount++;
01657             iend = IA[i + 1];
01658             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01659         }
01660         pre = i;
01661     } while (rear != front);
01662
01663     A->IC[group] = icount;
01664     A->color    = group;
01665     free(cq);
01666     free(newr);
01667     *groups = group;
01668 #else
01669     printf("### ERROR: %s has not been defined!\n", __FUNCTION__);
01670 #endif
01671 }
01672
01673
01687 void dCSRmat_Multicoloring(dCSRmat* A, INT* rowmax, INT* groups)
01688 {
01689     #if MULTI_COLOR_ORDER
01690         INT k, i, j, pre, group;
01691         INT igold, iend, iavg;
01692         INT icount;
01693         INT front, rear;
01694         INT n = A->row;
01695         INT* IA = A->IA;
01696         INT* JA = A->JA;
01697
01698         INT* cq = (INT*)malloc(sizeof(INT) * (n + 1));
01699         INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));
01700
01701         for (k = 0; k < n; k++) cq[k] = k;
01702
01703         group = 0;
01704
01705         for (k = 0; k < n; k++) {
01706             if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
01707         }
01708         *rowmax = group;
01709         #if 0
01710             iavg = IA[n]/n ;
01711             igold = (INT)MAX(iavg, group*0.618) +1;
01712             igold =group ;
01713         #endif
01714
01715         A->IC = (INT*)malloc(sizeof(INT) * (group + 2));
01716         A->ICMAP = (INT*)malloc(sizeof(INT) * (n));
01717
01718         front = n - 1;
01719         rear = n - 1;
01720
01721         memset(newr, -1, sizeof(INT) * (n + 1));
01722         memset(A->ICMAP, 0, sizeof(INT) * n);
01723

```

```

01724     group    = 0;
01725     icount   = 0;
01726     A->IC[0] = 0;
01727     pre      = 0;
01728
01729     do {
01730         // front = (front+1)%n;
01731         front++;
01732         if (front == n) front = 0; // front = front < n ? front : 0 ;
01733         i = cq[front];
01734
01735         if (i <= pre) {
01736             A->IC[group] = icount;
01737             A->ICMAP[icount] = i;
01738             group++;
01739             icount++;
01740             iend = IA[i + 1];
01741             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01742         } else if (newr[i] == group) {
01743             // rear = (rear + 1)%n;
01744             rear++;
01745             if (rear == n) rear = 0;
01746             cq[rear] = i;
01747         } else {
01748             A->ICMAP[icount] = i;
01749             icount++;
01750             iend = IA[i + 1];
01751             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01752         }
01753         pre = i;
01754     } while (rear != front);
01755
01756     A->IC[group] = icount;
01757     A->color    = group;
01758
01759     #if 0
01760     for(i=0; i < A->color; i++) {
01761         for(j=A -> IC[i]; j < A-> IC[i+1]; j++)
01762             printf("color %d ICMAP[%d] = %d \n", i, j, A-> ICMAP[j]);
01763         printf( "A.color = %d A.row= %d %d\n", A -> color, A -> row, A-> IC[i+1] - A-> IC[i] );
01764         getchar();
01765     }
01766     #endif
01767
01768     // printf(" Max Row Numbers %d avg %d igold %d max %d %d\n", group, iavg, igold,
01769     // (INT)MAX(iavg, group*0.618), A->IA[n]/n );
01770     free(cq);
01771     free(newr);
01772     *groups = group;
01773 #endif
01774 }
01775
01776 #if MULTI_COLOR_ORDER
01777 static void generate_S_theta(dCSRmat* A, iCSRmat* S, REAL theta)
01778 {
01779     const INT row = A->row, col = A->col;
01780     const INT row_plus_one = row + 1;
01781     const INT nnz          = A->IA[row] - A->IA[0];
01782
01783     INT index, i, j, begin_row, end_row;
01784     INT * ia = A->IA, *ja = A->JA;
01785     REAL* aj = A->val;
01786
01787     // get the diagonal entry of A
01788     // dvector diag; fasp_dcsr_getdiag(0, A, &diag);
01789
01790     /* generate S */
01791     REAL row_abs_sum;
01792
01793     // copy the structure of A to S
01794     S->row = row;
01795     S->col = col;
01796     S->nnz = nnz;
01797     S->val = NULL;
01798
01799     S->IA = (INT*)fasp_mem_calloc(row_plus_one, sizeof(INT));
01800
01801     S->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
01802
01803     fasp_iarray_cp(row_plus_one, ia, S->IA);
01804     fasp_iarray_cp(nnz, ja, S->JA);

```

```

01805
01806 #ifndef _OPENMP
01807 #pragma omp parallel for private(i, j, begin_row, end_row, row_abs_sum)
01808 #endif
01809     for (i = 0; i < row; ++i) {
01810         /* compute scaling factor and row sum */
01811         row_abs_sum = 0;
01812         begin_row = ia[i];
01813         end_row = ia[i + 1];
01814         for (j = begin_row; j < end_row; j++) {
01815             row_abs_sum += ABS(aj[j]);
01816         }
01817         row_abs_sum = row_abs_sum * theta;
01818
01819         /* deal with the element of S */
01820         for (j = begin_row; j < end_row; j++) {
01821             if ((row_abs_sum >= ABS(aj[j])) && (ja[j] != i)) {
01822                 S->JA[j] = -1;
01823             }
01824         }
01825     } // end for i
01826
01827     /* Compress the strength matrix */
01828     index = 0;
01829     for (i = 0; i < row; ++i) {
01830         S->IA[i] = index;
01831         begin_row = ia[i];
01832         end_row = ia[i + 1] - 1;
01833         for (j = begin_row; j <= end_row; j++) {
01834             if (S->JA[j] > -1) {
01835                 S->JA[index] = S->JA[j];
01836                 index++;
01837             }
01838         }
01839     }
01840
01841     if (index > 0) {
01842         S->IA[row] = index;
01843         S->nnz = index;
01844         S->JA = (INT*)fasp_mem_realloc(S->JA, index * sizeof(INT));
01845     } else {
01846         S->nnz = 0;
01847         S->JA = NULL;
01848     }
01849 }
01850 #endif
01851
01867 void dCSRmat_Multicoloring_Strong_Coupled(dCSRmat* A, iCSRmat* S, INT* flags,
01868                                           INT* groups)
01869 {
01870     #if MULTI_COLOR_ORDER
01871         INT k, i, j, pre, group;
01872         INT igold, iend, iavg;
01873         INT icount;
01874         INT front, rear;
01875         INT n = A->row;
01876         INT* IA = S->IA;
01877         INT* JA = S->JA;
01878
01879         INT* cq = (INT*)malloc(sizeof(INT) * (n + 1));
01880         INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));
01881
01882     #ifndef _OPENMP
01883     #pragma omp parallel for private(k)
01884     #endif
01885         for (k = 0; k < n; k++) {
01886             cq[k] = k;
01887         }
01888         group = 0;
01889         for (k = 0; k < n; k++) {
01890             if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
01891         }
01892         *flags = group;
01893     #if 1
01894         iavg = IA[n] / n;
01895         igold = (INT)MAX(iavg, group * 0.618) + 1;
01896         igold = group;
01897     #endif
01898
01899     A->IC = (INT*)malloc(sizeof(INT) * (group + 2));
01900     A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));

```

```

01901
01902     front = n - 1;
01903     rear  = n - 1;
01904
01905     memset(newr, -1, sizeof(INT) * (n + 1));
01906     memset(A->ICMAP, 0, sizeof(INT) * n);
01907
01908     group    = 0;
01909     icount   = 0;
01910     A->IC[0] = 0;
01911     pre      = 0;
01912
01913     do {
01914         // front = (front+1)%n;
01915         front++;
01916         if (front == n) front = 0; // front = front < n ? front : 0 ;
01917         i = cq[front];
01918
01919         if (i <= pre) {
01920             A->IC[group] = icount;
01921             A->ICMAP[icount] = i;
01922             group++;
01923             icount++;
01924 #if 0
01925             if ((IA[i+1]-IA[i]) > igold)
01926                 iend = MIN(IA[i+1], (IA[i] + igold));
01927             else
01928 #endif
01929                 iend = IA[i + 1];
01930             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01931         } else if (newr[i] == group) {
01932             // rear = (rear + 1)%n;
01933             rear++;
01934             if (rear == n) rear = 0;
01935             cq[rear] = i;
01936         } else {
01937             A->ICMAP[icount] = i;
01938             icount++;
01939 #if 0
01940             if ((IA[i+1] - IA[i]) > igold) iend = MIN(IA[i+1], (IA[i] + igold));
01941             else
01942 #endif
01943                 iend = IA[i + 1];
01944             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
01945         }
01946         pre = i;
01947     } while (rear != front);
01948
01949     A->IC[group] = icount;
01950     A->color    = group;
01951
01952 #if 0
01953     for(i=0; i < A->color; i++){
01954         for(j=A -> IC[i]; j < A-> IC[i+1];j++){
01955             printf("color %d ICMAP[%d] = %d \n", i,j,A-> ICMAP[j]);
01956             printf("A.color = %d A.row= %d %d\n",A -> color,A -> row,A-> IC[i+1] - A-> IC[i] );
01957             getchar();
01958         }
01959     }
01960 #endif
01961     printf(" Max Row Numbers %d avg %d igold %d max %d %d\n", group, iavg, igold,
01962           (INT)MAX(iavg, group * 0.618), A->IA[n] / n);
01963     free(cq);
01964     free(newr);
01965     *groups = group;
01966 #endif
01967 }
01968
01984 void dCSRmat_Multicoloring_Theta(dCSRmat* A, REAL theta, INT* rowmax, INT* groups)
01985 {
01986 #if MULTI_COLOR_ORDER
01987     INT k, i, j, pre, group;
01988     INT igold, iend, iavg;
01989     INT icount;
01990     INT front, rear;
01991     INT n = A->row;
01992     //-----
01993     iCSRmat S;
01994     INT * IA, *JA;
01995     if (theta > 0 && theta < 1.0) {
01996         generate_S_theta(A, &S, theta);

```

```

01997         IA = S.IA;
01998         JA = S.JA;
01999     } else if (theta == 1.0) {
02000
02001         A->IC = (INT*)malloc(sizeof(INT) * 2);
02002         A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));
02003         A->IC[0] = 0;
02004         A->IC[1] = n;
02005 #ifdef _OPENMP
02006 #pragma omp parallel for private(k)
02007 #endif
02008         for (k = 0; k < n; k++) A->ICMAP[k] = k;
02009
02010         A->color = 1;
02011         *groups = 1;
02012         *rowmax = 1;
02013         printf("Theta = %lf \n", theta);
02014
02015         return;
02016
02017     } else {
02018         IA = A->IA;
02019         JA = A->JA;
02020     }
02021     //-----
02022     INT* cq = (INT*)malloc(sizeof(INT) * (n + 1));
02023     INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));
02024
02025 #ifdef _OPENMP
02026 #pragma omp parallel for private(k)
02027 #endif
02028     for (k = 0; k < n; k++) {
02029         cq[k] = k;
02030     }
02031     group = 0;
02032     for (k = 0; k < n; k++) {
02033         if ((A->IA[k + 1] - A->IA[k]) > group) group = A->IA[k + 1] - A->IA[k];
02034     }
02035     *rowmax = group;
02036
02037     #if 0
02038         iavg = IA[n]/n ;
02039         igold = (INT)MAX(iavg, group*0.618) + 1;
02040         igold = group ;
02041     #endif
02042
02043     A->IC = (INT*)malloc(sizeof(INT) * (group + 2));
02044     A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));
02045
02046     front = n - 1;
02047     rear = n - 1;
02048
02049     memset(newr, -1, sizeof(INT) * (n + 1));
02050     memset(A->ICMAP, 0, sizeof(INT) * n);
02051
02052     group = 0;
02053     icount = 0;
02054     A->IC[0] = 0;
02055     pre = 0;
02056
02057     do {
02058         // front = (front+1)%n;
02059         front++;
02060         if (front == n) front = 0; // front = front < n ? front : 0 ;
02061         i = cq[front];
02062
02063         if (i <= pre) {
02064             A->IC[group] = icount;
02065             A->ICMAP[icount] = i;
02066             group++;
02067             icount++;
02068         #if 0
02069             if ((IA[i+1]-IA[i]) > igold)
02070                 iend = MIN(IA[i+1], (IA[i] + igold));
02071             else
02072         #endif
02073             iend = IA[i + 1];
02074             for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
02075         } else if (newr[i] == group) {
02076             // rear = (rear + 1)%n;
02077             rear++;

```

```

02078         if (rear == n) rear = 0;
02079         cq[rear] = i;
02080     } else {
02081         A->ICMAP[icount] = i;
02082         icount++;
02083 #if 0
02084         if ((IA[i+1] - IA[i]) > igold) iend = MIN(IA[i+1], (IA[i] + igold));
02085         else
02086 #endif
02087             iend = IA[i + 1];
02088         for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
02089     }
02090     pre = i;
02091
02092     //    printf("pre = %d\n",pre);
02093 } while (rear != front);
02094
02095 //    printf("group\n");
02096 A->IC[group] = icount;
02097 A->color    = group;
02098
02099 #if 0
02100     for(i=0; i < A->color; i++){
02101         for(j=A -> IC[i]; j < A-> IC[i+1];j++)
02102             printf("color %d ICMAP[%d] = %d \n", i,j,A-> ICMAP[j]);
02103         printf( "A.color = %d A.row= %d %d\n",A -> color,A -> row,A-> IC[i+1] - A-> IC[i] );
02104         getchar();
02105     }
02106     printf(" Max Row Numbers %d avg %d igold %d max %d %d\n", group, iavg, igold,
(INT)MAX(iavg,group*0.618),A->IA[n]/n );
02107 #endif
02108     free(cq);
02109     free(newr);
02110     if (theta > 0) {
02111         fasp_mem_free(S.IA);
02112         fasp_mem_free(S.JA);
02113     }
02114     *groups = group;
02115 #endif
02116     return;
02117 }
02118
02119 /*
02120 * TODO: Why it is not in ItrSmootherCSR.c? Move?
02121 * TODO: Add Doxygen!
02122 */
02123 void fasp_smoother_dcsr_gs_multicolor(dvector* u, dCSRmat* A, dvector* b, INT L,
02124                                       const INT order)
02125 {
02126 #if MULTI_COLOR_ORDER
02127     const INT nrow = A->row; // number of rows
02128     const INT * ia = A->IA, *ja = A->JA;
02129     const REAL *aj = A->val, *bval = b->val;
02130     REAL*      uval = u->val;
02131
02132     INT i, j, k, begin_row, end_row;
02133     REAL t, d = 0.0;
02134
02135     INT myid, mybegin, myend;
02136     INT color = A->color;
02137     INT* IC    = A->IC;
02138     INT* ICMAP = A->ICMAP;
02139     INT I;
02140
02141     // From color to 0 order
02142     if (order == -1) {
02143         while (L-->0) {
02144             for (myid = color - 1; myid > -1; myid--) {
02145                 mybegin = IC[myid];
02146                 myend   = IC[myid + 1];
02147 #ifdef _OPENMP
02148 #pragma omp parallel for private(I, i, t, begin_row, end_row, k, j, d)
02149 #endif
02150                 for (I = mybegin; I < myend; I++) {
02151                     i = ICMAP[I];
02152                     t = bval[i];
02153                     begin_row = ia[i], end_row = ia[i + 1];
02154                     for (k = begin_row; k < end_row; k++) {
02155                         j = ja[k];
02156                         if (i != j)
02157                             t -= aj[k] * uval[j];

```

```

02158             else
02159                 d = aj[k];
02160             } // end for k
02161             if (ABS(d) > SMALLREAL) uval[i] = t / d;
02162         } // end for I
02163     } // end for myid
02164 } // end while
02165 }
02166 // From 0 to color order
02167 else {
02168     while (L--) {
02169         for (myid = 0; myid < color; myid++) {
02170             mybegin = IC[myid];
02171             myend = IC[myid + 1];
02172 #ifdef _OPENMP
02173 #pragma omp parallel for private(I, i, t, begin_row, end_row, k, j, d)
02174 #endif
02175             for (I = mybegin; I < myend; I++) {
02176                 i = ICMAP[I];
02177                 t = bval[i];
02178                 begin_row = ia[i], end_row = ia[i + 1];
02179                 for (k = begin_row; k < end_row; k++) {
02180                     j = ja[k];
02181                     if (i != j)
02182                         t -= aj[k] * uval[j];
02183                     else
02184                         d = aj[k];
02185                 } // end for k
02186                 if (ABS(d) > SMALLREAL) uval[i] = t / d;
02187             } // end for I
02188         } // end for myid
02189     } // end while
02190 } // end if order
02191 #else
02192     printf("### ERROR: MULTI_COLOR_ORDER has not been turn on!!! \n");
02193 #endif
02194     return;
02195 }
02196
02197 /*-----*/
02198 /*--      End of File      --*/
02199 /*-----*/

```

9.81 BlaSparseCSRL.c File Reference

Sparse matrix operations for [dCSRLmat](#) matrices.

```
#include "fasp.h"
#include "fasp_funcs.h"
```

Functions

- [dCSRLmat](#) * [fasp_dcsrl_create](#) (const [INT](#) num_rows, const [INT](#) num_cols, const [INT](#) num_nonzeros)
Create a [dCSRLmat](#) object.
- void [fasp_dcsrl_free](#) ([dCSRLmat](#) *A)
Destroy a [dCSRLmat](#) object.

9.81.1 Detailed Description

Sparse matrix operations for [dCSRLmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#)

Reference: John Mellor-Crummy and John Garvin Optimizaing sparse matrix vector product computations using unroll and jam, Tech Report Rice Univ, Aug 2002.
Copyright (C) 2011–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseCSRL.c](#).

9.81.2 Function Documentation

9.81.2.1 fasp_dcsrl_create()

```
dCSRLmat * fasp_dcsrl_create (
    const INT num_rows,
    const INT num_cols,
    const INT num_nonzeros )
```

Create a [dCSRLmat](#) object.

Parameters

<i>num_rows</i>	Number of rows
<i>num_cols</i>	Number of cols
<i>num_nonzeros</i>	Number of nonzero entries

Author

Zhiyang Zhou

Date

01/07/2011

Definition at line 39 of file [BlaSparseCSRL.c](#).

9.81.2.2 fasp_dcsrl_free()

```
void fasp_dcsrl_free (
    dCSRLmat * A )
```

Destroy a [dCSRLmat](#) object.

Parameters

<i>A</i>	Pointer to the dCSRLmat type matrix
----------	---

Author

Zhiyang Zhou

Date

01/07/2011

Definition at line 67 of file [BlaSparseCSRL.c](#).

9.82 BlaSparseCSRL.c

[Go to the documentation of this file.](#)


```

00001
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--      Public Functions      --*/
00024 /*-----*/
00025
00039 dCSRmat * fasp_dcsr_create (const INT num_rows,
00040                             const INT num_cols,
00041                             const INT num_nonzeros)
00042 {
00043     dCSRmat *A = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));
00044
00045     A -> row      = num_rows;
00046     A -> col      = num_cols;
00047     A -> nnz      = num_nonzeros;
00048     A -> nz_diff  = NULL;
00049     A -> index    = NULL;
00050     A -> start    = NULL;
00051     A -> ja       = NULL;
00052     A -> val      = NULL;
00053
00054     return A;
00055 }
00056
00067 void fasp_dcsr_free (dCSRmat *A)
00068 {
00069     if (A) {
00070         if (A -> nz_diff) free(A -> nz_diff);
00071         if (A -> index)   free(A -> index);
00072         if (A -> start)   free(A -> start);
00073         if (A -> ja)      free(A -> ja);
00074         if (A -> val)     free(A -> val);
00075         free(A);
00076     }
00077 }
00078
00079 /*-----*/
00080 /*--      End of File      --*/
00081 /*-----*/

```

9.83 BlaSparseSTR.c File Reference

Sparse matrix operations for **dSTRmat** matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- **dSTRmat fasp_dstr_create** (const **INT** nx, const **INT** ny, const **INT** nz, const **INT** nc, const **INT** nband, **INT** *offsets)
Create STR sparse matrix data memory space.
- void **fasp_dstr_alloc** (const **INT** nx, const **INT** ny, const **INT** nz, const **INT** nxy, const **INT** ngrid, const **INT** nband, const **INT** nc, **INT** *offsets, **dSTRmat** *A)
Allocate STR sparse matrix memory space.
- void **fasp_dstr_free** (**dSTRmat** *A)
Free STR sparse matrix data memory space.
- void **fasp_dstr_cp** (const **dSTRmat** *A, **dSTRmat** *B)
*Copy a **dSTRmat** to a new one B=A.*

9.83.1 Detailed Description

Sparse matrix operations for **dSTRmat** matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseSTR.c](#).

9.83.2 Function Documentation

9.83.2.1 fasp_dstr_alloc()

```
void fasp_dstr_alloc (
    const INT nx,
    const INT ny,
    const INT nz,
    const INT nxy,
    const INT ngrid,
    const INT nband,
    const INT nc,
    INT * offsets,
    dSTRmat * A )
```

Allocate STR sparse matrix memory space.

Parameters

<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>nz</i>	Number of grids in z direction
<i>nxy</i>	Number of grids in x-y plane
<i>ngrid</i>	Number of grids
<i>nband</i>	Number of off-diagonal bands
<i>nc</i>	Number of components
<i>offsets</i>	Shift from diagonal
<i>A</i>	Pointer to the dSTRmat matrix

Author

Shiquan Zhang, Xiaozhe Hu

Date

05/17/2010

Definition at line 93 of file [BlaSparseSTR.c](#).

9.83.2.2 fasp_dstr_cp()

```
void fasp_dstr_cp (
```

```
const dSTRmat * A,
dSTRmat * B )
```

Copy a [dSTRmat](#) to a new one B=A.

Parameters

<i>A</i>	Pointer to the dSTRmat matrix
<i>B</i>	Pointer to the dSTRmat matrix

Author

Zhiyang Zhou

Date

04/21/2010

Definition at line 162 of file [BlaSparseSTR.c](#).

9.83.2.3 fasp_dstr_create()

```
dSTRmat fasp_dstr_create (
    const INT nx,
    const INT ny,
    const INT nz,
    const INT nc,
    const INT nband,
    INT * offsets )
```

Create STR sparse matrix data memory space.

Parameters

<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>nz</i>	Number of grids in z direction
<i>nc</i>	Number of components
<i>nband</i>	Number of off-diagonal bands
<i>offsets</i>	Shift from diagonal

Returns

The [dSTRmat](#) matrix

Author

Shiquan Zhang, Xiaozhe Hu

Date

05/17/2010

Definition at line 41 of file [BlaSparseSTR.c](#).

9.83.2.4 fasp_dstr_free()

```
void fasp_dstr_free (
    dSTRmat * A )
```

Free STR sparse matrix data memeory space.

Parameters

A	Pointer to the <code>dSTRmat</code> matrix
----------	--

Author

Shiquan Zhang, Xiaozhe Hu

Date

05/17/2010

Definition at line 136 of file `BlaSparseSTR.c`.

9.84 BlaSparseSTR.c

[Go to the documentation of this file.](#)

```
00001
00014 #include <math.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_functs.h"
00018
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00041 dSTRmat fasp_dstr_create (const INT  nx,
00042                          const INT  ny,
00043                          const INT  nz,
00044                          const INT  nc,
00045                          const INT  nband,
00046                          INT        *offsets)
00047 {
00048     dSTRmat A;
00049
00050     INT i;
00051
00052     A.nx=nx; A.ny=ny; A.nz=nz;
00053     A.nc=nc;
00054     A.nxy=A.nx*A.ny;
00055     A.ngrid=A.nxy*A.nz;
00056     A.nband=nband;
00057
00058     A.offsets=(INT*) fasp_mem_calloc(nband, sizeof(INT));
00059
00060     for (i=0;i<nband;++i) A.offsets[i]=offsets[i];
00061
00062     A.diag=(REAL*) fasp_mem_calloc(A.ngrid*A.nc*A.nc, sizeof(REAL));
00063
00064     A.offdiag=(REAL**) fasp_mem_calloc(nband, sizeof(REAL*));
00065
00066     for (i=0;i<A.nband;++i) {
00067         A.offdiag[i]=(REAL*) fasp_mem_calloc((A.ngrid-ABS(A.offsets[i]))*A.nc*A.nc, sizeof(REAL));
00068     }
00069
00070     return(A);
00071 }
00072
00093 void fasp_dstr_alloc (const INT  nx,
00094                     const INT  ny,
00095                     const INT  nz,
00096                     const INT  nxy,
00097                     const INT  ngrid,
```

```

00098         const INT  nband,
00099         const INT  nc,
00100         INT        *offsets,
00101         dSTRmat    *A)
00102 {
00103     INT i;
00104
00105     A->nx=nx;
00106     A->ny=ny;
00107     A->nz=nz;
00108     A->nxy=nxy;
00109     A->ngrid=ngrid;
00110     A->nband=nband;
00111     A->nc=nc;
00112
00113     A->offsets=(INT*) fasp_mem_calloc(nband, sizeof(INT));
00114
00115     for (i=0;i<nband;++i) A->offsets[i]=offsets[i];
00116
00117     A->diag=(REAL*) fasp_mem_calloc(ngrid*nc*nc, sizeof(REAL));
00118
00119     A->offdiag = (REAL **) fasp_mem_calloc(A->nband, sizeof(REAL*));
00120
00121     for (i=0;i<nband;++i) {
00122         A->offdiag[i]=(REAL*) fasp_mem_calloc((ngrid-ABS(offsets[i]))*nc*nc, sizeof(REAL));
00123     }
00124 }
00125
00136 void fasp_dstr_free (dSTRmat *A)
00137 {
00138     INT i;
00139
00140     fasp_mem_free(A->offsets); A->offsets = NULL;
00141     fasp_mem_free(A->diag);    A->diag   = NULL;
00142
00143     for ( i = 0; i < A->nband; ++i ) {
00144         fasp_mem_free(A->offdiag[i]); A->offdiag[i] = NULL;
00145     }
00146
00147     A->nx = A->ny = A->nz = A->nxy=0;
00148     A->ngrid = A->nband = A->nc=0;
00149 }
00150
00162 void fasp_dstr_cp (const dSTRmat *A,
00163                  dSTRmat      *B)
00164 {
00165     const INT nc2 = (A->nc)*(A->nc);
00166
00167     INT i;
00168     B->nx = A->nx;
00169     B->ny = A->ny;
00170     B->nz = A->nz;
00171     B->nxy = A->nxy;
00172     B->ngrid = A->ngrid;
00173     B->nc = A->nc;
00174     B->nband = A->nband;
00175
00176     memcpy(B->offsets,A->offsets,(A->nband)*sizeof(INT));
00177     memcpy(B->diag,A->diag,(A->ngrid*nc2)*sizeof(REAL));
00178     for (i=0;i<A->nband;++i) {
00179         memcpy(B->offdiag[i],A->offdiag[i],
00180              ((A->ngrid - ABS(A->offsets[i]))*nc2)*sizeof(REAL));
00181     }
00182 }
00183
00184 /*-----*/
00185 /*--      End of File      --*/
00186 /*-----*/

```

9.85 BlaSparseUtil.c File Reference

Routines for sparse matrix operations.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_sparse_abybms_` (INT *ia, INT *ja, INT *ib, INT *jb, INT *nap, INT *map, INT *mbp, INT *ic, INT *jc)
Multiplication of two sparse matrices: calculating the nonzero structure of the result if jc is not null. If jc is null only finds num of nonzeros.
- void `fasp_sparse_abyb_` (INT *ia, INT *ja, REAL *a, INT *ib, INT *jb, REAL *b, INT *nap, INT *map, INT *mbp, INT *ic, INT *jc, REAL *c)
Multiplication of two sparse matrices.
- void `fasp_sparse_iit_` (INT *ia, INT *ja, INT *na, INT *ma, INT *iat, INT *jat)
Transpose a boolean matrix (only given by ia, ja)
- void `fasp_sparse_aat_` (INT *ia, INT *ja, REAL *a, INT *na, INT *ma, INT *iat, INT *jat, REAL *at)
Transpose a boolean matrix (only given by ia, ja)
- void `fasp_sparse_aplbms_` (INT *ia, INT *ja, INT *ib, INT *jb, INT *nab, INT *mab, INT *ic, INT *jc)
Addition of two sparse matrices: calculating the nonzero structure of the result if jc is not null. if jc is null only finds num of nonzeros.
- void `fasp_sparse_aplusb_` (INT *ia, INT *ja, REAL *a, INT *ib, INT *jb, REAL *b, INT *nab, INT *mab, INT *ic, INT *jc, REAL *c)
Addition of two sparse matrices.
- void `fasp_sparse_rapms_` (INT *ir, INT *jr, INT *ia, INT *ja, INT *ip, INT *jp, INT *nin, INT *ncin, INT *iac, INT *jac, INT *maxrout)
Calculates the nonzero structure of $R \cdot A \cdot P$, if jac is not null. If jac is null only finds num of nonzeros.
- void `fasp_sparse_wtams_` (INT *jw, INT *ia, INT *ja, INT *nwp, INT *map, INT *jv, INT *nvp, INT *icp)
Finds the nonzeros in the result of $v^t = w^t A$, where w is a sparse vector and A is sparse matrix. jv is an integer array containing the indices of the nonzero elements in the result.
- void `fasp_sparse_wta_` (INT *jw, REAL *w, INT *ia, INT *ja, REAL *a, INT *nwp, INT *map, INT *jv, REAL *v, INT *nvp)
Calculate $v^t = w^t A$, where w is a sparse vector and A is sparse matrix. v is an array of dimension = number of columns in A.
- void `fasp_sparse_ytxbig_` (INT *jy, REAL *y, INT *nyp, REAL *x, REAL *s)
Calculates $s = y^t x$. y-sparse, x - no.
- void `fasp_sparse_ytx_` (INT *jy, REAL *y, INT *jx, REAL *x, INT *nyp, INT *npx, INT *icp, REAL *s)
Calculates $s = y^t x$. y is sparse, x is sparse.
- void `fasp_sparse_rapcmp_` (INT *ir, INT *jr, REAL *r, INT *ia, INT *ja, REAL *a, INT *ipt, INT *jpt, REAL *pt, INT *nin, INT *ncin, INT *iac, INT *jac, REAL *ac, INT *idummy)
Calculates $R \cdot A \cdot P$ after the nonzero structure of the result is known. iac,jac,ac have to be allocated before call to this function.
- `ivector fasp_sparse_mis` (dCSRmat *A)
Get the maximal independet set of a CSR matrix.

9.85.1 Detailed Description

Routines for sparse matrix operations.

Note

Most algorithms work as follows: (a) Boolean operations (to determine the nonzero structure); (b) Numerical part, where the result is calculated.

Parameter notation :I: is input; :O: is output; :IO: is both

This file contains Level-1 (Bla) functions. It requires: [AuxMemory.c](#)

Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSparseUtil.c](#).

9.85.2 Function Documentation

9.85.2.1 fasp_sparse_aat_()

```
void fasp_sparse_aat_ (
    INT * ia,
    INT * ja,
    REAL * a,
    INT * na,
    INT * ma,
    INT * iat,
    INT * jat,
    REAL * at )
```

Transpose a boolean matrix (only given by ia, ja)

Parameters

<i>ia</i>	array of row pointers (as usual in CSR)
<i>ja</i>	array of column indices
<i>a</i>	array of entries of teh input
<i>na</i>	number of rows of A
<i>ma</i>	number of cols of A
<i>iat</i>	array of row pointers in the result
<i>jat</i>	array of column indices
<i>at</i>	array of entries of the result

Definition at line 273 of file [BlaSparseUtil.c](#).

9.85.2.2 fasp_sparse_abyb_()

```
void fasp_sparse_abyb_ (
    INT * ia,
    INT * ja,
    REAL * a,
    INT * ib,
    INT * jb,
    REAL * b,
    INT * nap,
    INT * map,
    INT * mbp,
    INT * ic,
    INT * jc,
    REAL * c )
```

Multiplication of two sparse matrices.

Parameters

<i>ia</i>	array of row pointers 1st multiplicand
<i>ja</i>	array of column indices 1st multiplicand
<i>a</i>	entries of the 1st multiplicand
<i>ib</i>	array of row pointers 2nd multiplicand
<i>jb</i>	array of column indices 2nd multiplicand
<i>b</i>	entries of the 2nd multiplicand
<i>ic</i>	array of row pointers in $c=a*b$
<i>jc</i>	array of column indices in $c=a*b$
<i>c</i>	entries of the result: $c=a*b$
<i>nap</i>	number of rows in the 1st multiplicand
<i>map</i>	number of columns in the 1st multiplicand
<i>mbp</i>	number of columns in the 2nd multiplicand

Modified by Chensong Zhang on 09/11/2012

Definition at line 127 of file [BlaSparseUtil.c](#).

9.85.2.3 fasp_sparse_abybms_()

```
void fasp_sparse_abybms_ (
    INT * ia,
    INT * ja,
    INT * ib,
    INT * jb,
    INT * nap,
    INT * map,
    INT * mbp,
    INT * ic,
    INT * jc )
```

Multiplication of two sparse matrices: calculating the nonzero structure of the result if *jc* is not null. If *jc* is null only finds num of nonzeros.

Parameters

<i>ia</i>	array of row pointers 1st multiplicand
<i>ja</i>	array of column indices 1st multiplicand
<i>ib</i>	array of row pointers 2nd multiplicand
<i>jb</i>	array of column indices 2nd multiplicand
<i>nap</i>	number of rows of A
<i>map</i>	number of cols of A
<i>mbp</i>	number of cols of b
<i>ic</i>	array of row pointers in the result (this is also computed here again, so that we can have a stand alone call of this routine, if for some reason the number of nonzeros in the result is known)
<i>jc</i>	array of column indices in the result $c=a*b$

Modified by Chensong Zhang on 09/11/2012

Definition at line 52 of file [BlaSparseUtil.c](#).

9.85.2.4 fasp_sparse_aplbms_()

```
void void fasp_sparse_aplbms_ (
    INT * ia,
    INT * ja,
    INT * ib,
    INT * jb,
    INT * nab,
    INT * mab,
    INT * ic,
    INT * jc )
```

Addition of two sparse matrices: calculating the nonzero structure of the result if jc is not null. if jc is null only finds num of nonzeros.

Parameters

<i>ia</i>	array of row pointers 1st summand
<i>ja</i>	array of column indices 1st summand
<i>ib</i>	array of row pointers 2nd summand
<i>jb</i>	array of column indices 2nd summand
<i>nab</i>	number of rows
<i>mab</i>	number of cols
<i>ic</i>	array of row pointers in the result (this is also computed here again, so that we can have a stand alone call of this routine, if for some reason the number of nonzeros in the result is known)
<i>jc</i>	array of column indices in the result c=a+b

Definition at line 359 of file [BlaSparseUtil.c](#).

9.85.2.5 fasp_sparse_aplusb_()

```
void fasp_sparse_aplusb_ (
    INT * ia,
    INT * ja,
    REAL * a,
    INT * ib,
    INT * jb,
    REAL * b,
    INT * nab,
    INT * mab,
    INT * ic,
    INT * jc,
    REAL * c )
```

Addition of two sparse matrices.

Parameters

<i>ia</i>	array of row pointers 1st summand
<i>ja</i>	array of column indices 1st summand
<i>a</i>	entries of the 1st summand
<i>ib</i>	array of row pointers 2nd summand
<i>jb</i>	array of column indices 2nd summand
<i>b</i>	entries of the 2nd summand

Parameters

<i>nab</i>	number of rows
<i>mab</i>	number of cols
<i>ic</i>	array of row pointers in c=a+b
<i>jc</i>	array of column indices in c=a+b
<i>c</i>	entries of the result: c=a+b

Definition at line 431 of file [BlaSparseUtil.c](#).

9.85.2.6 fasp_sparse_iit_()

```
void fasp_sparse_iit_ (
    INT * ia,
    INT * ja,
    INT * na,
    INT * ma,
    INT * iat,
    INT * jat )
```

Transpose a boolean matrix (only given by ia, ja)

Parameters

<i>ia</i>	array of row pointers (as usual in CSR)
<i>ja</i>	array of column indices
<i>na</i>	number of rows
<i>ma</i>	number of cols
<i>iat</i>	array of row pointers in the result
<i>jat</i>	array of column indices

Definition at line 197 of file [BlaSparseUtil.c](#).

9.85.2.7 fasp_sparse_mis()

```
ivector fasp_sparse_mis (
    dCSRmat * A )
```

Get the maximal independet set of a CSR matrix.

Parameters

<i>A</i>	pointer to the matrix
----------	-----------------------

Note

Only use the sparsity of A, index starts from 1 (fortran)!!

Definition at line 907 of file BlasSparseUtil.c.

9.85.2.8 fasp_sparse_rapcmp_()

```
void fasp_sparse_rapcmp_ (
    INT * ir,
    INT * jr,
    REAL * r,
    INT * ia,
    INT * ja,
    REAL * a,
    INT * ipt,
    INT * jpt,
    REAL * pt,
    INT * nin,
    INT * ncin,
    INT * iac,
    INT * jac,
    REAL * ac,
    INT * idummy )
```

Calculates $R \cdot A \cdot P$ after the nonzero structure of the result is known. iac,jac,ac have to be allocated before call to this function.

Note

:I: is input :O: is output :IO: is both

Parameters

<i>ir</i>	:I: array of row pointers for R
<i>jr</i>	:I: array of column indices for R
<i>r</i>	:I: entries of R
<i>ia</i>	:I: array of row pointers for A
<i>ja</i>	:I: array of column indices for A
<i>a</i>	:I: entries of A
<i>ipt</i>	:I: array of row pointers for P
<i>jpt</i>	:I: array of column indices for P
<i>pt</i>	:I: entries of P
<i>nin</i>	:I: number of rows in R
<i>ncin</i>	:I: number of rows in
<i>iac</i>	:O: array of row pointers for P
<i>jac</i>	:O: array of column indices for P
<i>ac</i>	:O: entries of P
<i>idummy</i>	not changed

Note

Compute $R \cdot A \cdot P$ for known nonzero structure of the result the result is stored in iac,jac,ac!

Definition at line 787 of file [BlaSparseUtil.c](#).

9.85.2.9 fasp_sparse_rapms_()

```
void fasp_sparse_rapms_ (
    INT * ir,
    INT * jr,
    INT * ia,
    INT * ja,
    INT * ip,
    INT * jp,
    INT * nin,
    INT * ncin,
    INT * iac,
    INT * jac,
    INT * maxrout )
```

Calculates the nonzero structure of $R \cdot A \cdot P$, if jac is not null. If jac is null only finds num of nonzeros.

Note

:I: is input :O: is output :IO: is both

Parameters

<i>ir</i>	:I: array of row pointers for R
<i>jr</i>	:I: array of column indices for R
<i>ia</i>	:I: array of row pointers for A
<i>ja</i>	:I: array of column indices for A
<i>ip</i>	:I: array of row pointers for P
<i>jp</i>	:I: array of column indices for P
<i>nin</i>	:I: number of rows in R
<i>ncin</i>	:I: number of columns in R
<i>iac</i>	:O: array of row pointers for Ac
<i>jac</i>	:O: array of column indices for Ac
<i>maxrout</i>	:O: the maximum nonzeros per row for R

Note

Computes the sparsity pattern of $R \cdot A \cdot P$. maxrout is output and is the maximum nonzeros per row for r. On output we also have iac (if jac is null) and jac (if jac entry is not null). R is (nc,n) A is (n,n) and P is (n,nc)!

Modified by Chensong Zhang on 09/11/2012

Definition at line 515 of file [BlaSparseUtil.c](#).

9.85.2.10 fasp_sparse_wta_()

```
void fasp_sparse_wta_ (
    INT * jw,
```

```

REAL * w,
INT * ia,
INT * ja,
REAL * a,
INT * nwp,
INT * map,
INT * jv,
REAL * v,
INT * nvp )

```

Calculate $v^t = w^t A$, where w is a sparse vector and A is sparse matrix. v is an array of dimension = number of columns in A .

Note

:I: is input :O: is output :IO: is both

Parameters

<i>jw</i>	:I: indices such that $w[jw]$ is nonzero
<i>w</i>	:I: the values of w
<i>ia</i>	:I: array of row pointers for A
<i>ja</i>	:I: array of column indices for A
<i>a</i>	:I: entries of A
<i>nwp</i>	:I: number of nonzeros in w (the length of w)
<i>map</i>	:I: number of columns in A
<i>jv</i>	:O: indices such that $v[jv]$ is nonzero
<i>v</i>	:O: the result $v^t = w^t A$
<i>nvp</i>	:I: number of nonzeros in v

Definition at line 648 of file [BlaSparseUtil.c](#).

9.85.2.11 fasp_sparse_wtams_()

```

void fasp_sparse_wtams_ (
    INT * jw,
    INT * ia,
    INT * ja,
    INT * nwp,
    INT * map,
    INT * jv,
    INT * nvp,
    INT * icp )

```

Finds the nonzeros in the result of $v^t = w^t A$, where w is a sparse vector and A is sparse matrix. jv is an integer array containing the indices of the nonzero elements in the result.

:I: is input :O: is output :IO: is both

Parameters

<i>jw</i>	:I: indices such that $w[jw]$ is nonzero
<i>ia</i>	:I: array of row pointers for A
<i>ja</i>	:I: array of column indices for A

Parameters

<i>nwp</i>	:l: number of non zeroes in w (the length of w)
<i>map</i>	:l: number of columns in A
<i>jv</i>	:O: indices such that v[jv] is nonzero
<i>nvp</i>	:l: number of non zeroes in v
<i>icp</i>	:IO: is a working array of length (*map) which on output satisfies icp[jv[k]-1]=k; Values of icp[] at positions * other than (jv[k]-1) remain unchanged.

Modified by Chensong Zhang on 09/11/2012

Definition at line 596 of file [BlaSparseUtil.c](#).

9.85.2.12 fasp_sparse_ytx_()

```
void fasp_sparse_ytx_ (
    INT * jy,
    REAL * y,
    INT * jx,
    REAL * x,
    INT * nyp,
    INT * nxp,
    INT * icp,
    REAL * s )
```

Calculates $s = y^t x$. y is sparse, x is sparse.

Note

:l: is input :O: is output :IO: is both

Parameters

<i>jy</i>	:l: indices such that y[jy] is nonzero
<i>y</i>	:l: is a sparse vector.
<i>nyp</i>	:l: number of non zeroes in y
<i>jx</i>	:l: indices such that x[jx] is nonzero
<i>x</i>	:l: is a sparse vector.
<i>nxp</i>	:l: number of non zeroes in x
<i>icp</i>	???
<i>s</i>	:O: $s = y^t x$.

Definition at line 733 of file [BlaSparseUtil.c](#).

9.85.2.13 fasp_sparse_ytxbig_()

```
void fasp_sparse_ytxbig_ (
    INT * jy,
    REAL * y,
    INT * nyp,
    REAL * x,
    REAL * s )
```

Calculates $s = y^t x$. y -sparse, x - no.

Note

:I: is input :O: is output :IO: is both

Parameters

<i>jy</i>	:I: indices such that $y[jy]$ is nonzero
<i>y</i>	:I: is a sparse vector
<i>nyp</i>	:I: number of non zeroes in v
<i>x</i>	:I: also a vector assumed to have entry for any $j=jy[i]-1$; for $i=1:nyp$. This means that x here does not have to be sparse
<i>s</i>	:O: $s = y^t x$

Definition at line 699 of file [BlaSparseUtil.c](#).

9.86 BlaSparseUtil.c

[Go to the documentation of this file.](#)

```

00001
00021 #include <math.h>
00022 #include <time.h>
00023
00024 #include "fasp.h"
00025 #include "fasp_functs.h"
00026
00027 /*-----*/
00028 /*--      Public Functions      --*/
00029 /*-----*/
00052 void fasp_sparse_abybms_ (INT *ia,
00053                          INT *ja,
00054                          INT *ib,
00055                          INT *jb,
00056                          INT *nap,
00057                          INT *map,
00058                          INT *mbp,
00059                          INT *ic,
00060                          INT *jc)
00061 {
00062     /* FORM ic when jc is null and both when jc is not null for
00063 the ic and jc are for c=a*b, a and b sparse */
00064     /* na = number of rows of a */
00065     /* mb = number of columns of b */
00066     unsigned int jcform=0;
00067     INT na,mb,icpp,iastrt,ibstrt,iaend,ibend,i,j,k,jia,jib;
00068     INT *icp;
00069     if (jc) jcform=1;
00070     na=*nap;
00071     mb=*mbp;
00072     icpp = 1;
00073     icp=(INT *) calloc(mb,sizeof(INT));
00074
00075     for (i = 0; i < mb; ++i) icp[i] = 0;
00076
00077     for (i = 0; i < na; ++i) {
00078         ic[i] = icpp;
00079         iastrt = ia[i]-1;
00080         iaend = ia[i+1]-1;
00081         if (iaend > iastrt) {
00082             for (jia = iastrt; jia < iaend; ++jia) {
00083                 j = ja[jia]-1;
00084                 ibstrt = ib[j]-1;
00085                 ibend = ib[j+1]-1;
00086                 if (ibend > ibstrt) {
00087                     for (jib = ibstrt; jib < ibend; ++jib) {
00088                         k = jb[jib]-1;
00089                         if (icp[k] != i+1) {
00090                             if (jcform) jc[icpp-1] = k+1;

```

```

00091                                     ++icpp;
00092                                     icp[k] = i+1;
00093                                     } //if
00094                                 } //for
00095                             } //if
00096                         } //for
00097                     } //if
00098                 } //for (i...
00099                 ic[na] = icpp;
00100
00101                 if (icp) free(icp);
00102
00103                 return;
00104             }
00105
00127 void fasp_sparse_abyb_ (INT *ia,
00128                        INT *ja,
00129                        REAL *a,
00130                        INT *ib,
00131                        INT *jb,
00132                        REAL *b,
00133                        INT *nap,
00134                        INT *map,
00135                        INT *mbp,
00136                        INT *ic,
00137                        INT *jc,
00138                        REAL *c)
00139 {
00140     INT na,mb,iastrt,ibstrt,iaend,ibend,icstrt,icend,i,j,k,ji,jia,jib;
00141     REAL **x;
00142     REAL x0;
00143     /*
00144 C-----
00145 C...   C = A*B
00146 C-----
00147 */
00148     na=*nap;
00149     mb=*mbp;
00150     x=(REAL *)calloc(mb,sizeof(REAL));
00151     for (i = 0; i < na; ++i) {
00152         icstrt = ic[i]-1;
00153         icend = ic[i+1]-1;
00154         if (icend > icstrt) {
00155             for (ji = icstrt; ji < icend; ++ji) {
00156                 k=jc[ji]-1;
00157                 x[k] = 0e+0;
00158             }
00159             iastrt = ia[i]-1;
00160             iaend = ia[i+1]-1;
00161             if (iaend > iastrt) {
00162                 for (jia = iastrt; jia < iaend ; ++jia) {
00163                     j = ja[jia]-1;
00164                     x0 = a[jia];
00165                     ibstrt = ib[j]-1;
00166                     ibend = ib[j+1]-1;
00167                     if (ibend > ibstrt) {
00168                         for (jib = ibstrt; jib < ibend; ++jib) {
00169                             k = jb[jib]-1;
00170                             x[k] += x0*b[jib];
00171                         }
00172                     } // end if
00173                 } // end for
00174             }
00175             for (ji = icstrt; ji < icend; ++ji) {
00176                 k=jc[ji]-1;
00177                 c[ji]=x[k];
00178             } // end for
00179         } // end if
00180     } //end do
00181     if (x) free(x);
00182     return;
00183 }
00184
00197 void fasp_sparse_iit_ (INT *ia,
00198                      INT *ja,
00199                      INT *na,
00200                      INT *ma,
00201                      INT *iat,
00202                      INT *jat)
00203 {
00204     /*C=====*/

```



```

00205     INT i, j, jp, n, m, mh, nh, iaa, iab, k;
00206     /*
00207 C-----
00208 C...   Transposition of a graph (or the matrix) symbolically.
00209 C...
00210 C...   Input:
00211 C...       IA, JA   - given graph (or matrix).
00212 C...       N        - number of rows of the matrix.
00213 C...       M        - number of columns of the matrix.
00214 C...
00215 C...   Output:
00216 C...       IAT, JAT - transposed graph (or matrix).
00217 C...
00218 C...   Note:
00219 C...       N+1 is the dimension of IA.
00220 C...       M+1 is the dimension of IAT.
00221 C-----
00222 */
00223     n=*na;
00224     m=*ma;
00225     mh = m + 1;
00226     nh = n + 1;
00227     for (i = 1; i < mh; ++i) {
00228         iat[i] = 0;
00229     }
00230     iab = ia[nh-1] - 1;
00231     for (i = 1; i <= iab; ++i) {
00232         j = ja[i-1] + 2;
00233         if (j <= mh)
00234             iat[j-1] = iat[j-1] + 1;
00235     }
00236     iat[0] = 1;
00237     iat[1] = 1;
00238     if (m != 1) {
00239         for (i = 2; i < mh; ++i) {
00240             iat[i] = iat[i] + iat[i-1];
00241         }
00242     }
00243     for (i = 1; i <= n; ++i) {
00244         iaa = ia[i-1];
00245         iab = ia[i] - 1;
00246         if (iab >= iaa) {
00247             for (jp = iaa; jp <= iab; ++jp) {
00248                 j = ja[jp-1] + 1;
00249                 k = iat[j-1];
00250                 jat[k-1] = i;
00251                 iat[j-1] = k + 1;
00252             }
00253         }
00254     }
00255     return;
00256 }
00257
00273 void fasp_sparse_aat_ (INT  *ia,
00274                        INT  *ja,
00275                        REAL *a,
00276                        INT  *na,
00277                        INT  *ma,
00278                        INT  *iat,
00279                        INT  *jat,
00280                        REAL *at)
00281 {
00282     /*C=====*/
00283     INT i, j, jp, n, m, mh, nh, iaa, iab, k;
00284     /*
00285 C-----
00286 C...   Transposition of a matrix.
00287 C...
00288 C...   Input:
00289 C...       IA, JA   - given graph (or matrix).
00290 C...       N        - number of rows of the matrix.
00291 C...       M        - number of columns of the matrix.
00292 C...
00293 C...   Output:
00294 C...       IAT, JAT, AT - transposed matrix
00295 C...
00296 C...   Note:
00297 C...       N+1 is the dimension of IA.
00298 C...       M+1 is the dimension of IAT.
00299 C-----
00300 */

```

```

00301     n=*na;
00302     m=*ma;
00303     mh = m + 1;
00304     nh = n + 1;
00305
00306     for (i = 1; i < mh; ++i) {
00307         iat[i] = 0;
00308     }
00309     iab = ia[nh-1] - 1; /* Size of ja */
00310     for (i = 1; i <= iab; ++i) {
00311         j = ja[i-1] + 2;
00312         if (j <= mh) {
00313             iat[j-1] = iat[j-1] + 1;
00314         }
00315     }
00316     iat[0] = 1;
00317     iat[1] = 1;
00318     if (m != 1) {
00319         for (i = 2; i < mh; ++i) {
00320             iat[i] = iat[i] + iat[i-1];
00321         }
00322     }
00323
00324     for (i=1; i<=n; ++i) {
00325         iaa = ia[i-1];
00326         iab = ia[i] - 1;
00327         if (iab >= iaa) {
00328             for (jp = iaa; jp <= iab; ++jp) {
00329                 j = ja[jp-1] + 1;
00330                 k = iat[j-1];
00331                 jat[k-1] = i;
00332                 at[k-1] = a[jp-1];
00333                 iat[j-1] = k + 1;
00334             }
00335         }
00336     }
00337
00338     return;
00339 }
00340
00359 void fasp_sparse_aplbms_ (INT *ia,
00360                          INT *ja,
00361                          INT *ib,
00362                          INT *jb,
00363                          INT *nab,
00364                          INT *mab,
00365                          INT *ic,
00366                          INT *jc)
00367 {
00368     unsigned int jcform=0;
00369     INT icpp, il, i, j, jp, n, m, iastrt, iaend, ibstrt, ibend;
00370     INT *icp;
00371     /*
00372 c... addition of two general sparse matrices (symbolic part) :
00373 c= a + b.
00374 */
00375     if (jc) jcform=1;
00376     n=*nab;
00377     m=*mab;
00378     icp=(INT *) calloc(m,sizeof(INT));
00379     for (i=0; i< m; ++i) icp[i] = 0;
00380     icpp = 1;
00381     for (i=0; i< n; ++i) {
00382         ic[i] = icpp;
00383         il=i+1;
00384         iastrt = ia[i]-1;
00385         iaend = ia[i+1]-1;
00386         if (iaend > iastrt) {
00387             for (jp = iastrt; jp < iaend; ++jp) {
00388                 j = ja[jp];
00389                 if (jcform) jc[icpp-1] = j;
00390                 ++icpp;
00391                 icp[j-1] = il;
00392             }
00393         }
00394         ibstrt = ib[i] - 1;
00395         ibend = ib[i+1] - 1;
00396         if (ibend > ibstrt) {
00397             for (jp = ibstrt; jp < ibend; ++jp) {
00398                 j = jb[jp];
00399                 if (icp[j-1] != il) {

```

```

00400             if (jcform) jc[icpp-1] = j;
00401             ++icpp;
00402         }
00403     }
00404 }
00405 } // // loop i=0; i< n
00406 ic[n] = icpp;
00407 if (icp) free(icp);
00408 return;
00409 }
00410
00431 void fasp_sparse_aplusb_ (INT *ia,
00432                          INT *ja,
00433                          REAL *a,
00434                          INT *ib,
00435                          INT *jb,
00436                          REAL *b,
00437                          INT *nab,
00438                          INT *mab,
00439                          INT *ic,
00440                          INT *jc,
00441                          REAL *c)
00442 {
00443     INT n,m,icpp,il,i,j,iastrt,iaend,ibstrt,ibend,icstrt,icend;
00444     REAL *x;
00445     /*
00446     c... addition of two general sparse matrices (numerical part) :
00447     c= a + b
00448     */
00449     n=*nab;
00450     m=*mab;
00451     x=(REAL *)calloc(m,sizeof(REAL));
00452     for (i=0;i<n;++i) {
00453         il=i+1;
00454         icstrt = ic[i]-1;
00455         icend = ic[il]-1;
00456         if (icend > icstrt) {
00457             for (icpp = icstrt;icpp<icend;++icpp) {
00458                 j=jc[icpp]-1;
00459                 x[j] = 0e+00;
00460             }
00461             iastrt = ia[i]-1;
00462             iaend = ia[il]-1;
00463             if (iaend > iastrt) {
00464                 for (icpp = iastrt;icpp<iaend;++icpp) {
00465                     j=ja[icpp]-1;
00466                     x[j] = a[icpp];
00467                 }
00468             }
00469             ibstrt = ib[i]-1;
00470             ibend = ib[il]-1;
00471             if (ibend > ibstrt) {
00472                 for (icpp = ibstrt;icpp<ibend;++icpp) {
00473                     j = jb[icpp]-1;
00474                     x[j] = x[j] + b[icpp];
00475                 }
00476             }
00477             for (icpp = icstrt;icpp<icend;++icpp) {
00478                 j=jc[icpp]-1;
00479                 c[icpp] = x[j];
00480             }
00481         } // if (icstrt > icend)...
00482     } // loop i=0; i< n
00483     if (x) free(x);
00484     return;
00485 }
00486
00515 void fasp_sparse_rapms_ (INT *ir,
00516                         INT *jr,
00517                         INT *ia,
00518                         INT *ja,
00519                         INT *ip,
00520                         INT *jp,
00521                         INT *nin,
00522                         INT *ncin,
00523                         INT *iac,
00524                         INT *jac,
00525                         INT *maxrout)
00526 {
00527     INT i,jk,jak,jpk,ic,jc,nc,icpl,ira,irb,ipa,ipb;
00528     INT maxri,maxr,iaa,iab,iacp,ifl,jf1,jacform=0;

```

```

00529     INT *ix;
00530
00531     nc = *ncin;
00532     ix=(INT *) calloc(nc,sizeof(INT));
00533     if (jac) jacform=1;
00534     maxr = 0;
00535     for (i =0;i<nc; ++i) {
00536         ix[i]=0;
00537         ira=ir[i];
00538         irb=ir[i+1];
00539         maxri=irb-ira;
00540         if (maxr < maxri) maxr=maxri;
00541     }
00542     iac[0] = 1;
00543     iacp = iac[0]-1;
00544     for (ic = 0;ic<nc;ic++) {
00545         ira=ir[ic]-1;
00546         icpl=ic+1;
00547         irb=ir[icpl]-1;
00548         for (jk = ira;jk<irb;jk++) {
00549             ifl = jr[jk]-1;
00550             iaa = ia[ifl]-1;
00551             iab = ia[ifl+1]-1;
00552             for (jak = iaa;jak < iab;jak++) {
00553                 jfl = ja[jak]-1;
00554                 ipa = ip[jfl]-1;
00555                 ipb = ip[jfl+1]-1;
00556                 for (jpk = ipa;jpk < ipb;jpk++) {
00557                     jc = jp[jpk]-1;
00558                     if (ix[jc] != icpl) {
00559                         ix[jc]=icpl;
00560                         if (jacform) jac[iacp] = jc+1;
00561                         iacp++;
00562                     }
00563                 }
00564             }
00565         }
00566         iac[icpl] = iacp+1;
00567     }
00568     *maxrout=maxr;
00569     if (ix) free(ix);
00570     return;
00571 }
00572
00596 void fasp_sparse_wtams_ (INT *jw,
00597                          INT *ia,
00598                          INT *ja,
00599                          INT *nwp,
00600                          INT *map,
00601                          INT *jv,
00602                          INT *nvp,
00603                          INT *icp)
00604 {
00605     INT nw,nv,iastrt,iaend,j,k,jiw,jia;
00606     if (*nwp<=0) {*nvp=0; return;}
00607     nw=*nwp;
00608     nv = 0;
00609     for (jiw = 0;jiw < nw; ++jiw) {
00610         j = jw[jiw]-1;
00611         iastrt = ia[j]-1;
00612         iaend = ia[j+1]-1;
00613         if (iaend > iastrt) {
00614             for (jia = iastrt ;jia< iaend;jia++) {
00615                 k = ja[jia]-1;
00616                 if (!icp[k]) {
00617                     jv[nv] = k+1;
00618                     nv++;
00619                     icp[k] = nv;
00620                 }
00621             }
00622         }
00623     }
00624     *nvp=nv;
00625     return;
00626 }
00627
00648 void fasp_sparse_wta_ (INT *jw,
00649                      REAL *w,
00650                      INT *ia,
00651                      INT *ja,
00652                      REAL *a,

```

```

00653             INT *nwp,
00654             INT *map,
00655             INT *jv,
00656             REAL *v,
00657             INT *nvp)
00658 {
00659     INT nw,nv,iastrt,iaend,j,k,ji,jiw,jia;
00660     REAL v0;
00661
00662     if (*nwp<=0) {*nvp=-1; return;}
00663     nw=*nwp;
00664     nv=*nvp;
00665     for (ji = 0;ji < nv;++ji) {
00666         k=jv[ji]-1;
00667         v[k] = 0e+0;
00668     }
00669     for (jiw = 0;jiw<nw; ++jiw) {
00670         j = jw[jiw]-1;
00671         v0 = w[jiw];
00672         iastrt = ia[j]-1;
00673         iaend = ia[j+1]-1;
00674         if (iaend > iastrt) {
00675             for (jia = iastrt;jia < iaend;jia++) {
00676                 k = ja[jia]-1;
00677                 v[k] += v0*a[jia];
00678             }
00679         } // end if
00680     } // end for
00681     return;
00682 }
00683
00699 void fasp_sparse_ytxbig_ (INT *jy,
00700                          REAL *y,
00701                          INT *nyp,
00702                          REAL *x,
00703                          REAL *s)
00704 {
00705     INT i,ii;
00706     *s=0e+00;
00707     if (*nyp > 0) {
00708         for (i = 0;i< *nyp; ++i) {
00709             ii = jy[i]-1;
00710             *s += y[i]*x[ii];
00711         }
00712     }
00713     return;
00714 }
00715
00733 void fasp_sparse_ytx_ (INT *jy,
00734                      REAL *y,
00735                      INT *jx,
00736                      REAL *x,
00737                      INT *nyp,
00738                      INT *nxp,
00739                      INT *icp,
00740                      REAL *s)
00741 { // not tested
00742     INT i,j,i0,ii;
00743     *s=0e+00;
00744     if ((*nyp > 0) && (*nxp > 0)) {
00745         for (i = 0;i< *nyp; ++i) {
00746             j = jy[i]-1;
00747             i0=icp[j];
00748             if (i0) {
00749                 ii=jx[i0]-1;
00750                 *s += y[i]*x[ii];
00751             }
00752         }
00753     }
00754     return;
00755 }
00756
00787 void fasp_sparse_rapcmp_ (INT *ir,
00788                        INT *jr,
00789                        REAL *r,
00790                        INT *ia,
00791                        INT *ja,
00792                        REAL *a,
00793                        INT *ipt,
00794                        INT *jpt,
00795                        REAL *pt,

```

```

00796             INT *nin,
00797             INT *ncin,
00798             INT *iac,
00799             INT *jac,
00800             REAL *ac,
00801             INT *idummy)
00802 {
00803     INT i,j,k,n,nc,nv,nw,nptjc,iacst,iacen,ic,jc,is,js,jkc,iastrt,iaend,ji,jia;
00804     REAL aij,v0;
00805     INT *icp=NULL, *jv=NULL,*jris=NULL, *jptjs=NULL;
00806     REAL *v=NULL, *ris=NULL, *ptjs=NULL;
00807     n=*nin;
00808     nc=*ncin;
00809
00810     v = (REAL *) calloc(n,sizeof(REAL));
00811     icp = (INT *) calloc(n,sizeof(INT));
00812     jv = (INT *) calloc(n,sizeof(INT));
00813     if (!(icp && v && jv)) {
00814         fprintf(stderr,"### ERROR: Could not allocate memory!\n");
00815         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00816     }
00817     for (i=0;i<n;++i) {
00818         icp[i] = 0;
00819         jv[i] = 0;
00820         v[i]=0e+00;
00821     }
00822     for (ic = 0;ic<nc;ic++) {
00823         nw = ir[ic+1]-ir[ic];
00824         if (nw<=0) continue;
00825         is = ir[ic]-1;
00826         jris=jr+is;
00827         // wtams_(jris,ia, ja, &nw,&n,jv, &nv, icp);
00828         // void wtams_(INT *jw,INT *ia, INT *ja, INT *nwp,INT *map,
00829         // INT *jv, INT *nvp, INT *icp)
00830         // INT nw,ma,nv,iastrt,iaend,i,j,k,ji,jia;
00831         nv = 0;
00832         for (ji = 0;ji < nw; ++ji) {
00833             j = *(jris+ji)-1;
00834             iastrt = ia[j]-1;
00835             iaend = ia[j+1]-1;
00836             if (iaend > iastrt) {
00837                 for (jia = iastrt ;jia< iaend;jia++) {
00838                     k = ja[jia]-1;
00839                     if (!icp[k]) {
00840                         *(jv+nv) = k+1;
00841                         nv++;
00842                         icp[k] = nv;
00843                     } //end if
00844                 } //end for
00845             } //end if
00846         } //end for loop for forming the nonz struct of (r_i)^t*A
00847         ris=r+is;
00848         // wta_(jris, ris,ia, ja, a,&nw, &n, jv, v, &nv);
00849         for (ji = 0;ji < nv;++ji) {
00850             k=jv[ji]-1;
00851             v[k] = 0e+00;
00852         }
00853         for (ji = 0;ji<nw ; ++ji) {
00854             j = *(jris+ji)-1;
00855             v0 = *(ris+ji);
00856             iastrt = ia[j]-1;
00857             iaend = ia[j+1]-1;
00858             if (iaend > iastrt) {
00859                 for (jia = iastrt;jia < iaend;jia++) {
00860                     k = ja[jia]-1;
00861                     v[k] += v0*a[jia];
00862                 }
00863             } // end if
00864         } //end for loop for calculating the product (r_i)^t*A
00865         iacst=iac[ic]-1;
00866         iacen=iac[ic+1]-1;
00867         for (jkc = iacst; jkc<iacen;jkc++) {
00868             jc = jac[jkc]-1;
00869             nptjc = ipt[jc+1]-ipt[jc];
00870             js = ipt[jc]-1;
00871             jptjs = jpt+js;
00872             ptjs = pt+js;
00873             // ytxbig_(jptjs,ptjs,&nptjc,v,&aij);
00874             aij=0e+00;
00875             if (nptjc > 0) {
00876                 for (i = 0;i< nptjc; ++i) {

```

```

00877             j = *(jptjs+i)-1;
00878             aij += (*(ptjs+i))*(*(v+j));
00879         } //end for
00880     } //end if
00881     ac[jkc] = aij;
00882 } //end for
00883 // set nos the values of v and icp back to 0;
00884 for (i=0; i < nv; ++i) {
00885     j=jv[i]-1;
00886     icp[j]=0;
00887     v[j]=0e+00;
00888 } //end for
00889 } //end for
00890
00891 if (v) free(v);
00892 if (icp) free(icp);
00893 if (jv) free(jv);
00894
00895 return;
00896 }
00897
00907 ivector fasp_sparse_mis (dCSRmat *A)
00908 {
00909     // information of A
00910     INT n = A->row;
00911     INT *IA = A->IA;
00912     INT *JA = A->JA;
00913
00914     // local variables
00915     INT i,j;
00916     INT row_begin, row_end;
00917     INT count=0;
00918     INT *flag;
00919     flag = (INT *)fasp_mem_calloc(n, sizeof(INT));
00920     //for (i=0;i<n;i++) flag[i]=0;
00921     memset(flag, 0, sizeof(INT)*n);
00922
00923     // work space
00924     INT *work = (INT*)fasp_mem_calloc(n,sizeof(INT));
00925
00926     // return vector
00927     ivector MIS;
00928
00929     // main loop
00930     for (i=0;i<n;i++) {
00931         if (flag[i] == 0) {
00932             flag[i] = 1;
00933             row_begin = IA[i] - 1; row_end = IA[i+1] - 1;
00934             for (j = row_begin; j<row_end; j++) {
00935                 if (flag[JA[j]-1] > 0) {
00936                     flag[i] = -1;
00937                     break;
00938                 }
00939             }
00940             if (flag[i]) {
00941                 work[count] = i; count++;
00942                 for (j = row_begin; j<row_end; j++) {
00943                     flag[JA[j]-1] = -1;
00944                 }
00945             }
00946         } // end if
00947     } // end for
00948
00949     // form MIS
00950     MIS.row = count;
00951     work = (INT *)fasp_mem_realloc(work, count*sizeof(INT));
00952     MIS.val = work;
00953
00954     // clean
00955     fasp_mem_free(flag); flag = NULL;
00956
00957     //return
00958     return MIS;
00959 }
00960
00961 /*-----*/
00962 /*--      End of File      --*/
00963 /*-----*/

```

9.87 BlaSpmvBLC.c File Reference

Linear algebraic operations for [dBLCmat](#) matrices.

```
#include <time.h>
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"
```

Functions

- void [fasp_blas_dblc_aApy](#) (const [REAL](#) alpha, const [dBLCmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha * A * x + y$.*
- void [fasp_blas_dblc_mxv](#) (const [dBLCmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = A * x$.*
- void [fasp_blas_ldblc_aApy](#) (const [REAL](#) alpha, const [dBLCmat](#) *A, const [LONGREAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha * A * x + y$.*

9.87.1 Detailed Description

Linear algebraic operations for [dBLCmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [BlaSpmvCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSpmvBLC.c](#).

9.87.2 Function Documentation

9.87.2.1 [fasp_blas_dblc_aApy\(\)](#)

```
void fasp_blas_dblc_aApy (
    const REAL alpha,
    const dBLCmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = \alpha * A * x + y$.

Parameters

<i>alpha</i>	REAL factor a
<i>A</i>	Pointer to dBLCmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Xiaozhe Hu

Date

06/04/2010

Definition at line 38 of file BlasSpmvBLC.c.

9.87.2.2 fasp_blas_dblc_mxv()

```
void fasp_blas_dblc_mxv (
    const dBLCmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = A*x$.**Parameters**

<i>A</i>	Pointer to dBLCmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Chensong Zhang

Date

04/27/2013

Definition at line 164 of file BlasSpmvBLC.c.

9.87.2.3 fasp_blas_ldblc_aApy()

```
void fasp_blas_ldblc_aApy (
    const REAL alpha,
    const dBLCmat * A,
    const LONGREAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = \alpha*A*x + y$.**Parameters**

<i>alpha</i>	REAL factor a
<i>A</i>	Pointer to dBLCmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Lai Ting

Date

08/01/2022

Definition at line 296 of file [BlaSpmvBLC.c](#).**9.88 BlaSpmvBLC.c**[Go to the documentation of this file.](#)

```

00001
00014 #include <time.h>
00015
00016 #include "fasp.h"
00017 #include "fasp_block.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/
00021 /*--      Public Functions      --*/
00022 /*-----*/
00023
00038 void fasp_blas_dblc_aApy (const REAL      alpha,
00039                          const dBLCmat    *A,
00040                          const REAL      *x,
00041                          REAL            *y)
00042 {
00043     // information of A
00044     const INT brow = A->brow;
00045
00046     // local variables
00047     register dCSRmat *A11, *A12, *A21, *A22;
00048     register dCSRmat *A13, *A23, *A31, *A32, *A33;
00049
00050     INT row1, col1;
00051     INT row2, col2;
00052
00053     const register REAL *x1, *x2, *x3;
00054     register REAL      *y1, *y2, *y3;
00055
00056     INT i, j;
00057     INT start_row, start_col;
00058
00059     switch (brow) {
00060
00061         case 2:
00062             A11 = A->blocks[0];
00063             A12 = A->blocks[1];
00064             A21 = A->blocks[2];
00065             A22 = A->blocks[3];
00066
00067             row1 = A11->row;
00068             col1 = A11->col;
00069
00070             x1 = x;
00071             x2 = &(x[col1]);
00072             y1 = y;
00073             y2 = &(y[row1]);
00074
00075             // y1 = alpha*A11*x1 + alpha*A12*x2 + y1
00076             if (A11) fasp_blas_dcsr_aApy(alpha, A11, x1, y1);
00077             if (A12) fasp_blas_dcsr_aApy(alpha, A12, x2, y1);
00078
00079             // y2 = alpha*A21*x1 + alpha*A22*x2 + y2
00080             if (A21) fasp_blas_dcsr_aApy(alpha, A21, x1, y2);
00081             if (A22) fasp_blas_dcsr_aApy(alpha, A22, x2, y2);
00082
00083             break;
00084
00085         case 3:
00086             A11 = A->blocks[0];
00087             A12 = A->blocks[1];
00088             A13 = A->blocks[2];
00089             A21 = A->blocks[3];

```

```

00090         A22 = A->blocks[4];
00091         A23 = A->blocks[5];
00092         A31 = A->blocks[6];
00093         A32 = A->blocks[7];
00094         A33 = A->blocks[8];
00095
00096         row1 = A11->row;
00097         col1 = A11->col;
00098         row2 = A22->row;
00099         col2 = A22->col;
00100
00101         x1 = x;
00102         x2 = &(x[col1]);
00103         x3 = &(x[col1+col2]);
00104         y1 = y;
00105         y2 = &(y[row1]);
00106         y3 = &(y[row1+row2]);
00107
00108         // y1 = alpha*A11*x1 + alpha*A12*x2 + alpha*A13*x3 + y1
00109         if (A11) fasp_blas_dcsr_aAxy(alpha, A11, x1, y1);
00110         if (A12) fasp_blas_dcsr_aAxy(alpha, A12, x2, y1);
00111         if (A13) fasp_blas_dcsr_aAxy(alpha, A13, x3, y1);
00112
00113         // y2 = alpha*A21*x1 + alpha*A22*x2 + alpha*A23*x3 + y2
00114         if (A21) fasp_blas_dcsr_aAxy(alpha, A21, x1, y2);
00115         if (A22) fasp_blas_dcsr_aAxy(alpha, A22, x2, y2);
00116         if (A23) fasp_blas_dcsr_aAxy(alpha, A23, x3, y2);
00117
00118         // y3 = alpha*A31*x1 + alpha*A32*x2 + alpha*A33*x3 + y2
00119         if (A31) fasp_blas_dcsr_aAxy(alpha, A31, x1, y3);
00120         if (A32) fasp_blas_dcsr_aAxy(alpha, A32, x2, y3);
00121         if (A33) fasp_blas_dcsr_aAxy(alpha, A33, x3, y3);
00122
00123         break;
00124
00125     default:
00126
00127         start_row = 0;
00128         start_col = 0;
00129
00130         for (i=0; i<brow; i++) {
00131
00132             for (j=0; j<brow; j++) {
00133
00134                 if (A->blocks[i*brow+j]) {
00135                     fasp_blas_dcsr_aAxy(alpha, A->blocks[i*brow+j],
00136                                         &(x[start_col]), &(y[start_row]));
00137                 }
00138                 start_col = start_col + A->blocks[j*brow+j]->col;
00139             }
00140
00141             start_row = start_row + A->blocks[i*brow+i]->row;
00142             start_col = 0;
00143         }
00144
00145         break;
00146     } // end of switch
00147
00148 } // end of function
00149
00150 }
00151
00164 void fasp_blas_dblc_mnv (const dBLCmat *A,
00165                         const REAL *x,
00166                         REAL *y)
00167 {
00168     // information of A
00169     const INT brow = A->brow;
00170
00171     // local variables
00172     register dCSRmat *A11, *A12, *A21, *A22;
00173     register dCSRmat *A13, *A23, *A31, *A32, *A33;
00174
00175     INT row1, col1;
00176     INT row2, col2;
00177
00178     const register REAL *x1, *x2, *x3;
00179     register REAL *y1, *y2, *y3;
00180
00181     INT i, j;
00182     INT start_row, start_col;

```

```

00183
00184     switch (brow) {
00185
00186         case 2:
00187             A11 = A->blocks[0];
00188             A12 = A->blocks[1];
00189             A21 = A->blocks[2];
00190             A22 = A->blocks[3];
00191
00192             row1 = A11->row;
00193             col1 = A11->col;
00194
00195             x1 = x;
00196             x2 = &(x[col1]);
00197             y1 = y;
00198             y2 = &(y[row1]);
00199
00200             // y1 = A11*x1 + A12*x2
00201             if (A11) fasp_blas_dcsr_mxv(A11, x1, y1);
00202             if (A12) fasp_blas_dcsr_aAxy(1.0, A12, x2, y1);
00203
00204             // y2 = A21*x1 + A22*x2
00205             if (A21) fasp_blas_dcsr_mxv(A21, x1, y2);
00206             if (A22) fasp_blas_dcsr_aAxy(1.0, A22, x2, y2);
00207
00208             break;
00209
00210         case 3:
00211             A11 = A->blocks[0];
00212             A12 = A->blocks[1];
00213             A13 = A->blocks[2];
00214             A21 = A->blocks[3];
00215             A22 = A->blocks[4];
00216             A23 = A->blocks[5];
00217             A31 = A->blocks[6];
00218             A32 = A->blocks[7];
00219             A33 = A->blocks[8];
00220
00221             row1 = A11->row;
00222             col1 = A11->col;
00223             row2 = A22->row;
00224             col2 = A22->col;
00225
00226             x1 = x;
00227             x2 = &(x[col1]);
00228             x3 = &(x[col1+col2]);
00229             y1 = y;
00230             y2 = &(y[row1]);
00231             y3 = &(y[row1+row2]);
00232
00233             // y1 = A11*x1 + A12*x2 + A13*x3 + y1
00234             if (A11) fasp_blas_dcsr_mxv(A11, x1, y1);
00235             if (A12) fasp_blas_dcsr_aAxy(1.0, A12, x2, y1);
00236             if (A13) fasp_blas_dcsr_aAxy(1.0, A13, x3, y1);
00237
00238             // y2 = A21*x1 + A22*x2 + A23*x3 + y2
00239             if (A21) fasp_blas_dcsr_mxv(A21, x1, y2);
00240             if (A22) fasp_blas_dcsr_aAxy(1.0, A22, x2, y2);
00241             if (A23) fasp_blas_dcsr_aAxy(1.0, A23, x3, y2);
00242
00243             // y3 = A31*x1 + A32*x2 + A33*x3 + y2
00244             if (A31) fasp_blas_dcsr_mxv(A31, x1, y3);
00245             if (A32) fasp_blas_dcsr_aAxy(1.0, A32, x2, y3);
00246             if (A33) fasp_blas_dcsr_aAxy(1.0, A33, x3, y3);
00247
00248             break;
00249
00250         default:
00251
00252             start_row = 0;
00253             start_col = 0;
00254
00255             for (i=0; i<brow; i++) {
00256
00257                 for (j=0; j<brow; j++){
00258
00259                     if (j==0) {
00260                         if (A->blocks[i*brow+j]){
00261                             fasp_blas_dcsr_mxv(A->blocks[i*brow+j], &(x[start_col]), &(y[start_row]));
00262                         }
00263                     }

```

```

00264         else {
00265             if (A->blocks[i*brow+j]){
00266                 fasp_blas_dcsr_aApy(1.0, A->blocks[i*brow+j], &(x[start_col]),
&(y[start_row]));
00267             }
00268         }
00269         start_col = start_col + A->blocks[j*brow+j]->col;
00270     }
00271
00272     start_row = start_row + A->blocks[i*brow+i]->row;
00273     start_col = 0;
00274 }
00275
00276     break;
00277
00278 } // end of switch
00279
00280 }
00281
00296 void fasp_blas_ldblc_aApy (const REAL      alpha,
00297                             const dBLCmat    *A,
00298                             const LONGREAL   *x,
00299                             REAL             *y)
00300 {
00301     // information of A
00302     const INT brow = A->brow;
00303
00304     // local variables
00305     register dCSRmat *A11, *A12, *A21, *A22;
00306     register dCSRmat *A13, *A23, *A31, *A32, *A33;
00307
00308     INT row1, col1;
00309     INT row2, col2;
00310
00311     const register LONGREAL *x1, *x2, *x3;
00312     register REAL          *y1, *y2, *y3;
00313
00314     INT i,j;
00315     INT start_row, start_col;
00316
00317     switch (brow) {
00318
00319         case 2:
00320             A11 = A->blocks[0];
00321             A12 = A->blocks[1];
00322             A21 = A->blocks[2];
00323             A22 = A->blocks[3];
00324
00325             row1 = A11->row;
00326             col1 = A11->col;
00327
00328             x1 = x;
00329             x2 = &(x[col1]);
00330             y1 = y;
00331             y2 = &(y[row1]);
00332
00333             // y1 = alpha*A11*x1 + alpha*A12*x2 + y1
00334             if (A11) fasp_blas_ldcsr_aApy(alpha, A11, x1, y1);
00335             if (A12) fasp_blas_ldcsr_aApy(alpha, A12, x2, y1);
00336
00337             // y2 = alpha*A21*x1 + alpha*A22*x2 + y2
00338             if (A21) fasp_blas_ldcsr_aApy(alpha, A21, x1, y2);
00339             if (A22) fasp_blas_ldcsr_aApy(alpha, A22, x2, y2);
00340
00341             break;
00342
00343         case 3:
00344             A11 = A->blocks[0];
00345             A12 = A->blocks[1];
00346             A13 = A->blocks[2];
00347             A21 = A->blocks[3];
00348             A22 = A->blocks[4];
00349             A23 = A->blocks[5];
00350             A31 = A->blocks[6];
00351             A32 = A->blocks[7];
00352             A33 = A->blocks[8];
00353
00354             row1 = A11->row;
00355             col1 = A11->col;
00356             row2 = A22->row;
00357             col2 = A22->col;

```

```

00358
00359         x1 = x;
00360         x2 = &(x[col1]);
00361         x3 = &(x[col1+col2]);
00362         y1 = y;
00363         y2 = &(y[row1]);
00364         y3 = &(y[row1+row2]);
00365
00366         // y1 = alpha*A11*x1 + alpha*A12*x2 + alpha*A13*x3 + y1
00367         if (A11) fasp_blas_ldcsr_aAxy(alpha, A11, x1, y1);
00368         if (A12) fasp_blas_ldcsr_aAxy(alpha, A12, x2, y1);
00369         if (A13) fasp_blas_ldcsr_aAxy(alpha, A13, x3, y1);
00370
00371         // y2 = alpha*A21*x1 + alpha*A22*x2 + alpha*A23*x3 + y2
00372         if (A21) fasp_blas_ldcsr_aAxy(alpha, A21, x1, y2);
00373         if (A22) fasp_blas_ldcsr_aAxy(alpha, A22, x2, y2);
00374         if (A23) fasp_blas_ldcsr_aAxy(alpha, A23, x3, y2);
00375
00376         // y3 = alpha*A31*x1 + alpha*A32*x2 + alpha*A33*x3 + y2
00377         if (A31) fasp_blas_ldcsr_aAxy(alpha, A31, x1, y3);
00378         if (A32) fasp_blas_ldcsr_aAxy(alpha, A32, x2, y3);
00379         if (A33) fasp_blas_ldcsr_aAxy(alpha, A33, x3, y3);
00380
00381         break;
00382
00383     default:
00384
00385         start_row = 0;
00386         start_col = 0;
00387
00388         for (i=0; i<brow; i++) {
00389
00390             for (j=0; j<brow; j++) {
00391
00392                 if (A->blocks[i*brow+j]) {
00393                     fasp_blas_ldcsr_aAxy(alpha, A->blocks[i*brow+j],
00394                                         &(x[start_col]), &(y[start_row]));
00395                 }
00396                 start_col = start_col + A->blocks[j*brow+j]->col;
00397             }
00398
00399             start_row = start_row + A->blocks[i*brow+i]->row;
00400             start_col = 0;
00401         }
00402
00403         break;
00404
00405     } // end of switch
00406 } // end of function
00407 }
00408
00409 /*-----*/
00410 /*--      End of File      --*/
00411 /*-----*/

```

9.89 BlaSpmvBSR.c File Reference

Linear algebraic operations for [dBSRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_blas_dbsr_axm](#) ([dBSRmat](#) *A, const [REAL](#) alpha)
Multiply a sparse matrix A in BSR format by a scalar alpha.
- void [fasp_blas_dbsr_aAxpby](#) (const [REAL](#) alpha, [dBSRmat](#) *A, [REAL](#) *x, const [REAL](#) beta, [REAL](#) *y)
Compute $y := \alpha A * x + \beta y$.
- void [fasp_blas_dbsr_aAxy](#) (const [REAL](#) alpha, const [dBSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)

- Compute $y := \alpha * A * x + y$.*
- void `fasp_blas_dbsr_aAxy_agg` (const [REAL](#) alpha, const [dBSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
 - Compute $y := \alpha * A * x + y$ where each small block matrix is an identity matrix.*
- void `fasp_blas_dbsr_mxv` (const [dBSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
 - Compute $y := A * x$.*
- void `fasp_blas_dbsr_mxv_agg` (const [dBSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
 - Compute $y := A * x$, where each small block matrices of A is an identity.*
- void `fasp_blas_dbsr_mxm` (const [dBSRmat](#) *A, const [dBSRmat](#) *B, [dBSRmat](#) *C)
 - Sparse matrix multiplication $C = A * B$.*
- void `fasp_blas_dbsr_rap1` (const [dBSRmat](#) *R, const [dBSRmat](#) *A, const [dBSRmat](#) *P, [dBSRmat](#) *B)
 - [dBSRmat](#) sparse matrix multiplication $B = R * A * P$*
- void `fasp_blas_dbsr_rap` (const [dBSRmat](#) *R, const [dBSRmat](#) *A, const [dBSRmat](#) *P, [dBSRmat](#) *B)
 - [dBSRmat](#) sparse matrix multiplication $B = R * A * P$*
- void `fasp_blas_dbsr_rap_agg` (const [dBSRmat](#) *R, const [dBSRmat](#) *A, const [dBSRmat](#) *P, [dBSRmat](#) *B)
 - [dBSRmat](#) sparse matrix multiplication $B = R * A * P$, where small block matrices in P and R are identity matrices!*

9.89.1 Detailed Description

Linear algebraic operations for [dBSRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaSmallMat.c](#), and [BlaArray.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSpmvBSR.c](#).

9.89.2 Function Documentation

9.89.2.1 `fasp_blas_dbsr_aAxyby()`

```
void fasp_blas_dbsr_aAxyby (
    const REAL alpha,
    dBSRmat * A,
    REAL * x,
    const REAL beta,
    REAL * y )
```

Compute $y := \alpha * A * x + \beta * y$.

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to the dBSRmat matrix
<i>x</i>	Pointer to the array x
<i>beta</i>	REAL factor beta
<i>y</i>	Pointer to the array y

Author

Zhiyang Zhou

Date

10/25/2010

Modified by Chunsheng Feng, Zheng Li on 06/29/2012

Note

Works for general nb (Xiaozhe)

Definition at line 67 of file [BlaSpmvBSR.c](#).**9.89.2.2 fasp_blas_dbsr_aApy()**

```
void fasp_blas_dbsr_aApy (
    const REAL alpha,
    const dBSRmat * A,
    const REAL * x,
    REAL * y )
```

Compute $y := \alpha A * x + y$.**Parameters**

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to the dBSRmat matrix
<i>x</i>	Pointer to the array x
<i>y</i>	Pointer to the array y

Author

Zhiyang Zhou

Date

10/25/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Note

Works for general nb (Xiaozhe)

Definition at line 348 of file [BlaSpmvBSR.c](#).**9.89.2.3 fasp_blas_dbsr_aApy_agg()**

```
void fasp_blas_dbsr_aApy_agg (
    const REAL alpha,
    const dBSRmat * A,
    const REAL * x,
    REAL * y )
```

Compute $y := \alpha A * x + y$ where each small block matrix is an identity matrix.

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to the dBSRmat matrix
<i>x</i>	Pointer to the array x
<i>y</i>	Pointer to the array y

Author

Xiaozhe Hu

Date

01/02/2014

Note

Works for general nb (Xiaozhe)

Definition at line [624](#) of file [BlasSpmvBSR.c](#).

9.89.2.4 fasp_blas_dbsr_axm()

```
void fasp_blas_dbsr_axm (  
    dBSRmat * A,  
    const REAL alpha )
```

Multiply a sparse matrix A in BSR format by a scalar alpha.

Parameters

<i>A</i>	Pointer to dBSRmat matrix A
<i>alpha</i>	REAL factor alpha

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line [38](#) of file [BlasSpmvBSR.c](#).

9.89.2.5 fasp_blas_dbsr_mxm()

```
void fasp_blas_dbsr_mxm (  
    const dBSRmat * A,  
    const dBSRmat * B,  
    dBSRmat * C )
```

Sparse matrix multiplication $C=A*B$.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix A
<i>B</i>	Pointer to the dBSRmat matrix B
<i>C</i>	Pointer to dBSRmat matrix equal to A*B

Author

Xiaozhe Hu

Date

05/26/2014

Note

This fct will be replaced! – Xiaozhe

Definition at line [4646](#) of file [BlaSpmvBSR.c](#).

9.89.2.6 fasp_blas_dbsr_m xv()

```
void fasp_blas_dbsr_m xv (
    const dBSRmat * A,
    const REAL * x,
    REAL * y )
```

Compute $y := A*x$.

Parameters

<i>A</i>	Pointer to the dBSRmat matrix
<i>x</i>	Pointer to the array x
<i>y</i>	Pointer to the array y

Author

Zhiyang Zhou

Date

10/25/2010

Note

Works for general nb (Xiaozhe)

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line [910](#) of file [BlaSpmvBSR.c](#).

9.89.2.7 fasp_blas_dbsr_m xv_agg()

```
void fasp_blas_dbsr_m xv_agg (
    const dBSRmat * A,
```

```
const REAL * x,
REAL * y )
```

Compute $y := A*x$, where each small block matrices of A is an identity.

Parameters

A	Pointer to the dBSRmat matrix
x	Pointer to the array x
y	Pointer to the array y

Author

Xiaozhe Hu

Date

01/02/2014

Note

Works for general nb (Xiaozhe)

Definition at line [2697](#) of file [BlaspmvBSR.c](#).

9.89.2.8 fasp_blas_dbsr_rap()

```
void fasp_blas_dbsr_rap (
    const dBSRmat * R,
    const dBSRmat * A,
    const dBSRmat * P,
    dBSRmat * B )
```

[dBSRmat](#) sparse matrix multiplication $B=R*A*P$

Parameters

R	Pointer to the dBSRmat matrix
A	Pointer to the dBSRmat matrix
P	Pointer to the dBSRmat matrix
B	Pointer to dBSRmat matrix equal to $R*A*P$ (output)

Author

Xiaozhe Hu, Chunsheng Feng, Zheng Li

Date

10/24/2012

Note

Ref. R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. Advances in Computational Mathematics, 1 (1993), pp. 127-137.

Definition at line [4961](#) of file [BlaspmvBSR.c](#).

9.89.2.9 fasp_blas_dbsr_rap1()

```
void fasp_blas_dbsr_rap1 (
    const dBSRmat * R,
    const dBSRmat * A,
    const dBSRmat * P,
    dBSRmat * B )
```

dBSRmat sparse matrix multiplication $B=R*A*P$

Parameters

<i>R</i>	Pointer to the dBSRmat matrix
<i>A</i>	Pointer to the dBSRmat matrix
<i>P</i>	Pointer to the dBSRmat matrix
<i>B</i>	Pointer to dBSRmat matrix equal to $R*A*P$ (output)

Author

Chunsheng Feng, Xiaoqiang Yue and Xiaozhe Hu

Date

08/08/2011

Note

Ref. R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. Advances in Computational Mathematics, 1 (1993), pp. 127-137.

Definition at line 4771 of file [BlaSpmvBSR.c](#).

9.89.2.10 fasp_blas_dbsr_rap_agg()

```
void fasp_blas_dbsr_rap_agg (
    const dBSRmat * R,
    const dBSRmat * A,
    const dBSRmat * P,
    dBSRmat * B )
```

dBSRmat sparse matrix multiplication $B=R*A*P$, where small block matrices in *P* and *R* are identity matrices!

Parameters

<i>R</i>	Pointer to the dBSRmat matrix
<i>A</i>	Pointer to the dBSRmat matrix
<i>P</i>	Pointer to the dBSRmat matrix
<i>B</i>	Pointer to dBSRmat matrix equal to $R*A*P$ (output)

Author

Xiaozhe Hu

Date

10/24/2012

Definition at line 5227 of file BlasSpmvBSR.c.

9.90 BlasSpmvBSR.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <math.h>
00015
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_functs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00038 void fasp_blas_dbsr_axm (dBSRmat      *A,
00039                        const REAL    alpha)
00040 {
00041     const INT nnz = A->NNZ;
00042     const INT nb  = A->nb;
00043
00044     // A direct calculation can be written as:
00045     fasp_blas_darray_ax(nnz*nb*nb, alpha, A->val);
00046 }
00047
00067 void fasp_blas_dbsr_aAxpby (const REAL    alpha,
00068                            dBSRmat      *A,
00069                            REAL          *x,
00070                            const REAL    beta,
00071                            REAL          *y )
00072 {
00073     /* members of A */
00074     INT ROW = A->ROW;
00075     INT nb  = A->nb;
00076     INT *IA = A->IA;
00077     INT *JA = A->JA;
00078     REAL *val = A->val;
00079
00080     /* local variables */
00081     INT size = ROW*nb;
00082     INT jump = nb*nb;
00083     INT i, j, k, iend;
00084     REAL temp;
00085     REAL *pA = NULL;
00086     REAL *px0 = NULL;
00087     REAL *py0 = NULL;
00088     REAL *py = NULL;
00089
00090     SHORT nthreads = 1, use_omp = FALSE;
00091
00092 #ifdef _OPENMP
00093     if ( ROW > OPENMP_HOLDS ) {
00094         use_omp = TRUE;
00095         nthreads = fasp_get_num_threads();
00096     }
00097 #endif
00098
00099     /*-----*/
00100     /*      Treat (alpha == 0.0) computation      */
00101     /*-----*/
00102
00103     if (alpha == 0.0) {
00104         fasp_blas_darray_ax(size, beta, y);
00105         return;
00106     }
00107
00108     /*-----*/
00109     /*      y = (beta/alpha)*y      */
00110     /*-----*/
00111

```

```

00112     temp = beta / alpha;
00113     if (temp != 1.0) {
00114         if (temp == 0.0) {
00115             memset(y, 0X0, size*sizeof(REAL));
00116         }
00117         else {
00118             //for (i = size; i--; ) y[i] *= temp; // modified by Xiaozhe, 03/11/2011
00119             fasp_blas_darray_ax(size, temp, y);
00120         }
00121     }
00122
00123     //-----
00124     //   y += A*x (Core Computation)
00125     //   each non-zero block elements are stored in row-major order
00126     //-----
00127
00128     switch (nb)
00129     {
00130         case 2:
00131         {
00132             if (use_openmp) {
00133                 INT myid, mybegin, myend;
00134 #ifdef _OPENMP
00135 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
00136 #endif
00137                 for (myid =0; myid < nthreads; myid++) {
00138                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00139                     for (i=mybegin; i < myend; ++i) {
00140                         py0 = &y[i*2];
00141                         iend = IA[i+1];
00142                         for (k = IA[i]; k < iend; ++k) {
00143                             j = JA[k];
00144                             pA = val+k*4; // &val[k*jump];
00145                             px0 = x+j*2; // &x[j*nb];
00146                             py = py0;
00147                             fasp_blas_smat_ypAx_nc2( pA, px0, py );
00148                         }
00149                     }
00150                 }
00151             }
00152             else {
00153                 for (i = 0; i < ROW; ++i) {
00154                     py0 = &y[i*2];
00155                     iend = IA[i+1];
00156                     for (k = IA[i]; k < iend; ++k) {
00157                         j = JA[k];
00158                         pA = val+k*4; // &val[k*jump];
00159                         px0 = x+j*2; // &x[j*nb];
00160                         py = py0;
00161                         fasp_blas_smat_ypAx_nc2( pA, px0, py );
00162                     }
00163                 }
00164             }
00165         }
00166         break;
00167
00168         case 3:
00169         {
00170             if (use_openmp) {
00171                 INT myid, mybegin, myend;
00172 #ifdef _OPENMP
00173 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
00174 #endif
00175                 for (myid =0; myid < nthreads; myid++) {
00176                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00177                     for (i=mybegin; i < myend; ++i) {
00178                         py0 = &y[i*3];
00179                         iend = IA[i+1];
00180                         for (k = IA[i]; k < iend; ++k) {
00181                             j = JA[k];
00182                             pA = val+k*9; // &val[k*jump];
00183                             px0 = x+j*3; // &x[j*nb];
00184                             py = py0;
00185                             fasp_blas_smat_ypAx_nc3( pA, px0, py );
00186                         }
00187                     }
00188                 }
00189             }
00190             else {
00191                 for (i = 0; i < ROW; ++i) {
00192                     py0 = &y[i*3];

```

```

00193         iend = IA[i+1];
00194         for (k = IA[i]; k < iend; ++k) {
00195             j = JA[k];
00196             pA = val+k*9; // &val[k*jump];
00197             px0 = x+j*3; // &x[j*nb];
00198             py = py0;
00199             fasp_blas_smat_ypAx_nc3( pA, px0, py );
00200         }
00201     }
00202 }
00203 }
00204     break;
00205
00206 case 5:
00207 {
00208     if (use_openmp) {
00209         INT myid, mybegin, myend;
00210 #ifdef _OPENMP
00211 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py, iend)
00212 #endif
00213         for (myid = 0; myid < nthreads; myid++) {
00214             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00215             for (i=mybegin; i < myend; ++i) {
00216                 py0 = &y[i*5];
00217                 iend = IA[i+1];
00218                 for (k = IA[i]; k < iend; ++k) {
00219                     j = JA[k];
00220                     pA = val+k*25; // &val[k*jump];
00221                     px0 = x+j*5; // &x[j*nb];
00222                     py = py0;
00223                     fasp_blas_smat_ypAx_nc5( pA, px0, py );
00224                 }
00225             }
00226         }
00227     }
00228     else {
00229         for (i = 0; i < ROW; ++i) {
00230             py0 = &y[i*5];
00231             iend = IA[i+1];
00232             for (k = IA[i]; k < iend; ++k) {
00233                 j = JA[k];
00234                 pA = val+k*25; // &val[k*jump];
00235                 px0 = x+j*5; // &x[j*nb];
00236                 py = py0;
00237                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
00238             }
00239         }
00240     }
00241 }
00242     break;
00243
00244 case 7:
00245 {
00246     if (use_openmp) {
00247         INT myid, mybegin, myend;
00248 #ifdef _OPENMP
00249 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py, iend)
00250 #endif
00251         for (myid = 0; myid < nthreads; myid++) {
00252             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00253             for (i=mybegin; i < myend; ++i) {
00254                 py0 = &y[i*7];
00255                 iend = IA[i+1];
00256                 for (k = IA[i]; k < iend; ++k) {
00257                     j = JA[k];
00258                     pA = val+k*49; // &val[k*jump];
00259                     px0 = x+j*7; // &x[j*nb]❖❖
00260                     py = py0;
00261                     fasp_blas_smat_ypAx_nc7( pA, px0, py );
00262                 }
00263             }
00264         }
00265     }
00266     else {
00267         for (i = 0; i < ROW; ++i) {
00268             py0 = &y[i*7];
00269             iend = IA[i+1];
00270             for (k = IA[i]; k < iend; ++k) {
00271                 j = JA[k];
00272                 pA = val+k*49; // &val[k*jump];
00273                 px0 = x+j*7; // &x[j*nb];

```

```

00274         py = py0;
00275         fasp_blas_smat_ypAx_nc7( pA, px0, py );
00276     }
00277 }
00278 }
00279 }
00280     break;
00281
00282     default:
00283     {
00284         if (use_omp) {
00285             INT myid, mybegin, myend;
00286 #ifdef _OPENMP
00287 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py, iend)
00288 #endif
00289             for (myid = 0; myid < nthreads; myid++) {
00290                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00291                 for (i = mybegin; i < myend; ++i) {
00292                     py0 = &y[i*nb];
00293                     iend = IA[i+1];
00294                     for (k = IA[i]; k < iend; ++k) {
00295                         j = JA[k];
00296                         pA = val+k*jump; // &val[k*jump];
00297                         px0 = x+j*nb; // &x[j*nb];
00298                         py = py0;
00299                         fasp_blas_smat_ypAx( pA, px0, py, nb );
00300                     }
00301                 }
00302             }
00303         }
00304         else {
00305             for (i = 0; i < ROW; ++i) {
00306                 py0 = &y[i*nb];
00307                 iend = IA[i+1];
00308                 for (k = IA[i]; k < iend; ++k) {
00309                     j = JA[k];
00310                     pA = val+k*jump; // &val[k*jump];
00311                     px0 = x+j*nb; // &x[j*nb];
00312                     py = py0;
00313                     fasp_blas_smat_ypAx( pA, px0, py, nb );
00314                 }
00315             }
00316         }
00317     }
00318     break;
00319 }
00320
00321 //-----
00322 //   y = alpha*y
00323 //-----
00324
00325 if (alpha != 1.0) {
00326     fasp_blas_darray_ax(size, alpha, y);
00327 }
00328 }
00329
00348 void fasp_blas_dbsr_aAxy (const REAL    alpha,
00349                          const dBSRmat *A,
00350                          const REAL    *x,
00351                          REAL          *y)
00352 {
00353     /* members of A */
00354     const INT    ROW = A->ROW;
00355     const INT    nb  = A->nb;
00356     const INT    *IA  = A->IA;
00357     const INT    *JA  = A->JA;
00358     const REAL    *val = A->val;
00359
00360     /* local variables */
00361     const REAL *pA    = NULL;
00362     const REAL *px0   = NULL;
00363     REAL *py0        = NULL;
00364     REAL *py         = NULL;
00365
00366     REAL temp = 0.0;
00367     INT size = ROW*nb;
00368     INT jump = nb*nb;
00369     INT i, j, k, iend;
00370
00371     SHORT nthreads = 1, use_omp = FALSE;
00372

```



```

00373 #ifdef _OPENMP
00374     if ( ROW > OPENMP_HOLDS ) {
00375         use_omp = TRUE;
00376         nthreads = fasp_get_num_threads();
00377     }
00378 #endif
00379
00380     //-----
00381     //   Treat (alpha == 0.0) computation
00382     //-----
00383
00384     if (alpha == 0.0){
00385         return; // Nothing to compute
00386     }
00387
00388     //-----
00389     //   y = (1.0/alpha)*y
00390     //-----
00391
00392     if (alpha != 1.0){
00393         temp = 1.0 / alpha;
00394         fasp_blas_darray_ax(size, temp, y);
00395     }
00396
00397     //-----
00398     //   y += A*x (Core Computation)
00399     //   each non-zero block elements are stored in row-major order
00400     //-----
00401
00402     switch (nb)
00403     {
00404         case 2:
00405         {
00406             if (use_omp) {
00407                 INT myid, mybegin, myend;
00408 #ifdef _OPENMP
00409 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
00410 #endif
00411                 for (myid = 0; myid < nthreads; myid++) {
00412                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00413                     for (i=mybegin; i < myend; ++i) {
00414                         py0 = &y[i*2];
00415                         iend = IA[i+1];
00416                         for (k = IA[i]; k < iend; ++k) {
00417                             j = JA[k];
00418                             pA = val+k*4; // &val[k*jump];
00419                             px0 = x+j*2; // &x[j*nb];
00420                             py = py0;
00421                             fasp_blas_smat_ypAx_nc2( pA, px0, py );
00422                         }
00423                     }
00424                 }
00425             }
00426             else {
00427                 for (i = 0; i < ROW; ++i) {
00428                     py0 = &y[i*2];
00429                     iend = IA[i+1];
00430                     for (k = IA[i]; k < iend; ++k) {
00431                         j = JA[k];
00432                         pA = val+k*4; // &val[k*jump];
00433                         px0 = x+j*2; // &x[j*nb];
00434                         py = py0;
00435                         fasp_blas_smat_ypAx_nc2( pA, px0, py );
00436                     }
00437                 }
00438             }
00439         }
00440         break;
00441
00442         case 3:
00443         {
00444             if (use_omp) {
00445                 INT myid, mybegin, myend;
00446 #ifdef _OPENMP
00447 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
00448 #endif
00449                 for (myid = 0; myid < nthreads; myid++) {
00450                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00451                     for (i=mybegin; i < myend; ++i) {
00452                         py0 = &y[i*3];
00453                         iend = IA[i+1];

```

```

00454         for (k = IA[i]; k < iend; ++k) {
00455             j = JA[k];
00456             pA = val+k*9; // &val[k*jump];
00457             px0 = x+j*3; // &x[j*nb];
00458             py = py0;
00459             fasp_blas_smat_ypAx_nc3( pA, px0, py );
00460         }
00461     }
00462 }
00463 }
00464 else {
00465     for (i = 0; i < ROW; ++i){
00466         py0 = &y[i*3];
00467         iend = IA[i+1];
00468         for (k = IA[i]; k < iend; ++k) {
00469             j = JA[k];
00470             pA = val+k*9; // &val[k*jump];
00471             px0 = x+j*3; // &x[j*nb];
00472             py = py0;
00473             fasp_blas_smat_ypAx_nc3( pA, px0, py );
00474         }
00475     }
00476 }
00477 }
00478 break;
00479
00480 case 5:
00481 {
00482     if (use_omp) {
00483         INT myid, mybegin, myend;
00484 #ifdef _OPENMP
00485 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
00486 #endif
00487         for (myid =0; myid < nthreads; myid++) {
00488             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00489             for (i=mybegin; i < myend; ++i) {
00490                 py0 = &y[i*5];
00491                 iend = IA[i+1];
00492                 for (k = IA[i]; k < iend; ++k) {
00493                     j = JA[k];
00494                     pA = val+k*25; // &val[k*jump];
00495                     px0 = x+j*5; // &x[j*nb];
00496                     py = py0;
00497                     fasp_blas_smat_ypAx_nc5( pA, px0, py );
00498                 }
00499             }
00500         }
00501     }
00502     else {
00503         for (i = 0; i < ROW; ++i){
00504             py0 = &y[i*5];
00505             iend = IA[i+1];
00506             for (k = IA[i]; k < iend; ++k) {
00507                 j = JA[k];
00508                 pA = val+k*25; // &val[k*jump];
00509                 px0 = x+j*5; // &x[j*nb];
00510                 py = py0;
00511                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
00512             }
00513         }
00514     }
00515 }
00516 break;
00517
00518 case 7:
00519 {
00520     if (use_omp) {
00521         INT myid, mybegin, myend;
00522 #ifdef _OPENMP
00523 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
00524 #endif
00525         for (myid =0; myid < nthreads; myid++) {
00526             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00527             for (i=mybegin; i < myend; ++i) {
00528                 py0 = &y[i*7];
00529                 iend = IA[i+1];
00530                 for (k = IA[i]; k < iend; ++k) {
00531                     j = JA[k];
00532                     pA = val+k*49; // &val[k*jump];
00533                     px0 = x+j*7; // &x[j*nb];
00534                     py = py0;

```

```

00535             fasp_blas_smat_ypAx_nc7( pA, px0, py );
00536         }
00537     }
00538 }
00539 }
00540 else {
00541     for (i = 0; i < ROW; ++i) {
00542         py0 = &y[i*7];
00543         iend = IA[i+1];
00544         for (k = IA[i]; k < iend; ++k) {
00545             j = JA[k];
00546             pA = val+k*49; // &val[k*jump];
00547             px0 = x+j*7; // &x[j*nb];
00548             py = py0;
00549             fasp_blas_smat_ypAx_nc7( pA, px0, py );
00550         }
00551     }
00552 }
00553 }
00554 }
00555 break;
00556
00557 default:
00558 {
00559     if (use_openmp) {
00560         INT myid, mybegin, myend;
00561 #ifdef _OPENMP
00562 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
00563 #endif
00564         for (myid =0; myid < nthreads; myid++) {
00565             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00566             for (i=mybegin; i < myend; ++i) {
00567                 py0 = &y[i*nb];
00568                 iend = IA[i+1];
00569                 for (k = IA[i]; k < iend; ++k) {
00570                     j = JA[k];
00571                     pA = val+k*jump; // &val[k*jump];
00572                     px0 = x+j*nb; // &x[j*nb];
00573                     py = py0;
00574                     fasp_blas_smat_ypAx( pA, px0, py, nb );
00575                 }
00576             }
00577         }
00578     }
00579 }
00580 else {
00581     for (i = 0; i < ROW; ++i) {
00582         py0 = &y[i*nb];
00583         iend = IA[i+1];
00584         for (k = IA[i]; k < iend; ++k) {
00585             j = JA[k];
00586             pA = val+k*jump; // &val[k*jump];
00587             px0 = x+j*nb; // &x[j*nb];
00588             py = py0;
00589             fasp_blas_smat_ypAx( pA, px0, py, nb );
00590         }
00591     }
00592 }
00593 }
00594 }
00595 break;
00596 }
00597
00598 //-----
00599 // y = alpha*y
00600 //-----
00601
00602 if (alpha != 1.0){
00603     fasp_blas_darray_ax(size, alpha, y);
00604 }
00605 return;
00606 }
00607
00624 void fasp_blas_dbsr_aApy_agg (const REAL    alpha,
00625                               const dBSRmat *A,
00626                               const REAL    *x,
00627                               REAL          *y)
00628 {
00629     /* members of A */
00630     const INT ROW = A->ROW;
00631     const INT nb  = A->nb;

```

```

00632     const INT  *IA  = A->IA;
00633     const INT  *JA  = A->JA;
00634
00635     /* local variables */
00636     const REAL *px0 = NULL;
00637     REAL      *py0 = NULL, *py = NULL;
00638     SHORT      nthreads = 1, use_omp = FALSE;
00639
00640     INT         size = ROW*nb;
00641     INT         i, j, k, iend;
00642     REAL        temp = 0.0;
00643
00644 #ifdef _OPENMP
00645     if ( ROW > OPENMP_HOLDS ) {
00646         use_omp = TRUE;
00647         nthreads = fasp_get_num_threads();
00648     }
00649 #endif
00650
00651     //-----
00652     //   Treat (alpha == 0.0) computation
00653     //-----
00654
00655     if (alpha == 0.0){
00656         return; // Nothing to compute
00657     }
00658
00659     //-----
00660     //   y = (1.0/alpha)*y
00661     //-----
00662
00663     if (alpha != 1.0){
00664         temp = 1.0 / alpha;
00665         fasp_blas_darray_ax(size, temp, y);
00666     }
00667
00668     //-----
00669     //   y += A*x (Core Computation)
00670     //   each non-zero block elements are stored in row-major order
00671     //-----
00672
00673     switch (nb)
00674     {
00675         case 2:
00676         {
00677             if (use_omp) {
00678                 INT myid, mybegin, myend;
00679 #ifdef _OPENMP
00680 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py,iend)
00681 #endif
00682                 for (myid = 0; myid < nthreads; myid++) {
00683                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00684                     for (i=mybegin; i < myend; ++i) {
00685                         py0 = &y[i*2];
00686                         iend = IA[i+1];
00687                         for (k = IA[i]; k < iend; ++k) {
00688                             j = JA[k];
00689                             px0 = x+j*2; // &x[j*nb];
00690                             py = py0;
00691                             py[0] += px0[0];
00692                             py[1] += px0[1];
00693                         }
00694                     }
00695                 }
00696             }
00697             else {
00698                 for (i = 0; i < ROW; ++i) {
00699                     py0 = &y[i*2];
00700                     iend = IA[i+1];
00701                     for (k = IA[i]; k < iend; ++k) {
00702                         j = JA[k];
00703                         px0 = x+j*2; // &x[j*nb];
00704                         py = py0;
00705                         py[0] += px0[0];
00706                         py[1] += px0[1];
00707                     }
00708                 }
00709             }
00710         }
00711         break;
00712

```

```

00713         case 3:
00714         {
00715             if (use_omp) {
00716                 INT myid, mybegin, myend;
00717 #ifdef _OPENMP
00718 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py,iend )
00719 #endif
00720                 for (myid =0; myid < nthreads; myid++) {
00721                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00722                     for (i=mybegin; i < myend; ++i) {
00723                         py0 = &y[i*3];
00724                         iend = IA[i+1];
00725                         for (k = IA[i]; k < iend; ++k) {
00726                             j = JA[k];
00727                             px0 = x+j*3; // &x[j*nb];
00728                             py = py0;
00729                             py[0] += px0[0];
00730                             py[1] += px0[1];
00731                             py[2] += px0[2];
00732                         }
00733                     }
00734                 }
00735             }
00736             else {
00737                 for (i = 0; i < ROW; ++i){
00738                     py0 = &y[i*3];
00739                     iend = IA[i+1];
00740                     for (k = IA[i]; k < iend; ++k) {
00741                         j = JA[k];
00742                         px0 = x+j*3; // &x[j*nb];
00743                         py = py0;
00744                         py[0] += px0[0];
00745                         py[1] += px0[1];
00746                         py[2] += px0[2];
00747                     }
00748                 }
00749             }
00750         }
00751         break;
00752
00753         case 5:
00754         {
00755             if (use_omp) {
00756                 INT myid, mybegin, myend;
00757 #ifdef _OPENMP
00758 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py,iend )
00759 #endif
00760                 for (myid =0; myid < nthreads; myid++) {
00761                     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00762                     for (i=mybegin; i < myend; ++i) {
00763                         py0 = &y[i*5];
00764                         iend = IA[i+1];
00765                         for (k = IA[i]; k < iend; ++k) {
00766                             j = JA[k];
00767                             px0 = x+j*5; // &x[j*nb];
00768                             py = py0;
00769                             py[0] += px0[0];
00770                             py[1] += px0[1];
00771                             py[2] += px0[2];
00772                             py[3] += px0[3];
00773                             py[4] += px0[4];
00774                         }
00775                     }
00776                 }
00777             }
00778             else {
00779                 for (i = 0; i < ROW; ++i){
00780                     py0 = &y[i*5];
00781                     iend = IA[i+1];
00782                     for (k = IA[i]; k < iend; ++k) {
00783                         j = JA[k];
00784                         px0 = x+j*5; // &x[j*nb];
00785                         py = py0;
00786                         py[0] += px0[0];
00787                         py[1] += px0[1];
00788                         py[2] += px0[2];
00789                         py[3] += px0[3];
00790                         py[4] += px0[4];
00791                     }
00792                 }
00793             }

```

```

00794     }
00795         break;
00796
00797     case 7:
00798     {
00799         if (use_omp) {
00800             INT myid, mybegin, myend;
00801 #ifdef _OPENMP
00802 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py, iend)
00803 #endif
00804             for (myid = 0; myid < nthreads; myid++) {
00805                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00806                 for (i = mybegin; i < myend; ++i) {
00807                     py0 = &y[i*7];
00808                     iend = IA[i+1];
00809                     for (k = IA[i]; k < iend; ++k) {
00810                         j = JA[k];
00811                         px0 = x+j*7; // &x[j*nb];
00812                         py = py0;
00813                         py[0] += px0[0];
00814                         py[1] += px0[1];
00815                         py[2] += px0[2];
00816                         py[3] += px0[3];
00817                         py[4] += px0[4];
00818                         py[5] += px0[5];
00819                         py[6] += px0[6];
00820                     }
00821                 }
00822             }
00823         }
00824         else {
00825             for (i = 0; i < ROW; ++i) {
00826                 py0 = &y[i*7];
00827                 iend = IA[i+1];
00828                 for (k = IA[i]; k < iend; ++k) {
00829                     j = JA[k];
00830                     px0 = x+j*7; // &x[j*nb];
00831                     py = py0;
00832                     py[0] += px0[0];
00833                     py[1] += px0[1];
00834                     py[2] += px0[2];
00835                     py[3] += px0[3];
00836                     py[4] += px0[4];
00837                     py[5] += px0[5];
00838                     py[6] += px0[6];
00839                 }
00840             }
00841         }
00842     }
00843 }
00844 break;
00845
00846 default:
00847 {
00848     if (use_omp) {
00849         INT myid, mybegin, myend;
00850 #ifdef _OPENMP
00851 #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py, iend)
00852 #endif
00853         for (myid = 0; myid < nthreads; myid++) {
00854             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00855             for (i = mybegin; i < myend; ++i) {
00856                 py0 = &y[i*nb];
00857                 iend = IA[i+1];
00858                 for (k = IA[i]; k < iend; ++k) {
00859                     j = JA[k];
00860                     px0 = x+j*nb; // &x[j*nb];
00861                     py = py0;
00862                     fasp_blas_darray_axpy(nb, 1.0, px0, py);
00863                 }
00864             }
00865         }
00866     }
00867     else {
00868         for (i = 0; i < ROW; ++i) {
00869             py0 = &y[i*nb];
00870             iend = IA[i+1];
00871             for (k = IA[i]; k < iend; ++k) {
00872                 j = JA[k];
00873                 px0 = x+j*nb; // &x[j*nb];
00874             }

```

```

00875         py = py0;
00876         fasp_blas_darray_axpy(nb, 1.0, px0, py);
00877     }
00878 }
00879 }
00880 }
00881 }
00882     break;
00883 }
00884
00885 //-----
00886 //   y = alpha*y
00887 //-----
00888
00889 if ( alpha != 1.0 ) fasp_blas_darray_ax(size, alpha, y);
00890
00891 return;
00892 }
00893
00910 void fasp_blas_dbsr_mxv (const dBSRmat *A,
00911                        const REAL *x,
00912                        REAL *y)
00913 {
00914     /* members of A */
00915     const INT ROW = A->ROW;
00916     const INT nb = A->nb;
00917     const INT *IA = A->IA;
00918     const INT *JA = A->JA;
00919     const REAL *val = A->val;
00920
00921     /* local variables */
00922     INT size = ROW*nb;
00923     INT jump = nb*nb;
00924     INT i, j, k, num_nnz_row;
00925
00926     const REAL *pA = NULL;
00927     const REAL *px0 = NULL;
00928     REAL *py0 = NULL;
00929     REAL *py = NULL;
00930
00931     SHORT use_omp = FALSE;
00932
00933 #ifdef _OPENMP
00934     INT myid, mybegin, myend, nthreads;
00935     if ( ROW > OPENMP_HOLDS ) {
00936         use_omp = TRUE;
00937         nthreads = fasp_get_num_threads();
00938     }
00939 #endif
00940
00941 //-----
00942 //   zero out 'y'
00943 //-----
00944 fasp_darray_set(size, y, 0.0);
00945
00946 //-----
00947 //   y = A*x (Core Computation)
00948 //   each non-zero block elements are stored in row-major order
00949 //-----
00950
00951 switch (nb)
00952 {
00953     case 3:
00954     {
00955         if (use_omp) {
00956 #ifdef _OPENMP
00957 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
00958 {
00959             myid = omp_get_thread_num();
00960             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00961             for (i=mybegin; i < myend; ++i)
00962             {
00963                 py0 = &y[i*3];
00964                 num_nnz_row = IA[i+1] - IA[i];
00965                 switch(num_nnz_row)
00966                 {
00967                     case 3:
00968                         k = IA[i];
00969                         j = JA[k];
00970                         pA = val+k*9;
00971                         px0 = x+j*3;

```

```

00972         py = py0;
00973         fasp_blas_smat_ypAx_nc3( pA, px0, py );
00974
00975         k ++;
00976         j = JA[k];
00977         pA = val+k*9;
00978         px0 = x+j*3;
00979         py = py0;
00980         fasp_blas_smat_ypAx_nc3( pA, px0, py );
00981
00982         k ++;
00983         j = JA[k];
00984         pA = val+k*9;
00985         px0 = x+j*3;
00986         py = py0;
00987         fasp_blas_smat_ypAx_nc3( pA, px0, py );
00988
00989         break;
00990
00991     case 4:
00992         k = IA[i];
00993         j = JA[k];
00994         pA = val+k*9;
00995         px0 = x+j*3;
00996         py = py0;
00997         fasp_blas_smat_ypAx_nc3( pA, px0, py );
00998
00999         k ++;
01000         j = JA[k];
01001         pA = val+k*9;
01002         px0 = x+j*3;
01003         py = py0;
01004         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01005
01006         k ++;
01007         j = JA[k];
01008         pA = val+k*9;
01009         px0 = x+j*3;
01010         py = py0;
01011         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01012
01013         k ++;
01014         j = JA[k];
01015         pA = val+k*9;
01016         px0 = x+j*3;
01017         py = py0;
01018         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01019
01020         break;
01021
01022     case 5:
01023         k = IA[i];
01024         j = JA[k];
01025         pA = val+k*9;
01026         px0 = x+j*3;
01027         py = py0;
01028         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01029
01030         k ++;
01031         j = JA[k];
01032         pA = val+k*9;
01033         px0 = x+j*3;
01034         py = py0;
01035         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01036
01037         k ++;
01038         j = JA[k];
01039         pA = val+k*9;
01040         px0 = x+j*3;
01041         py = py0;
01042         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01043
01044         k ++;
01045         j = JA[k];
01046         pA = val+k*9;
01047         px0 = x+j*3;
01048         py = py0;
01049         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01050
01051         k ++;
01052         j = JA[k];

```



```

01053         pA = val+k*9;
01054         px0 = x+j*3;
01055         py = py0;
01056         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01057
01058         break;
01059
01060     case 6:
01061         k = IA[i];
01062         j = JA[k];
01063         pA = val+k*9;
01064         px0 = x+j*3;
01065         py = py0;
01066         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01067
01068         k ++;
01069         j = JA[k];
01070         pA = val+k*9;
01071         px0 = x+j*3;
01072         py = py0;
01073         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01074
01075         k ++;
01076         j = JA[k];
01077         pA = val+k*9;
01078         px0 = x+j*3;
01079         py = py0;
01080         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01081
01082         k ++;
01083         j = JA[k];
01084         pA = val+k*9;
01085         px0 = x+j*3;
01086         py = py0;
01087         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01088
01089         k ++;
01090         j = JA[k];
01091         pA = val+k*9;
01092         px0 = x+j*3;
01093         py = py0;
01094         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01095
01096         k ++;
01097         j = JA[k];
01098         pA = val+k*9;
01099         px0 = x+j*3;
01100         py = py0;
01101         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01102
01103         break;
01104
01105     case 7:
01106         k = IA[i];
01107         j = JA[k];
01108         pA = val+k*9;
01109         px0 = x+j*3;
01110         py = py0;
01111         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01112
01113         k ++;
01114         j = JA[k];
01115         pA = val+k*9;
01116         px0 = x+j*3;
01117         py = py0;
01118         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01119
01120         k ++;
01121         j = JA[k];
01122         pA = val+k*9;
01123         px0 = x+j*3;
01124         py = py0;
01125         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01126
01127         k ++;
01128         j = JA[k];
01129         pA = val+k*9;
01130         px0 = x+j*3;
01131         py = py0;
01132         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01133

```

```

01134             k ++;
01135             j = JA[k];
01136             pA = val+k*9;
01137             px0 = x+j*3;
01138             py = py0;
01139             fasp_blas_smat_ypAx_nc3( pA, px0, py );
01140
01141             k ++;
01142             j = JA[k];
01143             pA = val+k*9;
01144             px0 = x+j*3;
01145             py = py0;
01146             fasp_blas_smat_ypAx_nc3( pA, px0, py );
01147
01148             k ++;
01149             j = JA[k];
01150             pA = val+k*9;
01151             px0 = x+j*3;
01152             py = py0;
01153             fasp_blas_smat_ypAx_nc3( pA, px0, py );
01154
01155             break;
01156
01157         default:
01158             for (k = IA[i]; k < IA[i+1]; ++k)
01159             {
01160                 j = JA[k];
01161                 pA = val+k*9;
01162                 px0 = x+j*3;
01163                 py = py0;
01164                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
01165             }
01166             break;
01167         }
01168     }
01169 }
01170 #endif
01171 }
01172 else {
01173     for (i = 0; i < ROW; ++i)
01174     {
01175         py0 = &y[i*3];
01176         num_nnz_row = IA[i+1] - IA[i];
01177         switch(num_nnz_row)
01178         {
01179             case 3:
01180                 k = IA[i];
01181                 j = JA[k];
01182                 pA = val+k*9; // &val[k*jump];
01183                 px0 = x+j*3; // &x[j*nb];
01184                 py = py0;
01185                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
01186
01187                 k ++;
01188                 j = JA[k];
01189                 pA = val+k*9; // &val[k*jump];
01190                 px0 = x+j*3; // &x[j*nb];
01191                 py = py0;
01192                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
01193
01194                 k ++;
01195                 j = JA[k];
01196                 pA = val+k*9; // &val[k*jump];
01197                 px0 = x+j*3; // &x[j*nb];
01198                 py = py0;
01199                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
01200
01201                 break;
01202
01203             case 4:
01204                 k = IA[i];
01205                 j = JA[k];
01206                 pA = val+k*9; // &val[k*jump];
01207                 px0 = x+j*3; // &x[j*nb];
01208                 py = py0;
01209                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
01210
01211                 k ++;
01212                 j = JA[k];
01213                 pA = val+k*9; // &val[k*jump];
01214                 px0 = x+j*3; // &x[j*nb];

```

```

01215         py = py0;
01216         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01217
01218         k ++;
01219         j = JA[k];
01220         pA = val+k*9; // &val[k*jump];
01221         px0 = x+j*3; // &x[j*nb];
01222         py = py0;
01223         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01224
01225         k ++;
01226         j = JA[k];
01227         pA = val+k*9; // &val[k*jump];
01228         px0 = x+j*3; // &x[j*nb];
01229         py = py0;
01230         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01231
01232         break;
01233
01234     case 5:
01235         k = IA[i];
01236         j = JA[k];
01237         pA = val+k*9; // &val[k*jump];
01238         px0 = x+j*3; // &x[j*nb];
01239         py = py0;
01240         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01241
01242         k ++;
01243         j = JA[k];
01244         pA = val+k*9; // &val[k*jump];
01245         px0 = x+j*3; // &x[j*nb];
01246         py = py0;
01247         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01248
01249         k ++;
01250         j = JA[k];
01251         pA = val+k*9; // &val[k*jump];
01252         px0 = x+j*3; // &x[j*nb];
01253         py = py0;
01254         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01255
01256         k ++;
01257         j = JA[k];
01258         pA = val+k*9; // &val[k*jump];
01259         px0 = x+j*3; // &x[j*nb];
01260         py = py0;
01261         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01262
01263         k ++;
01264         j = JA[k];
01265         pA = val+k*9; // &val[k*jump];
01266         px0 = x+j*3; // &x[j*nb];
01267         py = py0;
01268         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01269
01270         break;
01271
01272     case 6:
01273         k = IA[i];
01274         j = JA[k];
01275         pA = val+k*9; // &val[k*jump];
01276         px0 = x+j*3; // &x[j*nb];
01277         py = py0;
01278         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01279
01280         k ++;
01281         j = JA[k];
01282         pA = val+k*9; // &val[k*jump];
01283         px0 = x+j*3; // &x[j*nb];
01284         py = py0;
01285         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01286
01287         k ++;
01288         j = JA[k];
01289         pA = val+k*9; // &val[k*jump];
01290         px0 = x+j*3; // &x[j*nb];
01291         py = py0;
01292         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01293
01294         k ++;
01295         j = JA[k];

```

```

01296         pA = val+k*9; // &val[k*jump];
01297         px0 = x+j*3; // &x[j*nb];
01298         py = py0;
01299         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01300
01301         k ++;
01302         j = JA[k];
01303         pA = val+k*9; // &val[k*jump];
01304         px0 = x+j*3; // &x[j*nb];
01305         py = py0;
01306         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01307
01308         k ++;
01309         j = JA[k];
01310         pA = val+k*9; // &val[k*jump];
01311         px0 = x+j*3; // &x[j*nb];
01312         py = py0;
01313         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01314
01315         break;
01316
01317     case 7:
01318         k = IA[i];
01319         j = JA[k];
01320         pA = val+k*9; // &val[k*jump];
01321         px0 = x+j*3; // &x[j*nb];
01322         py = py0;
01323         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01324
01325         k ++;
01326         j = JA[k];
01327         pA = val+k*9; // &val[k*jump];
01328         px0 = x+j*3; // &x[j*nb];
01329         py = py0;
01330         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01331
01332         k ++;
01333         j = JA[k];
01334         pA = val+k*9; // &val[k*jump];
01335         px0 = x+j*3; // &x[j*nb];
01336         py = py0;
01337         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01338
01339         k ++;
01340         j = JA[k];
01341         pA = val+k*9; // &val[k*jump];
01342         px0 = x+j*3; // &x[j*nb];
01343         py = py0;
01344         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01345
01346         k ++;
01347         j = JA[k];
01348         pA = val+k*9; // &val[k*jump];
01349         px0 = x+j*3; // &x[j*nb];
01350         py = py0;
01351         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01352
01353         k ++;
01354         j = JA[k];
01355         pA = val+k*9; // &val[k*jump];
01356         px0 = x+j*3; // &x[j*nb];
01357         py = py0;
01358         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01359
01360         k ++;
01361         j = JA[k];
01362         pA = val+k*9; // &val[k*jump];
01363         px0 = x+j*3; // &x[j*nb];
01364         py = py0;
01365         fasp_blas_smat_ypAx_nc3( pA, px0, py );
01366
01367         break;
01368
01369     default:
01370         for (k = IA[i]; k < IA[i+1]; ++k)
01371         {
01372             j = JA[k];
01373             pA = val+k*9; // &val[k*jump];
01374             px0 = x+j*3; // &x[j*nb];
01375             py = py0;
01376             fasp_blas_smat_ypAx_nc3( pA, px0, py );

```

```

01377             }
01378             break;
01379         }
01380     }
01381 }
01382 }
01383 break;
01384
01385 case 5:
01386 {
01387     if (use_omp) {
01388 #ifdef _OPENMP
01389 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
01390 {
01391     myid = omp_get_thread_num();
01392     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01393     for (i=mybegin; i < myend; ++i)
01394     {
01395         py0 = &y[i*5];
01396         num_nnz_row = IA[i+1] - IA[i];
01397         switch(num_nnz_row)
01398         {
01399             case 3:
01400                 k = IA[i];
01401                 j = JA[k];
01402                 pA = val+k*25; // &val[k*jump];
01403                 px0 = x+j*5; // &x[j*nb];
01404                 py = py0;
01405                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01406
01407                 k ++;
01408                 j = JA[k];
01409                 pA = val+k*25; // &val[k*jump];
01410                 px0 = x+j*5; // &x[j*nb];
01411                 py = py0;
01412                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01413
01414                 k ++;
01415                 j = JA[k];
01416                 pA = val+k*25; // &val[k*jump];
01417                 px0 = x+j*5; // &x[j*nb];
01418                 py = py0;
01419                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01420
01421                 break;
01422
01423             case 4:
01424                 k = IA[i];
01425                 j = JA[k];
01426                 pA = val+k*25; // &val[k*jump];
01427                 px0 = x+j*5; // &x[j*nb];
01428                 py = py0;
01429                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01430
01431                 k ++;
01432                 j = JA[k];
01433                 pA = val+k*25; // &val[k*jump];
01434                 px0 = x+j*5; // &x[j*nb];
01435                 py = py0;
01436                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01437
01438                 k ++;
01439                 j = JA[k];
01440                 pA = val+k*25; // &val[k*jump];
01441                 px0 = x+j*5; // &x[j*nb];
01442                 py = py0;
01443                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01444
01445                 k ++;
01446                 j = JA[k];
01447                 pA = val+k*25; // &val[k*jump];
01448                 px0 = x+j*5; // &x[j*nb];
01449                 py = py0;
01450                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01451
01452                 break;
01453
01454             case 5:
01455                 k = IA[i];
01456                 j = JA[k];
01457                 pA = val+k*25; // &val[k*jump];

```

```

01458         px0 = x+j*5; // &x[j*nb];
01459         py = py0;
01460         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01461
01462         k ++;
01463         j = JA[k];
01464         pA = val+k*25; // &val[k*jump];
01465         px0 = x+j*5; // &x[j*nb];
01466         py = py0;
01467         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01468
01469         k ++;
01470         j = JA[k];
01471         pA = val+k*25; // &val[k*jump];
01472         px0 = x+j*5; // &x[j*nb];
01473         py = py0;
01474         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01475
01476         k ++;
01477         j = JA[k];
01478         pA = val+k*25; // &val[k*jump];
01479         px0 = x+j*5; // &x[j*nb];
01480         py = py0;
01481         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01482
01483         k ++;
01484         j = JA[k];
01485         pA = val+k*25; // &val[k*jump];
01486         px0 = x+j*5; // &x[j*nb];
01487         py = py0;
01488         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01489
01490         break;
01491
01492     case 6:
01493         k = IA[i];
01494         j = JA[k];
01495         pA = val+k*25; // &val[k*jump];
01496         px0 = x+j*5; // &x[j*nb];
01497         py = py0;
01498         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01499
01500         k ++;
01501         j = JA[k];
01502         pA = val+k*25; // &val[k*jump];
01503         px0 = x+j*5; // &x[j*nb];
01504         py = py0;
01505         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01506
01507         k ++;
01508         j = JA[k];
01509         pA = val+k*25; // &val[k*jump];
01510         px0 = x+j*5; // &x[j*nb];
01511         py = py0;
01512         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01513
01514         k ++;
01515         j = JA[k];
01516         pA = val+k*25; // &val[k*jump];
01517         px0 = x+j*5; // &x[j*nb];
01518         py = py0;
01519         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01520
01521         k ++;
01522         j = JA[k];
01523         pA = val+k*25; // &val[k*jump];
01524         px0 = x+j*5; // &x[j*nb];
01525         py = py0;
01526         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01527
01528         k ++;
01529         j = JA[k];
01530         pA = val+k*25; // &val[k*jump];
01531         px0 = x+j*5; // &x[j*nb];
01532         py = py0;
01533         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01534
01535         break;
01536
01537     case 7:
01538         k = IA[i];

```

```

01539         j = JA[k];
01540         pA = val+k*25; // &val[k*jump];
01541         px0 = x+j*5; // &x[j*nb];
01542         py = py0;
01543         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01544
01545         k ++;
01546         j = JA[k];
01547         pA = val+k*25; // &val[k*jump];
01548         px0 = x+j*5; // &x[j*nb];
01549         py = py0;
01550         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01551
01552         k ++;
01553         j = JA[k];
01554         pA = val+k*25; // &val[k*jump];
01555         px0 = x+j*5; // &x[j*nb];
01556         py = py0;
01557         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01558
01559         k ++;
01560         j = JA[k];
01561         pA = val+k*25; // &val[k*jump];
01562         px0 = x+j*5; // &x[j*nb];
01563         py = py0;
01564         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01565
01566         k ++;
01567         j = JA[k];
01568         pA = val+k*25; // &val[k*jump];
01569         px0 = x+j*5; // &x[j*nb];
01570         py = py0;
01571         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01572
01573         k ++;
01574         j = JA[k];
01575         pA = val+k*25; // &val[k*jump];
01576         px0 = x+j*5; // &x[j*nb];
01577         py = py0;
01578         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01579
01580         k ++;
01581         j = JA[k];
01582         pA = val+k*25; // &val[k*jump];
01583         px0 = x+j*5; // &x[j*nb];
01584         py = py0;
01585         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01586
01587         break;
01588
01589     default:
01590         for (k = IA[i]; k < IA[i+1]; ++k)
01591         {
01592             j = JA[k];
01593             pA = val+k*25; // &val[k*jump];
01594             px0 = x+j*5; // &x[j*nb];
01595             py = py0;
01596             fasp_blas_smat_ypAx_nc5( pA, px0, py );
01597         }
01598         break;
01599     }
01600 }
01601 }
01602 #endif
01603 }
01604 else {
01605     for (i = 0; i < ROW; ++i)
01606     {
01607         py0 = &y[i*5];
01608         num_nnz_row = IA[i+1] - IA[i];
01609         switch(num_nnz_row)
01610         {
01611             case 3:
01612                 k = IA[i];
01613                 j = JA[k];
01614                 pA = val+k*25; // &val[k*jump];
01615                 px0 = x+j*5; // &x[j*nb];
01616                 py = py0;
01617                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
01618
01619                 k ++;

```

```

01620         j = JA[k];
01621         pA = val+k*25; // &val[k*jump];
01622         px0 = x+j*5; // &x[j*nb];
01623         py = py0;
01624         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01625
01626         k ++;
01627         j = JA[k];
01628         pA = val+k*25; // &val[k*jump];
01629         px0 = x+j*5; // &x[j*nb];
01630         py = py0;
01631         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01632
01633         break;
01634
01635     case 4:
01636         k = IA[i];
01637         j = JA[k];
01638         pA = val+k*25; // &val[k*jump];
01639         px0 = x+j*5; // &x[j*nb];
01640         py = py0;
01641         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01642
01643         k ++;
01644         j = JA[k];
01645         pA = val+k*25; // &val[k*jump];
01646         px0 = x+j*5; // &x[j*nb];
01647         py = py0;
01648         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01649
01650         k ++;
01651         j = JA[k];
01652         pA = val+k*25; // &val[k*jump];
01653         px0 = x+j*5; // &x[j*nb];
01654         py = py0;
01655         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01656
01657         k ++;
01658         j = JA[k];
01659         pA = val+k*25; // &val[k*jump];
01660         px0 = x+j*5; // &x[j*nb];
01661         py = py0;
01662         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01663
01664         break;
01665
01666     case 5:
01667         k = IA[i];
01668         j = JA[k];
01669         pA = val+k*25; // &val[k*jump];
01670         px0 = x+j*5; // &x[j*nb];
01671         py = py0;
01672         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01673
01674         k ++;
01675         j = JA[k];
01676         pA = val+k*25; // &val[k*jump];
01677         px0 = x+j*5; // &x[j*nb];
01678         py = py0;
01679         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01680
01681         k ++;
01682         j = JA[k];
01683         pA = val+k*25; // &val[k*jump];
01684         px0 = x+j*5; // &x[j*nb];
01685         py = py0;
01686         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01687
01688         k ++;
01689         j = JA[k];
01690         pA = val+k*25; // &val[k*jump];
01691         px0 = x+j*5; // &x[j*nb];
01692         py = py0;
01693         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01694
01695         k ++;
01696         j = JA[k];
01697         pA = val+k*25; // &val[k*jump];
01698         px0 = x+j*5; // &x[j*nb];
01699         py = py0;
01700         fasp_blas_smat_ypAx_nc5( pA, px0, py );

```



```

01701
01702         break;
01703
01704     case 6:
01705         k = IA[i];
01706         j = JA[k];
01707         pA = val+k*25; // &val[k*jump];
01708         px0 = x+j*5; // &x[j*nb];
01709         py = py0;
01710         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01711
01712         k ++;
01713         j = JA[k];
01714         pA = val+k*25; // &val[k*jump];
01715         px0 = x+j*5; // &x[j*nb];
01716         py = py0;
01717         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01718
01719         k ++;
01720         j = JA[k];
01721         pA = val+k*25; // &val[k*jump];
01722         px0 = x+j*5; // &x[j*nb];
01723         py = py0;
01724         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01725
01726         k ++;
01727         j = JA[k];
01728         pA = val+k*25; // &val[k*jump];
01729         px0 = x+j*5; // &x[j*nb];
01730         py = py0;
01731         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01732
01733         k ++;
01734         j = JA[k];
01735         pA = val+k*25; // &val[k*jump];
01736         px0 = x+j*5; // &x[j*nb];
01737         py = py0;
01738         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01739
01740         k ++;
01741         j = JA[k];
01742         pA = val+k*25; // &val[k*jump];
01743         px0 = x+j*5; // &x[j*nb];
01744         py = py0;
01745         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01746
01747         break;
01748
01749     case 7:
01750         k = IA[i];
01751         j = JA[k];
01752         pA = val+k*25; // &val[k*jump];
01753         px0 = x+j*5; // &x[j*nb];
01754         py = py0;
01755         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01756
01757         k ++;
01758         j = JA[k];
01759         pA = val+k*25; // &val[k*jump];
01760         px0 = x+j*5; // &x[j*nb];
01761         py = py0;
01762         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01763
01764         k ++;
01765         j = JA[k];
01766         pA = val+k*25; // &val[k*jump];
01767         px0 = x+j*5; // &x[j*nb];
01768         py = py0;
01769         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01770
01771         k ++;
01772         j = JA[k];
01773         pA = val+k*25; // &val[k*jump];
01774         px0 = x+j*5; // &x[j*nb];
01775         py = py0;
01776         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01777
01778         k ++;
01779         j = JA[k];
01780         pA = val+k*25; // &val[k*jump];
01781         px0 = x+j*5; // &x[j*nb];

```

```

01782         py = py0;
01783         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01784
01785         k ++;
01786         j = JA[k];
01787         pA = val+k*25; // &val[k*jump];
01788         px0 = x+j*5; // &x[j*nb];
01789         py = py0;
01790         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01791
01792         k ++;
01793         j = JA[k];
01794         pA = val+k*25; // &val[k*jump];
01795         px0 = x+j*5; // &x[j*nb];
01796         py = py0;
01797         fasp_blas_smat_ypAx_nc5( pA, px0, py );
01798
01799         break;
01800
01801     default:
01802         for (k = IA[i]; k < IA[i+1]; ++k)
01803         {
01804             j = JA[k];
01805             pA = val+k*25; // &val[k*jump];
01806             px0 = x+j*5; // &x[j*nb];
01807             py = py0;
01808             fasp_blas_smat_ypAx_nc5( pA, px0, py );
01809         }
01810         break;
01811     }
01812 }
01813 }
01814 }
01815 break;
01816
01817 case 7:
01818 {
01819     if (use_omp) {
01820 #ifdef _OPENMP
01821 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
01822 {
01823     myid = omp_get_thread_num();
01824     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01825     for (i=mybegin; i < myend; ++i)
01826     {
01827         py0 = &y[i*7];
01828         num_nnz_row = IA[i+1] - IA[i];
01829         switch(num_nnz_row)
01830         {
01831             case 3:
01832                 k = IA[i];
01833                 j = JA[k];
01834                 pA = val+k*49; // &val[k*jump];
01835                 px0 = x+j*7; // &x[j*nb];
01836                 py = py0;
01837                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
01838
01839                 k ++;
01840                 j = JA[k];
01841                 pA = val+k*49; // &val[k*jump];
01842                 px0 = x+j*7; // &x[j*nb];
01843                 py = py0;
01844                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
01845
01846                 k ++;
01847                 j = JA[k];
01848                 pA = val+k*49; // &val[k*jump];
01849                 px0 = x+j*7; // &x[j*nb];
01850                 py = py0;
01851                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
01852
01853                 break;
01854
01855             case 4:
01856                 k = IA[i];
01857                 j = JA[k];
01858                 pA = val+k*49; // &val[k*jump];
01859                 px0 = x+j*7; // &x[j*nb];
01860                 py = py0;
01861                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
01862

```

```

01863         k ++;
01864         j = JA[k];
01865         pA = val+k*49; // &val[k*jump];
01866         px0 = x+j*7; // &x[j*nb];
01867         py = py0;
01868         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01869
01870         k ++;
01871         j = JA[k];
01872         pA = val+k*49; // &val[k*jump];
01873         px0 = x+j*7; // &x[j*nb];
01874         py = py0;
01875         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01876
01877         k ++;
01878         j = JA[k];
01879         pA = val+k*49; // &val[k*jump];
01880         px0 = x+j*7; // &x[j*nb];
01881         py = py0;
01882         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01883
01884         break;
01885
01886     case 5:
01887         k = IA[i];
01888         j = JA[k];
01889         pA = val+k*49; // &val[k*jump];
01890         px0 = x+j*7; // &x[j*nb];
01891         py = py0;
01892         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01893
01894         k ++;
01895         j = JA[k];
01896         pA = val+k*49; // &val[k*jump];
01897         px0 = x+j*7; // &x[j*nb];
01898         py = py0;
01899         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01900
01901         k ++;
01902         j = JA[k];
01903         pA = val+k*49; // &val[k*jump];
01904         px0 = x+j*7; // &x[j*nb];
01905         py = py0;
01906         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01907
01908         k ++;
01909         j = JA[k];
01910         pA = val+k*49; // &val[k*jump];
01911         px0 = x+j*7; // &x[j*nb];
01912         py = py0;
01913         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01914
01915         k ++;
01916         j = JA[k];
01917         pA = val+k*49; // &val[k*jump];
01918         px0 = x+j*7; // &x[j*nb];
01919         py = py0;
01920         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01921
01922         break;
01923
01924     case 6:
01925         k = IA[i];
01926         j = JA[k];
01927         pA = val+k*49; // &val[k*jump];
01928         px0 = x+j*7; // &x[j*nb];
01929         py = py0;
01930         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01931
01932         k ++;
01933         j = JA[k];
01934         pA = val+k*49; // &val[k*jump];
01935         px0 = x+j*7; // &x[j*nb];
01936         py = py0;
01937         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01938
01939         k ++;
01940         j = JA[k];
01941         pA = val+k*49; // &val[k*jump];
01942         px0 = x+j*7; // &x[j*nb];
01943         py = py0;

```

```

01944         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01945
01946         k ++;
01947         j = JA[k];
01948         pA = val+k*49; // &val[k*jump];
01949         px0 = x+j*7; // &x[j*nb];
01950         py = py0;
01951         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01952
01953         k ++;
01954         j = JA[k];
01955         pA = val+k*49; // &val[k*jump];
01956         px0 = x+j*7; // &x[j*nb];
01957         py = py0;
01958         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01959
01960         k ++;
01961         j = JA[k];
01962         pA = val+k*49; // &val[k*jump];
01963         px0 = x+j*7; // &x[j*nb];
01964         py = py0;
01965         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01966
01967         break;
01968
01969     case 7:
01970         k = IA[i];
01971         j = JA[k];
01972         pA = val+k*49; // &val[k*jump];
01973         px0 = x+j*7; // &x[j*nb];
01974         py = py0;
01975         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01976
01977         k ++;
01978         j = JA[k];
01979         pA = val+k*49; // &val[k*jump];
01980         px0 = x+j*7; // &x[j*nb];
01981         py = py0;
01982         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01983
01984         k ++;
01985         j = JA[k];
01986         pA = val+k*49; // &val[k*jump];
01987         px0 = x+j*7; // &x[j*nb];
01988         py = py0;
01989         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01990
01991         k ++;
01992         j = JA[k];
01993         pA = val+k*49; // &val[k*jump];
01994         px0 = x+j*7; // &x[j*nb];
01995         py = py0;
01996         fasp_blas_smat_ypAx_nc7( pA, px0, py );
01997
01998         k ++;
01999         j = JA[k];
02000         pA = val+k*49; // &val[k*jump];
02001         px0 = x+j*7; // &x[j*nb];
02002         py = py0;
02003         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02004
02005         k ++;
02006         j = JA[k];
02007         pA = val+k*49; // &val[k*jump];
02008         px0 = x+j*7; // &x[j*nb];
02009         py = py0;
02010         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02011
02012         k ++;
02013         j = JA[k];
02014         pA = val+k*49; // &val[k*jump];
02015         px0 = x+j*7; // &x[j*nb];
02016         py = py0;
02017         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02018
02019         break;
02020
02021     default:
02022         for (k = IA[i]; k < IA[i+1]; ++k)
02023         {
02024             j = JA[k];

```

```

02025             pA = val+k*49; // &val[k*jump];
02026             px0 = x+j*7; // &x[j*nb];
02027             py = py0;
02028             fasp_blas_smat_ypAx_nc7( pA, px0, py );
02029         }
02030         break;
02031     }
02032 }
02033 }
02034 #endif
02035 }
02036 else {
02037     for (i = 0; i < ROW; ++i)
02038     {
02039         py0 = &y[i*7];
02040         num_nnz_row = IA[i+1] - IA[i];
02041         switch(num_nnz_row)
02042         {
02043             case 3:
02044                 k = IA[i];
02045                 j = JA[k];
02046                 pA = val+k*49; // &val[k*jump];
02047                 px0 = x+j*7; // &x[j*nb];
02048                 py = py0;
02049                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02050
02051                 k ++;
02052                 j = JA[k];
02053                 pA = val+k*49; // &val[k*jump];
02054                 px0 = x+j*7; // &x[j*nb];
02055                 py = py0;
02056                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02057
02058                 k ++;
02059                 j = JA[k];
02060                 pA = val+k*49; // &val[k*jump];
02061                 px0 = x+j*7; // &x[j*nb];
02062                 py = py0;
02063                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02064
02065                 break;
02066
02067             case 4:
02068                 k = IA[i];
02069                 j = JA[k];
02070                 pA = val+k*49; // &val[k*jump];
02071                 px0 = x+j*7; // &x[j*nb];
02072                 py = py0;
02073                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02074
02075                 k ++;
02076                 j = JA[k];
02077                 pA = val+k*49; // &val[k*jump];
02078                 px0 = x+j*7; // &x[j*nb];
02079                 py = py0;
02080                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02081
02082                 k ++;
02083                 j = JA[k];
02084                 pA = val+k*49; // &val[k*jump];
02085                 px0 = x+j*7; // &x[j*nb];
02086                 py = py0;
02087                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02088
02089                 k ++;
02090                 j = JA[k];
02091                 pA = val+k*49; // &val[k*jump];
02092                 px0 = x+j*7; // &x[j*nb];
02093                 py = py0;
02094                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02095
02096                 break;
02097
02098             case 5:
02099                 k = IA[i];
02100                 j = JA[k];
02101                 pA = val+k*49; // &val[k*jump];
02102                 px0 = x+j*7; // &x[j*nb];
02103                 py = py0;
02104                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
02105

```

```

02106         k ++;
02107         j = JA[k];
02108         pA = val+k*49; // &val[k*jump];
02109         px0 = x+j*7; // &x[j*nb];
02110         py = py0;
02111         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02112
02113         k ++;
02114         j = JA[k];
02115         pA = val+k*49; // &val[k*jump];
02116         px0 = x+j*7; // &x[j*nb];
02117         py = py0;
02118         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02119
02120         k ++;
02121         j = JA[k];
02122         pA = val+k*49; // &val[k*jump];
02123         px0 = x+j*7; // &x[j*nb];
02124         py = py0;
02125         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02126
02127         k ++;
02128         j = JA[k];
02129         pA = val+k*49; // &val[k*jump];
02130         px0 = x+j*7; // &x[j*nb];
02131         py = py0;
02132         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02133
02134         break;
02135
02136     case 6:
02137         k = IA[i];
02138         j = JA[k];
02139         pA = val+k*49; // &val[k*jump];
02140         px0 = x+j*7; // &x[j*nb];
02141         py = py0;
02142         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02143
02144         k ++;
02145         j = JA[k];
02146         pA = val+k*49; // &val[k*jump];
02147         px0 = x+j*7; // &x[j*nb];
02148         py = py0;
02149         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02150
02151         k ++;
02152         j = JA[k];
02153         pA = val+k*49; // &val[k*jump];
02154         px0 = x+j*7; // &x[j*nb];
02155         py = py0;
02156         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02157
02158         k ++;
02159         j = JA[k];
02160         pA = val+k*49; // &val[k*jump];
02161         px0 = x+j*7; // &x[j*nb];
02162         py = py0;
02163         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02164
02165         k ++;
02166         j = JA[k];
02167         pA = val+k*49; // &val[k*jump];
02168         px0 = x+j*7; // &x[j*nb];
02169         py = py0;
02170         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02171
02172         k ++;
02173         j = JA[k];
02174         pA = val+k*49; // &val[k*jump];
02175         px0 = x+j*7; // &x[j*nb];
02176         py = py0;
02177         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02178
02179         break;
02180
02181     case 7:
02182         k = IA[i];
02183         j = JA[k];
02184         pA = val+k*49; // &val[k*jump];
02185         px0 = x+j*7; // &x[j*nb];
02186         py = py0;

```

```

02187         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02188
02189         k ++;
02190         j = JA[k];
02191         pA = val+k*49; // &val[k*jump];
02192         px0 = x+j*7; // &x[j*nb];
02193         py = py0;
02194         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02195
02196         k ++;
02197         j = JA[k];
02198         pA = val+k*49; // &val[k*jump];
02199         px0 = x+j*7; // &x[j*nb];
02200         py = py0;
02201         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02202
02203         k ++;
02204         j = JA[k];
02205         pA = val+k*49; // &val[k*jump];
02206         px0 = x+j*7; // &x[j*nb];
02207         py = py0;
02208         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02209
02210         k ++;
02211         j = JA[k];
02212         pA = val+k*49; // &val[k*jump];
02213         px0 = x+j*7; // &x[j*nb];
02214         py = py0;
02215         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02216
02217         k ++;
02218         j = JA[k];
02219         pA = val+k*49; // &val[k*jump];
02220         px0 = x+j*7; // &x[j*nb];
02221         py = py0;
02222         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02223
02224         k ++;
02225         j = JA[k];
02226         pA = val+k*49; // &val[k*jump];
02227         px0 = x+j*7; // &x[j*nb];
02228         py = py0;
02229         fasp_blas_smat_ypAx_nc7( pA, px0, py );
02230
02231         break;
02232
02233     default:
02234         for (k = IA[i]; k < IA[i+1]; ++k)
02235         {
02236             j = JA[k];
02237             pA = val+k*49; // &val[k*jump];
02238             px0 = x+j*7; // &x[j*nb];
02239             py = py0;
02240             fasp_blas_smat_ypAx_nc7( pA, px0, py );
02241         }
02242         break;
02243     }
02244 }
02245 }
02246 }
02247     break;
02248
02249     default:
02250     {
02251         if (use_omp) {
02252 #ifdef _OPENMP
02253 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
02254         {
02255             myid = omp_get_thread_num();
02256             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
02257             for (i=mybegin; i < myend; ++i)
02258             {
02259                 py0 = &y[i*nb];
02260                 num_nnz_row = IA[i+1] - IA[i];
02261                 switch(num_nnz_row)
02262                 {
02263                     case 3:
02264                         k = IA[i];
02265                         j = JA[k];
02266                         pA = val+k*jump; // &val[k*jump];
02267                         px0 = x+j*nb; // &x[j*nb];

```

```

02268         py = py0;
02269         fasp_blas_smat_ypAx( pA, px0, py, nb );
02270
02271         k ++;
02272         j = JA[k];
02273         pA = val+k*jump; // &val[k*jump];
02274         px0 = x+j*nb; // &x[j*nb];
02275         py = py0;
02276         fasp_blas_smat_ypAx( pA, px0, py, nb );
02277
02278         k ++;
02279         j = JA[k];
02280         pA = val+k*jump; // &val[k*jump];
02281         px0 = x+j*nb; // &x[j*nb];
02282         py = py0;
02283         fasp_blas_smat_ypAx( pA, px0, py, nb );
02284
02285         break;
02286
02287     case 4:
02288         k = IA[i];
02289         j = JA[k];
02290         pA = val+k*jump; // &val[k*jump];
02291         px0 = x+j*nb; // &x[j*nb];
02292         py = py0;
02293         fasp_blas_smat_ypAx( pA, px0, py, nb );
02294
02295         k ++;
02296         j = JA[k];
02297         pA = val+k*jump; // &val[k*jump];
02298         px0 = x+j*nb; // &x[j*nb];
02299         py = py0;
02300         fasp_blas_smat_ypAx( pA, px0, py, nb );
02301
02302         k ++;
02303         j = JA[k];
02304         pA = val+k*jump; // &val[k*jump];
02305         px0 = x+j*nb; // &x[j*nb];
02306         py = py0;
02307         fasp_blas_smat_ypAx( pA, px0, py, nb );
02308
02309         k ++;
02310         j = JA[k];
02311         pA = val+k*jump; // &val[k*jump];
02312         px0 = x+j*nb; // &x[j*nb];
02313         py = py0;
02314         fasp_blas_smat_ypAx( pA, px0, py, nb );
02315
02316         break;
02317
02318     case 5:
02319         k = IA[i];
02320         j = JA[k];
02321         pA = val+k*jump; // &val[k*jump];
02322         px0 = x+j*nb; // &x[j*nb];
02323         py = py0;
02324         fasp_blas_smat_ypAx( pA, px0, py, nb );
02325
02326         k ++;
02327         j = JA[k];
02328         pA = val+k*jump; // &val[k*jump];
02329         px0 = x+j*nb; // &x[j*nb];
02330         py = py0;
02331         fasp_blas_smat_ypAx( pA, px0, py, nb );
02332
02333         k ++;
02334         j = JA[k];
02335         pA = val+k*jump; // &val[k*jump];
02336         px0 = x+j*nb; // &x[j*nb];
02337         py = py0;
02338         fasp_blas_smat_ypAx( pA, px0, py, nb );
02339
02340         k ++;
02341         j = JA[k];
02342         pA = val+k*jump; // &val[k*jump];
02343         px0 = x+j*nb; // &x[j*nb];
02344         py = py0;
02345         fasp_blas_smat_ypAx( pA, px0, py, nb );
02346
02347         k ++;
02348         j = JA[k];

```



```

02349         pA = val+k*jump; // &val[k*jump];
02350         px0 = x+j*nb; // &x[j*nb];
02351         py = py0;
02352         fasp_blas_smat_ypAx( pA, px0, py, nb );
02353
02354         break;
02355
02356     case 6:
02357         k = IA[i];
02358         j = JA[k];
02359         pA = val+k*jump; // &val[k*jump];
02360         px0 = x+j*nb; // &x[j*nb];
02361         py = py0;
02362         fasp_blas_smat_ypAx( pA, px0, py, nb );
02363
02364         k ++;
02365         j = JA[k];
02366         pA = val+k*jump; // &val[k*jump];
02367         px0 = x+j*nb; // &x[j*nb];
02368         py = py0;
02369         fasp_blas_smat_ypAx( pA, px0, py, nb );
02370
02371         k ++;
02372         j = JA[k];
02373         pA = val+k*jump; // &val[k*jump];
02374         px0 = x+j*nb; // &x[j*nb];
02375         py = py0;
02376         fasp_blas_smat_ypAx( pA, px0, py, nb );
02377
02378         k ++;
02379         j = JA[k];
02380         pA = val+k*jump; // &val[k*jump];
02381         px0 = x+j*nb; // &x[j*nb];
02382         py = py0;
02383         fasp_blas_smat_ypAx( pA, px0, py, nb );
02384
02385         k ++;
02386         j = JA[k];
02387         pA = val+k*jump; // &val[k*jump];
02388         px0 = x+j*nb; // &x[j*nb];
02389         py = py0;
02390         fasp_blas_smat_ypAx( pA, px0, py, nb );
02391
02392         k ++;
02393         j = JA[k];
02394         pA = val+k*jump; // &val[k*jump];
02395         px0 = x+j*nb; // &x[j*nb];
02396         py = py0;
02397         fasp_blas_smat_ypAx( pA, px0, py, nb );
02398
02399         break;
02400
02401     case 7:
02402         k = IA[i];
02403         j = JA[k];
02404         pA = val+k*jump; // &val[k*jump];
02405         px0 = x+j*nb; // &x[j*nb];
02406         py = py0;
02407         fasp_blas_smat_ypAx( pA, px0, py, nb );
02408
02409         k ++;
02410         j = JA[k];
02411         pA = val+k*jump; // &val[k*jump];
02412         px0 = x+j*nb; // &x[j*nb];
02413         py = py0;
02414         fasp_blas_smat_ypAx( pA, px0, py, nb );
02415
02416         k ++;
02417         j = JA[k];
02418         pA = val+k*jump; // &val[k*jump];
02419         px0 = x+j*nb; // &x[j*nb];
02420         py = py0;
02421         fasp_blas_smat_ypAx( pA, px0, py, nb );
02422
02423         k ++;
02424         j = JA[k];
02425         pA = val+k*jump; // &val[k*jump];
02426         px0 = x+j*nb; // &x[j*nb];
02427         py = py0;
02428         fasp_blas_smat_ypAx( pA, px0, py, nb );
02429

```

```

02430         k ++;
02431         j = JA[k];
02432         pA = val+k*jump; // &val[k*jump];
02433         px0 = x+j*nb; // &x[j*nb];
02434         py = py0;
02435         fasp_blas_smat_ypAx( pA, px0, py, nb );
02436
02437         k ++;
02438         j = JA[k];
02439         pA = val+k*jump; // &val[k*jump];
02440         px0 = x+j*nb; // &x[j*nb];
02441         py = py0;
02442         fasp_blas_smat_ypAx( pA, px0, py, nb );
02443
02444         k ++;
02445         j = JA[k];
02446         pA = val+k*jump; // &val[k*jump];
02447         px0 = x+j*nb; // &x[j*nb];
02448         py = py0;
02449         fasp_blas_smat_ypAx( pA, px0, py, nb );
02450
02451         break;
02452
02453     default:
02454         for (k = IA[i]; k < IA[i+1]; ++k)
02455         {
02456             j = JA[k];
02457             pA = val+k*jump; // &val[k*jump];
02458             px0 = x+j*nb; // &x[j*nb];
02459             py = py0;
02460             fasp_blas_smat_ypAx( pA, px0, py, nb );
02461         }
02462         break;
02463     }
02464 }
02465 }
02466 #endif
02467 }
02468 else {
02469     for (i = 0; i < ROW; ++i)
02470     {
02471         py0 = &y[i*nb];
02472         num_nnz_row = IA[i+1] - IA[i];
02473         switch(num_nnz_row)
02474         {
02475             case 3:
02476                 k = IA[i];
02477                 j = JA[k];
02478                 pA = val+k*jump; // &val[k*jump];
02479                 px0 = x+j*nb; // &x[j*nb];
02480                 py = py0;
02481                 fasp_blas_smat_ypAx( pA, px0, py, nb );
02482
02483                 k ++;
02484                 j = JA[k];
02485                 pA = val+k*jump; // &val[k*jump];
02486                 px0 = x+j*nb; // &x[j*nb];
02487                 py = py0;
02488                 fasp_blas_smat_ypAx( pA, px0, py, nb );
02489
02490                 k ++;
02491                 j = JA[k];
02492                 pA = val+k*jump; // &val[k*jump];
02493                 px0 = x+j*nb; // &x[j*nb];
02494                 py = py0;
02495                 fasp_blas_smat_ypAx( pA, px0, py, nb );
02496
02497                 break;
02498
02499             case 4:
02500                 k = IA[i];
02501                 j = JA[k];
02502                 pA = val+k*jump; // &val[k*jump];
02503                 px0 = x+j*nb; // &x[j*nb];
02504                 py = py0;
02505                 fasp_blas_smat_ypAx( pA, px0, py, nb );
02506
02507                 k ++;
02508                 j = JA[k];
02509                 pA = val+k*jump; // &val[k*jump];
02510                 px0 = x+j*nb; // &x[j*nb];

```

```

02511         py = py0;
02512         fasp_blas_smat_ypAx( pA, px0, py, nb );
02513
02514         k ++;
02515         j = JA[k];
02516         pA = val+k*jump; // &val[k*jump];
02517         px0 = x+j*nb; // &x[j*nb];
02518         py = py0;
02519         fasp_blas_smat_ypAx( pA, px0, py, nb );
02520
02521         k ++;
02522         j = JA[k];
02523         pA = val+k*jump; // &val[k*jump];
02524         px0 = x+j*nb; // &x[j*nb];
02525         py = py0;
02526         fasp_blas_smat_ypAx( pA, px0, py, nb );
02527
02528         break;
02529
02530     case 5:
02531         k = IA[i];
02532         j = JA[k];
02533         pA = val+k*jump; // &val[k*jump];
02534         px0 = x+j*nb; // &x[j*nb];
02535         py = py0;
02536         fasp_blas_smat_ypAx( pA, px0, py, nb );
02537
02538         k ++;
02539         j = JA[k];
02540         pA = val+k*jump; // &val[k*jump];
02541         px0 = x+j*nb; // &x[j*nb];
02542         py = py0;
02543         fasp_blas_smat_ypAx( pA, px0, py, nb );
02544
02545         k ++;
02546         j = JA[k];
02547         pA = val+k*jump; // &val[k*jump];
02548         px0 = x+j*nb; // &x[j*nb];
02549         py = py0;
02550         fasp_blas_smat_ypAx( pA, px0, py, nb );
02551
02552         k ++;
02553         j = JA[k];
02554         pA = val+k*jump; // &val[k*jump];
02555         px0 = x+j*nb; // &x[j*nb];
02556         py = py0;
02557         fasp_blas_smat_ypAx( pA, px0, py, nb );
02558
02559         k ++;
02560         j = JA[k];
02561         pA = val+k*jump; // &val[k*jump];
02562         px0 = x+j*nb; // &x[j*nb];
02563         py = py0;
02564         fasp_blas_smat_ypAx( pA, px0, py, nb );
02565
02566         break;
02567
02568     case 6:
02569         k = IA[i];
02570         j = JA[k];
02571         pA = val+k*jump; // &val[k*jump];
02572         px0 = x+j*nb; // &x[j*nb];
02573         py = py0;
02574         fasp_blas_smat_ypAx( pA, px0, py, nb );
02575
02576         k ++;
02577         j = JA[k];
02578         pA = val+k*jump; // &val[k*jump];
02579         px0 = x+j*nb; // &x[j*nb];
02580         py = py0;
02581         fasp_blas_smat_ypAx( pA, px0, py, nb );
02582
02583         k ++;
02584         j = JA[k];
02585         pA = val+k*jump; // &val[k*jump];
02586         px0 = x+j*nb; // &x[j*nb];
02587         py = py0;
02588         fasp_blas_smat_ypAx( pA, px0, py, nb );
02589
02590         k ++;
02591         j = JA[k];

```

```

02592         pA = val+k*jump; // &val[k*jump];
02593         px0 = x+j*nb; // &x[j*nb];
02594         py = py0;
02595         fasp_blas_smat_ypAx( pA, px0, py, nb );
02596
02597         k ++;
02598         j = JA[k];
02599         pA = val+k*jump; // &val[k*jump];
02600         px0 = x+j*nb; // &x[j*nb];
02601         py = py0;
02602         fasp_blas_smat_ypAx( pA, px0, py, nb );
02603
02604         k ++;
02605         j = JA[k];
02606         pA = val+k*jump; // &val[k*jump];
02607         px0 = x+j*nb; // &x[j*nb];
02608         py = py0;
02609         fasp_blas_smat_ypAx( pA, px0, py, nb );
02610
02611         break;
02612
02613     case 7:
02614         k = IA[i];
02615         j = JA[k];
02616         pA = val+k*jump; // &val[k*jump];
02617         px0 = x+j*nb; // &x[j*nb];
02618         py = py0;
02619         fasp_blas_smat_ypAx( pA, px0, py, nb );
02620
02621         k ++;
02622         j = JA[k];
02623         pA = val+k*jump; // &val[k*jump];
02624         px0 = x+j*nb; // &x[j*nb];
02625         py = py0;
02626         fasp_blas_smat_ypAx( pA, px0, py, nb );
02627
02628         k ++;
02629         j = JA[k];
02630         pA = val+k*jump; // &val[k*jump];
02631         px0 = x+j*nb; // &x[j*nb];
02632         py = py0;
02633         fasp_blas_smat_ypAx( pA, px0, py, nb );
02634
02635         k ++;
02636         j = JA[k];
02637         pA = val+k*jump; // &val[k*jump];
02638         px0 = x+j*nb; // &x[j*nb];
02639         py = py0;
02640         fasp_blas_smat_ypAx( pA, px0, py, nb );
02641
02642         k ++;
02643         j = JA[k];
02644         pA = val+k*jump; // &val[k*jump];
02645         px0 = x+j*nb; // &x[j*nb];
02646         py = py0;
02647         fasp_blas_smat_ypAx( pA, px0, py, nb );
02648
02649         k ++;
02650         j = JA[k];
02651         pA = val+k*jump; // &val[k*jump];
02652         px0 = x+j*nb; // &x[j*nb];
02653         py = py0;
02654         fasp_blas_smat_ypAx( pA, px0, py, nb );
02655
02656         k ++;
02657         j = JA[k];
02658         pA = val+k*jump; // &val[k*jump];
02659         px0 = x+j*nb; // &x[j*nb];
02660         py = py0;
02661         fasp_blas_smat_ypAx( pA, px0, py, nb );
02662
02663         break;
02664
02665     default:
02666         for (k = IA[i]; k < IA[i+1]; ++k)
02667         {
02668             j = JA[k];
02669             pA = val+k*jump; // &val[k*jump];
02670             px0 = x+j*nb; // &x[j*nb];
02671             py = py0;
02672             fasp_blas_smat_ypAx( pA, px0, py, nb );

```

```

02673     }
02674     break;
02675 }
02676 }
02677 }
02678 }
02679     break;
02680 }
02681 }
02682
02697 void fasp_blas_dbsr_mvx_agg (const dBSRmat *A,
02698                             const REAL *x,
02699                             REAL *y)
02700 {
02701     /* members of A */
02702     const INT ROW = A->ROW;
02703     const INT nb = A->nb;
02704     const INT size = ROW*nb;
02705     const INT *IA = A->IA;
02706     const INT *JA = A->JA;
02707
02708     /* local variables */
02709     const REAL *px0 = NULL;
02710     REAL *py0 = NULL, *py = NULL;
02711     INT i,j,k, num_nnz_row;
02712     SHORT use_omp = FALSE;
02713
02714 #ifdef _OPENMP
02715     const REAL *val = A->val;
02716     const REAL *pA;
02717     INT myid, mybegin, myend, nthreads;
02718     if ( ROW > OPENMP_HOLDS ) {
02719         use_omp = TRUE;
02720         nthreads = fasp_get_num_threads();
02721     }
02722 #endif
02723
02724     //-----
02725     // zero out 'y'
02726     //-----
02727     fasp_darray_set(size, y, 0.0);
02728
02729     //-----
02730     // y = A*x (Core Computation)
02731     // each non-zero block elements are stored in row-major order
02732     //-----
02733
02734     switch (nb)
02735     {
02736     case 3:
02737     {
02738         if (use_omp) {
02739 #ifdef _OPENMP
02740 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
02741 {
02742             myid = omp_get_thread_num();
02743             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
02744             for (i=mybegin; i < myend; ++i) {
02745                 py0 = &y[i*3];
02746                 num_nnz_row = IA[i+1] - IA[i];
02747                 switch(num_nnz_row) {
02748                     case 3:
02749                         k = IA[i];
02750                         j = JA[k];
02751                         pA = val+k*9;
02752                         px0 = x+j*3;
02753                         py = py0;
02754                         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02755
02756                         k++;
02757                         j = JA[k];
02758                         pA = val+k*9;
02759                         px0 = x+j*3;
02760                         py = py0;
02761                         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02762
02763                         k++;
02764                         j = JA[k];
02765                         pA = val+k*9;
02766                         px0 = x+j*3;
02767                         py = py0;

```

```

02768         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02769
02770     break;
02771
02772     case 4:
02773         k = IA[i];
02774         j = JA[k];
02775         pA = val+k*9;
02776         px0 = x+j*3;
02777         py = py0;
02778         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02779
02780         k ++;
02781         j = JA[k];
02782         pA = val+k*9;
02783         px0 = x+j*3;
02784         py = py0;
02785         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02786
02787         k ++;
02788         j = JA[k];
02789         pA = val+k*9;
02790         px0 = x+j*3;
02791         py = py0;
02792         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02793
02794         k ++;
02795         j = JA[k];
02796         pA = val+k*9;
02797         px0 = x+j*3;
02798         py = py0;
02799         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02800
02801     break;
02802
02803     case 5:
02804         k = IA[i];
02805         j = JA[k];
02806         pA = val+k*9;
02807         px0 = x+j*3;
02808         py = py0;
02809         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02810
02811         k ++;
02812         j = JA[k];
02813         pA = val+k*9;
02814         px0 = x+j*3;
02815         py = py0;
02816         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02817
02818         k ++;
02819         j = JA[k];
02820         pA = val+k*9;
02821         px0 = x+j*3;
02822         py = py0;
02823         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02824
02825         k ++;
02826         j = JA[k];
02827         pA = val+k*9;
02828         px0 = x+j*3;
02829         py = py0;
02830         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02831
02832         k ++;
02833         j = JA[k];
02834         pA = val+k*9;
02835         px0 = x+j*3;
02836         py = py0;
02837         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02838
02839     break;
02840
02841     case 6:
02842         k = IA[i];
02843         j = JA[k];
02844         pA = val+k*9;
02845         px0 = x+j*3;
02846         py = py0;
02847         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02848

```

```

02849         k ++;
02850         j = JA[k];
02851         pA = val+k*9;
02852         px0 = x+j*3;
02853         py = py0;
02854         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02855
02856         k ++;
02857         j = JA[k];
02858         pA = val+k*9;
02859         px0 = x+j*3;
02860         py = py0;
02861         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02862
02863         k ++;
02864         j = JA[k];
02865         pA = val+k*9;
02866         px0 = x+j*3;
02867         py = py0;
02868         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02869
02870         k ++;
02871         j = JA[k];
02872         pA = val+k*9;
02873         px0 = x+j*3;
02874         py = py0;
02875         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02876
02877         k ++;
02878         j = JA[k];
02879         pA = val+k*9;
02880         px0 = x+j*3;
02881         py = py0;
02882         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02883
02884         break;
02885
02886     case 7:
02887         k = IA[i];
02888         j = JA[k];
02889         pA = val+k*9;
02890         px0 = x+j*3;
02891         py = py0;
02892         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02893
02894         k ++;
02895         j = JA[k];
02896         pA = val+k*9;
02897         px0 = x+j*3;
02898         py = py0;
02899         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02900
02901         k ++;
02902         j = JA[k];
02903         pA = val+k*9;
02904         px0 = x+j*3;
02905         py = py0;
02906         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02907
02908         k ++;
02909         j = JA[k];
02910         pA = val+k*9;
02911         px0 = x+j*3;
02912         py = py0;
02913         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02914
02915         k ++;
02916         j = JA[k];
02917         pA = val+k*9;
02918         px0 = x+j*3;
02919         py = py0;
02920         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02921
02922         k ++;
02923         j = JA[k];
02924         pA = val+k*9;
02925         px0 = x+j*3;
02926         py = py0;
02927         fasp_blas_smat_ypAx_nc3( pA, px0, py );
02928
02929         k ++;

```

```

02930             j = JA[k];
02931             pA = val+k*9;
02932             px0 = x+j*3;
02933             py = py0;
02934             fasp_blas_smat_ypAx_nc3( pA, px0, py );
02935
02936             break;
02937
02938         default:
02939             for (k = IA[i]; k < IA[i+1]; ++k)
02940             {
02941                 j = JA[k];
02942                 pA = val+k*9;
02943                 px0 = x+j*3;
02944                 py = py0;
02945                 fasp_blas_smat_ypAx_nc3( pA, px0, py );
02946             }
02947             break;
02948         }
02949     }
02950 }
02951 #endif
02952 }
02953 else {
02954     for (i = 0; i < ROW; ++i) {
02955         py0 = &y[i*3];
02956         num_nnz_row = IA[i+1] - IA[i];
02957         switch(num_nnz_row) {
02958             case 3:
02959                 k = IA[i];
02960                 j = JA[k];
02961                 px0 = x+j*3; // &x[j*nb];
02962                 py = py0;
02963                 py[0] += px0[0];
02964                 py[1] += px0[1];
02965                 py[2] += px0[2];
02966
02967                 k ++;
02968                 j = JA[k];
02969                 px0 = x+j*3; // &x[j*nb];
02970                 py = py0;
02971                 py[0] += px0[0];
02972                 py[1] += px0[1];
02973                 py[2] += px0[2];
02974
02975                 k ++;
02976                 j = JA[k];
02977                 px0 = x+j*3; // &x[j*nb];
02978                 py = py0;
02979                 py[0] += px0[0];
02980                 py[1] += px0[1];
02981                 py[2] += px0[2];
02982
02983                 break;
02984
02985             case 4:
02986                 k = IA[i];
02987                 j = JA[k];
02988                 px0 = x+j*3; // &x[j*nb];
02989                 py = py0;
02990                 py[0] += px0[0];
02991                 py[1] += px0[1];
02992                 py[2] += px0[2];
02993
02994                 k ++;
02995                 j = JA[k];
02996                 px0 = x+j*3; // &x[j*nb];
02997                 py = py0;
02998                 py[0] += px0[0];
02999                 py[1] += px0[1];
03000                 py[2] += px0[2];
03001
03002                 k ++;
03003                 j = JA[k];
03004                 px0 = x+j*3; // &x[j*nb];
03005                 py = py0;
03006                 py[0] += px0[0];
03007                 py[1] += px0[1];
03008                 py[2] += px0[2];
03009
03010                 k ++;

```



```
03011         j = JA[k];
03012         px0 = x+j*3; // &x[j*nb];
03013         py = py0;
03014         py[0] += px0[0];
03015         py[1] += px0[1];
03016         py[2] += px0[2];
03017
03018         break;
03019
03020     case 5:
03021         k = IA[i];
03022         j = JA[k];
03023         px0 = x+j*3; // &x[j*nb];
03024         py = py0;
03025         py[0] += px0[0];
03026         py[1] += px0[1];
03027         py[2] += px0[2];
03028
03029         k ++;
03030         j = JA[k];
03031         px0 = x+j*3; // &x[j*nb];
03032         py = py0;
03033         py[0] += px0[0];
03034         py[1] += px0[1];
03035         py[2] += px0[2];
03036
03037         k ++;
03038         j = JA[k];
03039         px0 = x+j*3; // &x[j*nb];
03040         py = py0;
03041         py[0] += px0[0];
03042         py[1] += px0[1];
03043         py[2] += px0[2];
03044
03045         k ++;
03046         j = JA[k];
03047         px0 = x+j*3; // &x[j*nb];
03048         py = py0;
03049         py[0] += px0[0];
03050         py[1] += px0[1];
03051         py[2] += px0[2];
03052
03053         k ++;
03054         j = JA[k];
03055         px0 = x+j*3; // &x[j*nb];
03056         py = py0;
03057         py[0] += px0[0];
03058         py[1] += px0[1];
03059         py[2] += px0[2];
03060
03061         break;
03062
03063     case 6:
03064         k = IA[i];
03065         j = JA[k];
03066         px0 = x+j*3; // &x[j*nb];
03067         py = py0;
03068         py[0] += px0[0];
03069         py[1] += px0[1];
03070         py[2] += px0[2];
03071
03072         k ++;
03073         j = JA[k];
03074         px0 = x+j*3; // &x[j*nb];
03075         py = py0;
03076         py[0] += px0[0];
03077         py[1] += px0[1];
03078         py[2] += px0[2];
03079
03080         k ++;
03081         j = JA[k];
03082         px0 = x+j*3; // &x[j*nb];
03083         py = py0;
03084         py[0] += px0[0];
03085         py[1] += px0[1];
03086         py[2] += px0[2];
03087
03088         k ++;
03089         j = JA[k];
03090         px0 = x+j*3; // &x[j*nb];
03091         py = py0;
```

```
03092         py[0] += px0[0];
03093         py[1] += px0[1];
03094         py[2] += px0[2];
03095
03096         k++;
03097         j = JA[k];
03098         px0 = x+j*3; // &x[j*nb];
03099         py = py0;
03100         py[0] += px0[0];
03101         py[1] += px0[1];
03102         py[2] += px0[2];
03103
03104         k++;
03105         j = JA[k];
03106         px0 = x+j*3; // &x[j*nb];
03107         py = py0;
03108         py[0] += px0[0];
03109         py[1] += px0[1];
03110         py[2] += px0[2];
03111
03112         break;
03113
03114     case 7:
03115         k = IA[i];
03116         j = JA[k];
03117         px0 = x+j*3; // &x[j*nb];
03118         py = py0;
03119         py[0] += px0[0];
03120         py[1] += px0[1];
03121         py[2] += px0[2];
03122
03123         k++;
03124         j = JA[k];
03125         px0 = x+j*3; // &x[j*nb];
03126         py = py0;
03127         py[0] += px0[0];
03128         py[1] += px0[1];
03129         py[2] += px0[2];
03130
03131         k++;
03132         j = JA[k];
03133         px0 = x+j*3; // &x[j*nb];
03134         py = py0;
03135         py[0] += px0[0];
03136         py[1] += px0[1];
03137         py[2] += px0[2];
03138
03139         k++;
03140         j = JA[k];
03141         px0 = x+j*3; // &x[j*nb];
03142         py = py0;
03143         py[0] += px0[0];
03144         py[1] += px0[1];
03145         py[2] += px0[2];
03146
03147         k++;
03148         j = JA[k];
03149         px0 = x+j*3; // &x[j*nb];
03150         py = py0;
03151         py[0] += px0[0];
03152         py[1] += px0[1];
03153         py[2] += px0[2];
03154
03155         k++;
03156         j = JA[k];
03157         px0 = x+j*3; // &x[j*nb];
03158         py = py0;
03159         py[0] += px0[0];
03160         py[1] += px0[1];
03161         py[2] += px0[2];
03162
03163         k++;
03164         j = JA[k];
03165         px0 = x+j*3; // &x[j*nb];
03166         py = py0;
03167         py[0] += px0[0];
03168         py[1] += px0[1];
03169         py[2] += px0[2];
03170
03171         break;
03172
```

```

03173         default:
03174             for (k = IA[i]; k < IA[i+1]; ++k) {
03175                 j = JA[k];
03176                 px0 = x+j*3; // &x[j*nb];
03177                 py = py0;
03178                 py[0] += px0[0];
03179                 py[1] += px0[1];
03180                 py[2] += px0[2];
03181             }
03182             break;
03183         }
03184     }
03185 }
03186 }
03187 break;
03188
03189 case 5:
03190 {
03191     if (use_omp) {
03192 #ifdef _OPENMP
03193 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
03194 {
03195     myid = omp_get_thread_num();
03196     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
03197     for (i=mybegin; i < myend; ++i) {
03198         py0 = &y[i*5];
03199         num_nnz_row = IA[i+1] - IA[i];
03200         switch(num_nnz_row) {
03201
03202             case 3:
03203                 k = IA[i];
03204                 j = JA[k];
03205                 pA = val+k*25; // &val[k*jump];
03206                 px0 = x+j*5; // &x[j*nb];
03207                 py = py0;
03208                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03209
03210                 k ++;
03211                 j = JA[k];
03212                 pA = val+k*25; // &val[k*jump];
03213                 px0 = x+j*5; // &x[j*nb];
03214                 py = py0;
03215                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03216
03217                 k ++;
03218                 j = JA[k];
03219                 pA = val+k*25; // &val[k*jump];
03220                 px0 = x+j*5; // &x[j*nb];
03221                 py = py0;
03222                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03223
03224                 break;
03225
03226             case 4:
03227                 k = IA[i];
03228                 j = JA[k];
03229                 pA = val+k*25; // &val[k*jump];
03230                 px0 = x+j*5; // &x[j*nb];
03231                 py = py0;
03232                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03233
03234                 k ++;
03235                 j = JA[k];
03236                 pA = val+k*25; // &val[k*jump];
03237                 px0 = x+j*5; // &x[j*nb];
03238                 py = py0;
03239                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03240
03241                 k ++;
03242                 j = JA[k];
03243                 pA = val+k*25; // &val[k*jump];
03244                 px0 = x+j*5; // &x[j*nb];
03245                 py = py0;
03246                 fasp_blas_smat_ypAx_nc5( pA, px0, py );
03247
03248                 k ++;
03249                 j = JA[k];
03250                 pA = val+k*25; // &val[k*jump];
03251                 px0 = x+j*5; // &x[j*nb];
03252                 py = py0;
03253                 fasp_blas_smat_ypAx_nc5( pA, px0, py );

```

```

03254
03255         break;
03256
03257     case 5:
03258         k = IA[i];
03259         j = JA[k];
03260         pA = val+k*25; // &val[k*jump];
03261         px0 = x+j*5; // &x[j*nb];
03262         py = py0;
03263         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03264
03265         k ++;
03266         j = JA[k];
03267         pA = val+k*25; // &val[k*jump];
03268         px0 = x+j*5; // &x[j*nb];
03269         py = py0;
03270         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03271
03272         k ++;
03273         j = JA[k];
03274         pA = val+k*25; // &val[k*jump];
03275         px0 = x+j*5; // &x[j*nb];
03276         py = py0;
03277         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03278
03279         k ++;
03280         j = JA[k];
03281         pA = val+k*25; // &val[k*jump];
03282         px0 = x+j*5; // &x[j*nb];
03283         py = py0;
03284         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03285
03286         k ++;
03287         j = JA[k];
03288         pA = val+k*25; // &val[k*jump];
03289         px0 = x+j*5; // &x[j*nb];
03290         py = py0;
03291         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03292
03293     break;
03294
03295     case 6:
03296         k = IA[i];
03297         j = JA[k];
03298         pA = val+k*25; // &val[k*jump];
03299         px0 = x+j*5; // &x[j*nb];
03300         py = py0;
03301         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03302
03303         k ++;
03304         j = JA[k];
03305         pA = val+k*25; // &val[k*jump];
03306         px0 = x+j*5; // &x[j*nb];
03307         py = py0;
03308         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03309
03310         k ++;
03311         j = JA[k];
03312         pA = val+k*25; // &val[k*jump];
03313         px0 = x+j*5; // &x[j*nb];
03314         py = py0;
03315         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03316
03317         k ++;
03318         j = JA[k];
03319         pA = val+k*25; // &val[k*jump];
03320         px0 = x+j*5; // &x[j*nb];
03321         py = py0;
03322         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03323
03324         k ++;
03325         j = JA[k];
03326         pA = val+k*25; // &val[k*jump];
03327         px0 = x+j*5; // &x[j*nb];
03328         py = py0;
03329         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03330
03331         k ++;
03332         j = JA[k];
03333         pA = val+k*25; // &val[k*jump];
03334         px0 = x+j*5; // &x[j*nb];

```

```

03335         py = py0;
03336         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03337
03338         break;
03339
03340     case 7:
03341         k = IA[i];
03342         j = JA[k];
03343         pA = val+k*25; // &val[k*jump];
03344         px0 = x+j*5; // &x[j*nb];
03345         py = py0;
03346         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03347
03348         k++;
03349         j = JA[k];
03350         pA = val+k*25; // &val[k*jump];
03351         px0 = x+j*5; // &x[j*nb];
03352         py = py0;
03353         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03354
03355         k++;
03356         j = JA[k];
03357         pA = val+k*25; // &val[k*jump];
03358         px0 = x+j*5; // &x[j*nb];
03359         py = py0;
03360         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03361
03362         k++;
03363         j = JA[k];
03364         pA = val+k*25; // &val[k*jump];
03365         px0 = x+j*5; // &x[j*nb];
03366         py = py0;
03367         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03368
03369         k++;
03370         j = JA[k];
03371         pA = val+k*25; // &val[k*jump];
03372         px0 = x+j*5; // &x[j*nb];
03373         py = py0;
03374         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03375
03376         k++;
03377         j = JA[k];
03378         pA = val+k*25; // &val[k*jump];
03379         px0 = x+j*5; // &x[j*nb];
03380         py = py0;
03381         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03382
03383         k++;
03384         j = JA[k];
03385         pA = val+k*25; // &val[k*jump];
03386         px0 = x+j*5; // &x[j*nb];
03387         py = py0;
03388         fasp_blas_smat_ypAx_nc5( pA, px0, py );
03389
03390         break;
03391
03392     default:
03393         for (k = IA[i]; k < IA[i+1]; ++k)
03394         {
03395             j = JA[k];
03396             pA = val+k*25; // &val[k*jump];
03397             px0 = x+j*5; // &x[j*nb];
03398             py = py0;
03399             fasp_blas_smat_ypAx_nc5( pA, px0, py );
03400         }
03401         break;
03402     }
03403 }
03404 #endif
03405
03406 }
03407 else {
03408     for (i = 0; i < ROW; ++i) {
03409         py0 = &y[i*5];
03410         num_nnz_row = IA[i+1] - IA[i];
03411         switch(num_nnz_row) {
03412
03413             case 3:
03414                 k = IA[i];
03415                 j = JA[k];

```

```

03416         px0 = x+j*5; // &x[j*nb];
03417         py = py0;
03418         py[0] += px0[0];
03419         py[1] += px0[1];
03420         py[2] += px0[2];
03421         py[3] += px0[3];
03422         py[4] += px0[4];
03423
03424         k ++;
03425         j = JA[k];
03426         px0 = x+j*5; // &x[j*nb];
03427         py = py0;
03428         py[0] += px0[0];
03429         py[1] += px0[1];
03430         py[2] += px0[2];
03431         py[3] += px0[3];
03432         py[4] += px0[4];
03433
03434         k ++;
03435         j = JA[k];
03436         px0 = x+j*5; // &x[j*nb];
03437         py = py0;
03438         py[0] += px0[0];
03439         py[1] += px0[1];
03440         py[2] += px0[2];
03441         py[3] += px0[3];
03442         py[4] += px0[4];
03443
03444         break;
03445
03446     case 4:
03447         k = IA[i];
03448         j = JA[k];
03449         px0 = x+j*5; // &x[j*nb];
03450         py = py0;
03451         py[0] += px0[0];
03452         py[1] += px0[1];
03453         py[2] += px0[2];
03454         py[3] += px0[3];
03455         py[4] += px0[4];
03456
03457         k ++;
03458         j = JA[k];
03459         px0 = x+j*5; // &x[j*nb];
03460         py = py0;
03461         py[0] += px0[0];
03462         py[1] += px0[1];
03463         py[2] += px0[2];
03464         py[3] += px0[3];
03465         py[4] += px0[4];
03466
03467         k ++;
03468         j = JA[k];
03469         px0 = x+j*5; // &x[j*nb];
03470         py = py0;
03471         py[0] += px0[0];
03472         py[1] += px0[1];
03473         py[2] += px0[2];
03474         py[3] += px0[3];
03475         py[4] += px0[4];
03476
03477         k ++;
03478         j = JA[k];
03479         px0 = x+j*5; // &x[j*nb];
03480         py = py0;
03481         py[0] += px0[0];
03482         py[1] += px0[1];
03483         py[2] += px0[2];
03484         py[3] += px0[3];
03485         py[4] += px0[4];
03486
03487         break;
03488
03489     case 5:
03490         k = IA[i];
03491         j = JA[k];
03492         px0 = x+j*5; // &x[j*nb];
03493         py = py0;
03494         py[0] += px0[0];
03495         py[1] += px0[1];
03496         py[2] += px0[2];

```

```

03497         py[3] += px0[3];
03498         py[4] += px0[4];
03499
03500         k++;
03501         j = JA[k];
03502         px0 = x+j*5; // &x[j*nb];
03503         py = py0;
03504         py[0] += px0[0];
03505         py[1] += px0[1];
03506         py[2] += px0[2];
03507         py[3] += px0[3];
03508         py[4] += px0[4];
03509
03510         k++;
03511         j = JA[k];
03512         px0 = x+j*5; // &x[j*nb];
03513         py = py0;
03514         py[0] += px0[0];
03515         py[1] += px0[1];
03516         py[2] += px0[2];
03517         py[3] += px0[3];
03518         py[4] += px0[4];
03519
03520         k++;
03521         j = JA[k];
03522         px0 = x+j*5; // &x[j*nb];
03523         py = py0;
03524         py[0] += px0[0];
03525         py[1] += px0[1];
03526         py[2] += px0[2];
03527         py[3] += px0[3];
03528         py[4] += px0[4];
03529
03530         k++;
03531         j = JA[k];
03532         px0 = x+j*5; // &x[j*nb];
03533         py = py0;
03534         py[0] += px0[0];
03535         py[1] += px0[1];
03536         py[2] += px0[2];
03537         py[3] += px0[3];
03538         py[4] += px0[4];
03539
03540         break;
03541
03542     case 6:
03543         k = IA[i];
03544         j = JA[k];
03545         px0 = x+j*5; // &x[j*nb];
03546         py = py0;
03547         py[0] += px0[0];
03548         py[1] += px0[1];
03549         py[2] += px0[2];
03550         py[3] += px0[3];
03551         py[4] += px0[4];
03552
03553         k++;
03554         j = JA[k];
03555         px0 = x+j*5; // &x[j*nb];
03556         py = py0;
03557         py[0] += px0[0];
03558         py[1] += px0[1];
03559         py[2] += px0[2];
03560         py[3] += px0[3];
03561         py[4] += px0[4];
03562
03563         k++;
03564         j = JA[k];
03565         px0 = x+j*5; // &x[j*nb];
03566         py = py0;
03567         py[0] += px0[0];
03568         py[1] += px0[1];
03569         py[2] += px0[2];
03570         py[3] += px0[3];
03571         py[4] += px0[4];
03572
03573         k++;
03574         j = JA[k];
03575         px0 = x+j*5; // &x[j*nb];
03576         py = py0;
03577         py[0] += px0[0];

```

```
03578         py[1] += px0[1];
03579         py[2] += px0[2];
03580         py[3] += px0[3];
03581         py[4] += px0[4];
03582
03583         k ++;
03584         j = JA[k];
03585         px0 = x+j*5; // &x[j*nb];
03586         py = py0;
03587         py[0] += px0[0];
03588         py[1] += px0[1];
03589         py[2] += px0[2];
03590         py[3] += px0[3];
03591         py[4] += px0[4];
03592
03593         k ++;
03594         j = JA[k];
03595         px0 = x+j*5; // &x[j*nb];
03596         py = py0;
03597         py[0] += px0[0];
03598         py[1] += px0[1];
03599         py[2] += px0[2];
03600         py[3] += px0[3];
03601         py[4] += px0[4];
03602
03603         break;
03604
03605     case 7:
03606         k = IA[i];
03607         j = JA[k];
03608         px0 = x+j*5; // &x[j*nb];
03609         py = py0;
03610         py[0] += px0[0];
03611         py[1] += px0[1];
03612         py[2] += px0[2];
03613         py[3] += px0[3];
03614         py[4] += px0[4];
03615
03616         k ++;
03617         j = JA[k];
03618         px0 = x+j*5; // &x[j*nb];
03619         py = py0;
03620         py[0] += px0[0];
03621         py[1] += px0[1];
03622         py[2] += px0[2];
03623         py[3] += px0[3];
03624         py[4] += px0[4];
03625
03626         k ++;
03627         j = JA[k];
03628         px0 = x+j*5; // &x[j*nb];
03629         py = py0;
03630         py[0] += px0[0];
03631         py[1] += px0[1];
03632         py[2] += px0[2];
03633         py[3] += px0[3];
03634         py[4] += px0[4];
03635
03636         k ++;
03637         j = JA[k];
03638         px0 = x+j*5; // &x[j*nb];
03639         py = py0;
03640         py[0] += px0[0];
03641         py[1] += px0[1];
03642         py[2] += px0[2];
03643         py[3] += px0[3];
03644         py[4] += px0[4];
03645
03646         k ++;
03647         j = JA[k];
03648         px0 = x+j*5; // &x[j*nb];
03649         py = py0;
03650         py[0] += px0[0];
03651         py[1] += px0[1];
03652         py[2] += px0[2];
03653         py[3] += px0[3];
03654         py[4] += px0[4];
03655
03656         k ++;
03657         j = JA[k];
03658         px0 = x+j*5; // &x[j*nb];
```



```

03659         py = py0;
03660         py[0] += px0[0];
03661         py[1] += px0[1];
03662         py[2] += px0[2];
03663         py[3] += px0[3];
03664         py[4] += px0[4];
03665
03666         k++;
03667         j = JA[k];
03668         px0 = x+j*5; // &x[j*nb];
03669         py = py0;
03670         py[0] += px0[0];
03671         py[1] += px0[1];
03672         py[2] += px0[2];
03673         py[3] += px0[3];
03674         py[4] += px0[4];
03675
03676         break;
03677
03678     default:
03679         for (k = IA[i]; k < IA[i+1]; ++k) {
03680             j = JA[k];
03681             px0 = x+j*5; // &x[j*nb];
03682             py = py0;
03683             py[0] += px0[0];
03684             py[1] += px0[1];
03685             py[2] += px0[2];
03686             py[3] += px0[3];
03687             py[4] += px0[4];
03688         }
03689         break;
03690     }
03691 }
03692 }
03693 }
03694 break;
03695
03696 case 7:
03697 {
03698     if (use_omp) {
03699 #ifdef _OPENMP
03700 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
03701 {
03702     myid = omp_get_thread_num();
03703     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
03704     for (i=mybegin; i < myend; ++i) {
03705         py0 = &y[i*7];
03706         num_nnz_row = IA[i+1] - IA[i];
03707         switch(num_nnz_row) {
03708
03709             case 3:
03710                 k = IA[i];
03711                 j = JA[k];
03712                 pA = val+k*49; // &val[k*jump];
03713                 px0 = x+j*7; // &x[j*nb];
03714                 py = py0;
03715                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
03716
03717                 k++;
03718                 j = JA[k];
03719                 pA = val+k*49; // &val[k*jump];
03720                 px0 = x+j*7; // &x[j*nb];
03721                 py = py0;
03722                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
03723
03724                 k++;
03725                 j = JA[k];
03726                 pA = val+k*49; // &val[k*jump];
03727                 px0 = x+j*7; // &x[j*nb];
03728                 py = py0;
03729                 fasp_blas_smat_ypAx_nc7( pA, px0, py );
03730
03731                 break;
03732
03733             case 4:
03734                 k = IA[i];
03735                 j = JA[k];
03736                 pA = val+k*49; // &val[k*jump];
03737                 px0 = x+j*7; // &x[j*nb];
03738                 py = py0;
03739                 fasp_blas_smat_ypAx_nc7( pA, px0, py );

```

```

03740
03741         k ++;
03742         j = JA[k];
03743         pA = val+k*49; // &val[k*jump];
03744         px0 = x+j*7; // &x[j*nb];
03745         py = py0;
03746         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03747
03748         k ++;
03749         j = JA[k];
03750         pA = val+k*49; // &val[k*jump];
03751         px0 = x+j*7; // &x[j*nb];
03752         py = py0;
03753         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03754
03755         k ++;
03756         j = JA[k];
03757         pA = val+k*49; // &val[k*jump];
03758         px0 = x+j*7; // &x[j*nb];
03759         py = py0;
03760         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03761
03762         break;
03763
03764     case 5:
03765         k = IA[i];
03766         j = JA[k];
03767         pA = val+k*49; // &val[k*jump];
03768         px0 = x+j*7; // &x[j*nb];
03769         py = py0;
03770         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03771
03772         k ++;
03773         j = JA[k];
03774         pA = val+k*49; // &val[k*jump];
03775         px0 = x+j*7; // &x[j*nb];
03776         py = py0;
03777         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03778
03779         k ++;
03780         j = JA[k];
03781         pA = val+k*49; // &val[k*jump];
03782         px0 = x+j*7; // &x[j*nb];
03783         py = py0;
03784         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03785
03786         k ++;
03787         j = JA[k];
03788         pA = val+k*49; // &val[k*jump];
03789         px0 = x+j*7; // &x[j*nb];
03790         py = py0;
03791         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03792
03793         k ++;
03794         j = JA[k];
03795         pA = val+k*49; // &val[k*jump];
03796         px0 = x+j*7; // &x[j*nb];
03797         py = py0;
03798         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03799
03800         break;
03801
03802     case 6:
03803         k = IA[i];
03804         j = JA[k];
03805         pA = val+k*49; // &val[k*jump];
03806         px0 = x+j*7; // &x[j*nb];
03807         py = py0;
03808         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03809
03810         k ++;
03811         j = JA[k];
03812         pA = val+k*49; // &val[k*jump];
03813         px0 = x+j*7; // &x[j*nb];
03814         py = py0;
03815         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03816
03817         k ++;
03818         j = JA[k];
03819         pA = val+k*49; // &val[k*jump];
03820         px0 = x+j*7; // &x[j*nb];

```

```

03821         py = py0;
03822         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03823
03824         k ++;
03825         j = JA[k];
03826         pA = val+k*49; // &val[k*jump];
03827         px0 = x+j*7; // &x[j*nb];
03828         py = py0;
03829         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03830
03831         k ++;
03832         j = JA[k];
03833         pA = val+k*49; // &val[k*jump];
03834         px0 = x+j*7; // &x[j*nb];
03835         py = py0;
03836         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03837
03838         k ++;
03839         j = JA[k];
03840         pA = val+k*49; // &val[k*jump];
03841         px0 = x+j*7; // &x[j*nb];
03842         py = py0;
03843         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03844
03845         break;
03846
03847     case 7:
03848         k = IA[i];
03849         j = JA[k];
03850         pA = val+k*49; // &val[k*jump];
03851         px0 = x+j*7; // &x[j*nb];
03852         py = py0;
03853         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03854
03855         k ++;
03856         j = JA[k];
03857         pA = val+k*49; // &val[k*jump];
03858         px0 = x+j*7; // &x[j*nb];
03859         py = py0;
03860         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03861
03862         k ++;
03863         j = JA[k];
03864         pA = val+k*49; // &val[k*jump];
03865         px0 = x+j*7; // &x[j*nb];
03866         py = py0;
03867         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03868
03869         k ++;
03870         j = JA[k];
03871         pA = val+k*49; // &val[k*jump];
03872         px0 = x+j*7; // &x[j*nb];
03873         py = py0;
03874         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03875
03876         k ++;
03877         j = JA[k];
03878         pA = val+k*49; // &val[k*jump];
03879         px0 = x+j*7; // &x[j*nb];
03880         py = py0;
03881         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03882
03883         k ++;
03884         j = JA[k];
03885         pA = val+k*49; // &val[k*jump];
03886         px0 = x+j*7; // &x[j*nb];
03887         py = py0;
03888         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03889
03890         k ++;
03891         j = JA[k];
03892         pA = val+k*49; // &val[k*jump];
03893         px0 = x+j*7; // &x[j*nb];
03894         py = py0;
03895         fasp_blas_smat_ypAx_nc7( pA, px0, py );
03896
03897         break;
03898
03899     default:
03900         for (k = IA[i]; k < IA[i+1]; ++k) {
03901             j = JA[k];

```

```

03902             pA = val+k*49; // &val[k*jump];
03903             px0 = x+j*7; // &x[j*nb];
03904             py = py0;
03905             fasp_blas_smat_ypAx_nc7( pA, px0, py );
03906         }
03907         break;
03908     }
03909 }
03910 }
03911 #endif
03912 }
03913 else {
03914     for (i = 0; i < ROW; ++i) {
03915         py0 = &y[i*7];
03916         num_nnz_row = IA[i+1] - IA[i];
03917         switch(num_nnz_row) {
03918
03919             case 3:
03920                 k = IA[i];
03921                 j = JA[k];
03922                 px0 = x+j*7; // &x[j*nb];
03923                 py = py0;
03924                 py[0] += px0[0];
03925                 py[1] += px0[1];
03926                 py[2] += px0[2];
03927                 py[3] += px0[3];
03928                 py[4] += px0[4];
03929                 py[5] += px0[5];
03930                 py[6] += px0[6];
03931
03932                 k ++;
03933                 j = JA[k];
03934                 px0 = x+j*7; // &x[j*nb];
03935                 py = py0;
03936                 py[0] += px0[0];
03937                 py[1] += px0[1];
03938                 py[2] += px0[2];
03939                 py[3] += px0[3];
03940                 py[4] += px0[4];
03941                 py[5] += px0[5];
03942                 py[6] += px0[6];
03943
03944                 k ++;
03945                 j = JA[k];
03946                 px0 = x+j*7; // &x[j*nb];
03947                 py = py0;
03948                 py[0] += px0[0];
03949                 py[1] += px0[1];
03950                 py[2] += px0[2];
03951                 py[3] += px0[3];
03952                 py[4] += px0[4];
03953                 py[5] += px0[5];
03954                 py[6] += px0[6];
03955
03956                 break;
03957
03958             case 4:
03959                 k = IA[i];
03960                 j = JA[k];
03961                 px0 = x+j*7; // &x[j*nb];
03962                 py = py0;
03963                 py[0] += px0[0];
03964                 py[1] += px0[1];
03965                 py[2] += px0[2];
03966                 py[3] += px0[3];
03967                 py[4] += px0[4];
03968                 py[5] += px0[5];
03969                 py[6] += px0[6];
03970
03971                 k ++;
03972                 j = JA[k];
03973                 px0 = x+j*7; // &x[j*nb];
03974                 py = py0;
03975                 py[0] += px0[0];
03976                 py[1] += px0[1];
03977                 py[2] += px0[2];
03978                 py[3] += px0[3];
03979                 py[4] += px0[4];
03980                 py[5] += px0[5];
03981                 py[6] += px0[6];
03982

```

```

03983         k ++;
03984         j = JA[k];
03985         px0 = x+j*7; // &x[j*nb];
03986         py = py0;
03987         py[0] += px0[0];
03988         py[1] += px0[1];
03989         py[2] += px0[2];
03990         py[3] += px0[3];
03991         py[4] += px0[4];
03992         py[5] += px0[5];
03993         py[6] += px0[6];
03994
03995         k ++;
03996         j = JA[k];
03997         px0 = x+j*7; // &x[j*nb];
03998         py = py0;
03999         py[0] += px0[0];
04000         py[1] += px0[1];
04001         py[2] += px0[2];
04002         py[3] += px0[3];
04003         py[4] += px0[4];
04004         py[5] += px0[5];
04005         py[6] += px0[6];
04006
04007         break;
04008
04009     case 5:
04010         k = IA[i];
04011         j = JA[k];
04012         px0 = x+j*7; // &x[j*nb];
04013         py = py0;
04014         py[0] += px0[0];
04015         py[1] += px0[1];
04016         py[2] += px0[2];
04017         py[3] += px0[3];
04018         py[4] += px0[4];
04019         py[5] += px0[5];
04020         py[6] += px0[6];
04021
04022         k ++;
04023         j = JA[k];
04024         px0 = x+j*7; // &x[j*nb];
04025         py = py0;
04026         py[0] += px0[0];
04027         py[1] += px0[1];
04028         py[2] += px0[2];
04029         py[3] += px0[3];
04030         py[4] += px0[4];
04031         py[5] += px0[5];
04032         py[6] += px0[6];
04033
04034         k ++;
04035         j = JA[k];
04036         px0 = x+j*7; // &x[j*nb];
04037         py = py0;
04038         py[0] += px0[0];
04039         py[1] += px0[1];
04040         py[2] += px0[2];
04041         py[3] += px0[3];
04042         py[4] += px0[4];
04043         py[5] += px0[5];
04044         py[6] += px0[6];
04045
04046         k ++;
04047         j = JA[k];
04048         px0 = x+j*7; // &x[j*nb];
04049         py = py0;
04050         py[0] += px0[0];
04051         py[1] += px0[1];
04052         py[2] += px0[2];
04053         py[3] += px0[3];
04054         py[4] += px0[4];
04055         py[5] += px0[5];
04056         py[6] += px0[6];
04057
04058         k ++;
04059         j = JA[k];
04060         px0 = x+j*7; // &x[j*nb];
04061         py = py0;
04062         py[0] += px0[0];
04063         py[1] += px0[1];

```

```
04064         py[2] += px0[2];
04065         py[3] += px0[3];
04066         py[4] += px0[4];
04067         py[5] += px0[5];
04068         py[6] += px0[6];
04069
04070         break;
04071
04072     case 6:
04073         k = IA[i];
04074         j = JA[k];
04075         px0 = x+j*7; // &x[j*nb];
04076         py = py0;
04077         py[0] += px0[0];
04078         py[1] += px0[1];
04079         py[2] += px0[2];
04080         py[3] += px0[3];
04081         py[4] += px0[4];
04082         py[5] += px0[5];
04083         py[6] += px0[6];
04084
04085         k ++;
04086         j = JA[k];
04087         px0 = x+j*7; // &x[j*nb];
04088         py = py0;
04089         py[0] += px0[0];
04090         py[1] += px0[1];
04091         py[2] += px0[2];
04092         py[3] += px0[3];
04093         py[4] += px0[4];
04094         py[5] += px0[5];
04095         py[6] += px0[6];
04096
04097         k ++;
04098         j = JA[k];
04099         px0 = x+j*7; // &x[j*nb];
04100         py = py0;
04101         py[0] += px0[0];
04102         py[1] += px0[1];
04103         py[2] += px0[2];
04104         py[3] += px0[3];
04105         py[4] += px0[4];
04106         py[5] += px0[5];
04107         py[6] += px0[6];
04108
04109         k ++;
04110         j = JA[k];
04111         px0 = x+j*7; // &x[j*nb];
04112         py = py0;
04113         py[0] += px0[0];
04114         py[1] += px0[1];
04115         py[2] += px0[2];
04116         py[3] += px0[3];
04117         py[4] += px0[4];
04118         py[5] += px0[5];
04119         py[6] += px0[6];
04120
04121         k ++;
04122         j = JA[k];
04123         px0 = x+j*7; // &x[j*nb];
04124         py = py0;
04125         py[0] += px0[0];
04126         py[1] += px0[1];
04127         py[2] += px0[2];
04128         py[3] += px0[3];
04129         py[4] += px0[4];
04130         py[5] += px0[5];
04131         py[6] += px0[6];
04132
04133         k ++;
04134         j = JA[k];
04135         px0 = x+j*7; // &x[j*nb];
04136         py = py0;
04137         py[0] += px0[0];
04138         py[1] += px0[1];
04139         py[2] += px0[2];
04140         py[3] += px0[3];
04141         py[4] += px0[4];
04142         py[5] += px0[5];
04143         py[6] += px0[6];
04144
```

```
04145         break;
04146
04147     case 7:
04148         k = IA[i];
04149         j = JA[k];
04150         px0 = x+j*7; // &x[j*nb];
04151         py = py0;
04152         py[0] += px0[0];
04153         py[1] += px0[1];
04154         py[2] += px0[2];
04155         py[3] += px0[3];
04156         py[4] += px0[4];
04157         py[5] += px0[5];
04158         py[6] += px0[6];
04159
04160         k ++;
04161         j = JA[k];
04162         px0 = x+j*7; // &x[j*nb];
04163         py = py0;
04164         py[0] += px0[0];
04165         py[1] += px0[1];
04166         py[2] += px0[2];
04167         py[3] += px0[3];
04168         py[4] += px0[4];
04169         py[5] += px0[5];
04170         py[6] += px0[6];
04171
04172         k ++;
04173         j = JA[k];
04174         px0 = x+j*7; // &x[j*nb];
04175         py = py0;
04176         py[0] += px0[0];
04177         py[1] += px0[1];
04178         py[2] += px0[2];
04179         py[3] += px0[3];
04180         py[4] += px0[4];
04181         py[5] += px0[5];
04182         py[6] += px0[6];
04183
04184         k ++;
04185         j = JA[k];
04186         px0 = x+j*7; // &x[j*nb];
04187         py = py0;
04188         py[0] += px0[0];
04189         py[1] += px0[1];
04190         py[2] += px0[2];
04191         py[3] += px0[3];
04192         py[4] += px0[4];
04193         py[5] += px0[5];
04194         py[6] += px0[6];
04195
04196         k ++;
04197         j = JA[k];
04198         px0 = x+j*7; // &x[j*nb];
04199         py = py0;
04200         py[0] += px0[0];
04201         py[1] += px0[1];
04202         py[2] += px0[2];
04203         py[3] += px0[3];
04204         py[4] += px0[4];
04205         py[5] += px0[5];
04206         py[6] += px0[6];
04207
04208         k ++;
04209         j = JA[k];
04210         px0 = x+j*7; // &x[j*nb];
04211         py = py0;
04212         py[0] += px0[0];
04213         py[1] += px0[1];
04214         py[2] += px0[2];
04215         py[3] += px0[3];
04216         py[4] += px0[4];
04217         py[5] += px0[5];
04218         py[6] += px0[6];
04219
04220         k ++;
04221         j = JA[k];
04222         px0 = x+j*7; // &x[j*nb];
04223         py = py0;
04224         py[0] += px0[0];
04225         py[1] += px0[1];
```

```

04226         py[2] += px0[2];
04227         py[3] += px0[3];
04228         py[4] += px0[4];
04229         py[5] += px0[5];
04230         py[6] += px0[6];
04231
04232         break;
04233
04234     default:
04235         for (k = IA[i]; k < IA[i+1]; ++k) {
04236             j = JA[k];
04237             px0 = x+j*7; // &x[j*nb];
04238             py = py0;
04239             py[0] += px0[0];
04240             py[1] += px0[1];
04241             py[2] += px0[2];
04242             py[3] += px0[3];
04243             py[4] += px0[4];
04244             py[5] += px0[5];
04245             py[6] += px0[6];
04246         }
04247         break;
04248     }
04249 }
04250 }
04251 }
04252 break;
04253
04254 default:
04255 {
04256     if (use_omp) {
04257 #ifdef _OPENMP
04258 #pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
04259 {
04260     myid = omp_get_thread_num();
04261     fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
04262     for (i=mybegin; i < myend; ++i) {
04263         py0 = &y[i*nb];
04264         num_nnz_row = IA[i+1] - IA[i];
04265         switch(num_nnz_row) {
04266
04267             case 3:
04268                 k = IA[i];
04269                 j = JA[k];
04270                 px0 = x+j*nb; // &x[j*nb];
04271                 py = py0;
04272                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04273
04274                 k ++;
04275                 j = JA[k];
04276                 px0 = x+j*nb; // &x[j*nb];
04277                 py = py0;
04278                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04279
04280                 k ++;
04281                 j = JA[k];
04282                 px0 = x+j*nb; // &x[j*nb];
04283                 py = py0;
04284                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04285
04286                 break;
04287
04288             case 4:
04289                 k = IA[i];
04290                 j = JA[k];
04291                 px0 = x+j*nb; // &x[j*nb];
04292                 py = py0;
04293                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04294
04295                 k ++;
04296                 j = JA[k];
04297                 px0 = x+j*nb; // &x[j*nb];
04298                 py = py0;
04299                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04300
04301                 k ++;
04302                 j = JA[k];
04303                 px0 = x+j*nb; // &x[j*nb];
04304                 py = py0;
04305                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04306

```



```

04307         k++;
04308         j = JA[k];
04309         px0 = x+j*nb; // &x[j*nb];
04310         py = py0;
04311         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04312
04313         break;
04314
04315     case 5:
04316         k = IA[i];
04317         j = JA[k];
04318         px0 = x+j*nb; // &x[j*nb];
04319         py = py0;
04320         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04321
04322         k++;
04323         j = JA[k];
04324         px0 = x+j*nb; // &x[j*nb];
04325         py = py0;
04326         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04327
04328         k++;
04329         j = JA[k];
04330         px0 = x+j*nb; // &x[j*nb];
04331         py = py0;
04332         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04333
04334         k++;
04335         j = JA[k];
04336         px0 = x+j*nb; // &x[j*nb];
04337         py = py0;
04338         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04339
04340         k++;
04341         j = JA[k];
04342         px0 = x+j*nb; // &x[j*nb];
04343         py = py0;
04344         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04345
04346         break;
04347
04348     case 6:
04349         k = IA[i];
04350         j = JA[k];
04351         px0 = x+j*nb; // &x[j*nb];
04352         py = py0;
04353         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04354
04355         k++;
04356         j = JA[k];
04357         px0 = x+j*nb; // &x[j*nb];
04358         py = py0;
04359         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04360
04361         k++;
04362         j = JA[k];
04363         px0 = x+j*nb; // &x[j*nb];
04364         py = py0;
04365         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04366
04367         k++;
04368         j = JA[k];
04369         px0 = x+j*nb; // &x[j*nb];
04370         py = py0;
04371         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04372
04373         k++;
04374         j = JA[k];
04375         px0 = x+j*nb; // &x[j*nb];
04376         py = py0;
04377         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04378
04379         k++;
04380         j = JA[k];
04381         px0 = x+j*nb; // &x[j*nb];
04382         py = py0;
04383         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04384
04385         break;
04386
04387     case 7:

```

```

04388         k = IA[i];
04389         j = JA[k];
04390         px0 = x+j*nb; // &x[j*nb];
04391         py = py0;
04392         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04393
04394         k ++;
04395         j = JA[k];
04396         px0 = x+j*nb; // &x[j*nb];
04397         py = py0;
04398         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04399
04400         k ++;
04401         j = JA[k];
04402         px0 = x+j*nb; // &x[j*nb];
04403         py = py0;
04404         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04405
04406         k ++;
04407         j = JA[k];
04408         px0 = x+j*nb; // &x[j*nb];
04409         py = py0;
04410         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04411
04412         k ++;
04413         j = JA[k];
04414         px0 = x+j*nb; // &x[j*nb];
04415         py = py0;
04416         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04417
04418         k ++;
04419         j = JA[k];
04420         px0 = x+j*nb; // &x[j*nb];
04421         py = py0;
04422         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04423
04424         k ++;
04425         j = JA[k];
04426         px0 = x+j*nb; // &x[j*nb];
04427         py = py0;
04428         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04429
04430         break;
04431
04432     default:
04433         for (k = IA[i]; k < IA[i+1]; ++k) {
04434             j = JA[k];
04435             px0 = x+j*nb; // &x[j*nb];
04436             py = py0;
04437             fasp_blas_darray_axpy (nb, 1.0, px0, py);
04438         }
04439         break;
04440     }
04441 }
04442 }
04443 #endif
04444 }
04445 else {
04446     for (i = 0; i < ROW; ++i) {
04447         py0 = &y[i*nb];
04448         num_nnz_row = IA[i+1] - IA[i];
04449         switch(num_nnz_row) {
04450
04451             case 3:
04452                 k = IA[i];
04453                 j = JA[k];
04454                 px0 = x+j*nb; // &x[j*nb];
04455                 py = py0;
04456                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04457
04458                 k ++;
04459                 j = JA[k];
04460                 px0 = x+j*nb; // &x[j*nb];
04461                 py = py0;
04462                 fasp_blas_darray_axpy (nb, 1.0, px0, py);
04463
04464                 k ++;
04465                 j = JA[k];
04466                 px0 = x+j*nb; // &x[j*nb];
04467                 py = py0;
04468                 fasp_blas_darray_axpy (nb, 1.0, px0, py);

```

```
04469
04470         break;
04471
04472     case 4:
04473         k = IA[i];
04474         j = JA[k];
04475         px0 = x+j*nb; // &x[j*nb];
04476         py = py0;
04477         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04478
04479         k ++;
04480         j = JA[k];
04481         px0 = x+j*nb; // &x[j*nb];
04482         py = py0;
04483         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04484
04485         k ++;
04486         j = JA[k];
04487         px0 = x+j*nb; // &x[j*nb];
04488         py = py0;
04489         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04490
04491         k ++;
04492         j = JA[k];
04493         px0 = x+j*nb; // &x[j*nb];
04494         py = py0;
04495         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04496
04497         break;
04498
04499     case 5:
04500         k = IA[i];
04501         j = JA[k];
04502         px0 = x+j*nb; // &x[j*nb];
04503         py = py0;
04504         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04505
04506         k ++;
04507         j = JA[k];
04508         px0 = x+j*nb; // &x[j*nb];
04509         py = py0;
04510         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04511
04512         k ++;
04513         j = JA[k];
04514         px0 = x+j*nb; // &x[j*nb];
04515         py = py0;
04516         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04517
04518         k ++;
04519         j = JA[k];
04520         px0 = x+j*nb; // &x[j*nb];
04521         py = py0;
04522         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04523
04524         k ++;
04525         j = JA[k];
04526         px0 = x+j*nb; // &x[j*nb];
04527         py = py0;
04528         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04529
04530         break;
04531
04532     case 6:
04533         k = IA[i];
04534         j = JA[k];
04535         px0 = x+j*nb; // &x[j*nb];
04536         py = py0;
04537         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04538
04539         k ++;
04540         j = JA[k];
04541         px0 = x+j*nb; // &x[j*nb];
04542         py = py0;
04543         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04544
04545         k ++;
04546         j = JA[k];
04547         px0 = x+j*nb; // &x[j*nb];
04548         py = py0;
04549         fasp_blas_darray_axpy (nb, 1.0, px0, py);
```

```

04550
04551         k++;
04552         j = JA[k];
04553         px0 = x+j*nb; // &x[j*nb];
04554         py = py0;
04555         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04556
04557         k++;
04558         j = JA[k];
04559         px0 = x+j*nb; // &x[j*nb];
04560         py = py0;
04561         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04562
04563         k++;
04564         j = JA[k];
04565         px0 = x+j*nb; // &x[j*nb];
04566         py = py0;
04567         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04568
04569         break;
04570
04571     case 7:
04572         k = IA[i];
04573         j = JA[k];
04574         px0 = x+j*nb; // &x[j*nb];
04575         py = py0;
04576         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04577
04578         k++;
04579         j = JA[k];
04580         px0 = x+j*nb; // &x[j*nb];
04581         py = py0;
04582         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04583
04584         k++;
04585         j = JA[k];
04586         px0 = x+j*nb; // &x[j*nb];
04587         py = py0;
04588         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04589
04590         k++;
04591         j = JA[k];
04592         px0 = x+j*nb; // &x[j*nb];
04593         py = py0;
04594         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04595
04596         k++;
04597         j = JA[k];
04598         px0 = x+j*nb; // &x[j*nb];
04599         py = py0;
04600         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04601
04602         k++;
04603         j = JA[k];
04604         px0 = x+j*nb; // &x[j*nb];
04605         py = py0;
04606         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04607
04608         k++;
04609         j = JA[k];
04610         px0 = x+j*nb; // &x[j*nb];
04611         py = py0;
04612         fasp_blas_darray_axpy (nb, 1.0, px0, py);
04613
04614         break;
04615
04616     default:
04617         for (k = IA[i]; k < IA[i+1]; ++k) {
04618             j = JA[k];
04619             px0 = x+j*nb; // &x[j*nb];
04620             py = py0;
04621             fasp_blas_darray_axpy (nb, 1.0, px0, py);
04622         }
04623         break;
04624     }
04625 }
04626 }
04627 }
04628     break;
04629 }
04630 }

```

```

04631
04646 void fasp_blas_dbsr_mxm (const dBSRmat *A,
04647                          const dBSRmat *B,
04648                          dBSRmat *C)
04649 {
04650
04651     INT i,j,k,l,count;
04652     INT *JD = (INT *)fasp_mem_calloc(B->COL,sizeof(INT));
04653
04654     const INT nb = A->nb;
04655     const INT nb2 = nb*nb;
04656
04657     // check A and B see if there are compatible for multiplication
04658     if ( (A->COL != B->ROW) && (A->nb != B->nb) ) {
04659         printf("### ERROR: Matrix sizes do not match!\n");
04660         fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
04661     }
04662
04663     C->ROW = A->ROW;
04664     C->COL = B->COL;
04665     C->nb = A->nb;
04666     C->storage_manner = A->storage_manner;
04667
04668     C->val = NULL;
04669     C->JA = NULL;
04670     C->IA = (INT*)fasp_mem_calloc(C->ROW+1,sizeof(INT));
04671
04672     REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));
04673
04674     for (i=0;i<B->COL;++i) JD[i]=-1;
04675
04676     // step 1: Find first the structure IA of C
04677     for (i=0;i<C->ROW;++i) {
04678         count=0;
04679
04680         for (k=A->IA[i];k<A->IA[i+1];++k) {
04681             for (j=B->IA[A->JA[k]];j<B->IA[A->JA[k]+1];++j) {
04682                 for (l=0;l<count;l++) {
04683                     if (JD[l]==B->JA[j]) break;
04684                 }
04685
04686                 if (l==count) {
04687                     JD[count]=B->JA[j];
04688                     count++;
04689                 }
04690             }
04691         }
04692         C->IA[i+1]=count;
04693         for (j=0;j<count;++j) {
04694             JD[j]=-1;
04695         }
04696     }
04697
04698     for (i=0;i<C->ROW;++i) C->IA[i+1]+=C->IA[i];
04699
04700     // step 2: Find the structure JA of C
04701     INT countJD;
04702
04703     C->JA=(INT*)fasp_mem_calloc(C->IA[C->ROW],sizeof(INT));
04704
04705     for (i=0;i<C->ROW;++i) {
04706         countJD=0;
04707         count=C->IA[i];
04708         for (k=A->IA[i];k<A->IA[i+1];++k) {
04709             for (j=B->IA[A->JA[k]];j<B->IA[A->JA[k]+1];++j) {
04710                 for (l=0;l<countJD;l++) {
04711                     if (JD[l]==B->JA[j]) break;
04712                 }
04713
04714                 if (l==countJD) {
04715                     C->JA[countJD]=B->JA[j];
04716                     JD[countJD]=B->JA[j];
04717                     count++;
04718                     countJD++;
04719                 }
04720             }
04721         }
04722
04723         //for (j=0;j<countJD;++j) JD[j]=-1;
04724         fasp_iarray_set(countJD, JD, -1);
04725     }

```

```

04726
04727     fasp_mem_free(JD); JD = NULL;
04728
04729     // step 3: Find the structure A of C
04730     C->val=(REAL*) fasp_mem_malloc((C->IA[C->ROW])*nb2,sizeof(REAL));
04731
04732     for (i=0;i<C->ROW;++i) {
04733         for (j=C->IA[i];j<C->IA[i+1];++j) {
04734
04735             fasp_darray_set(nb2, C->val+(j*nb2), 0x0);
04736
04737             for (k=A->IA[i];k<A->IA[i+1];++k) {
04738                 for (l=B->IA[A->JA[k]];l<B->IA[A->JA[k]+1];l++) {
04739                     if (B->JA[l]==C->JA[j]) {
04740                         fasp_blas_smat_mul (A->val+(k*nb2), B->val+(l*nb2), temp, nb);
04741                         fasp_blas_darray_axpy (nb2, 1.0, temp, C->val+(j*nb2));
04742                     } // end if
04743                 } // end for l
04744             } // end for k
04745         } // end for j
04746     } // end for i
04747
04748     C->NNZ = C->IA[C->ROW]-C->IA[0];
04749
04750     fasp_mem_free(temp); temp = NULL;
04751 }
04752
04753 void fasp_blas_dbsr_rapl (const dBSRmat *R,
04754                         const dBSRmat *A,
04755                         const dBSRmat *P,
04756                         dBSRmat *B)
04757 {
04758     const INT row=R->ROW, col=P->COL, nb=A->nb, nb2=A->nb*A->nb;
04759
04760     const REAL *rj=R->val, *aj=A->val, *pj=P->val;
04761     const INT *ir=R->IA, *ia=A->IA, *ip=P->IA;
04762     const INT *jr=R->JA, *ja=A->JA, *jp=P->JA;
04763
04764     REAL *acj;
04765     INT *iac, *jac;
04766
04767     INT nB=A->NNZ;
04768     INT i,il,j,jj,k,length;
04769     INT begin_row,end_row,begin_rowA,end_rowA,begin_rowR,end_rowR;
04770     INT istart,iistart,count;
04771
04772     INT *index=(INT *)fasp_mem_malloc(A->COL,sizeof(INT));
04773
04774     REAL *smat_tmp=(REAL *)fasp_mem_malloc(nb2,sizeof(REAL));
04775
04776     INT *iindex=(INT *)fasp_mem_malloc(col,sizeof(INT));
04777
04778     for (i=0; i<A->COL; ++i) index[i] = -2;
04779
04780     memcpy(iindex,index,col*sizeof(INT));
04781
04782     jac=(INT*) fasp_mem_malloc(nB,sizeof(INT));
04783
04784     iac=(INT*) fasp_mem_malloc(row+1,sizeof(INT));
04785
04786     REAL *temp=(REAL*) fasp_mem_malloc(A->COL*nb2,sizeof(REAL));
04787
04788     iac[0] = 0;
04789
04790     // First loop: form sparsity pattern of R*A*P
04791     for (i=0; i < row; ++i) {
04792         // reset istart and length at the beginning of each loop
04793         istart = -1; length = 0; il = i+1;
04794
04795         // go across the rows in R
04796         begin_rowR=ir[i]; end_rowR=ir[il];
04797         for (jj=begin_rowR; jj<end_rowR; ++jj) {
04798             j = jr[jj];
04799             // for each column in A
04800             begin_rowA=ia[j]; end_rowA=ia[j+1];
04801             for (k=begin_rowA; k<end_rowA; ++k) {
04802                 if (index[ja[k]] == -2) {
04803                     index[ja[k]] = istart;
04804                     istart = ja[k];
04805                     ++length;
04806                 }
04807             }
04808         }
04809     }

```

```

04824     }
04825     }
04826 }
04827
04828 // book-keeping [resetting length and setting iistart]
04829 count = length; iistart = -1; length = 0;
04830
04831 // use each column that would have resulted from R*A
04832 for (j=0; j < count; ++j) {
04833     jj = iistart;
04834     iistart = index[iistart];
04835     index[jj] = -2;
04836
04837     // go across the row of P
04838     begin_row=ip[jj]; end_row=ip[jj+1];
04839     for (k=begin_row; k<end_row; ++k) {
04840         // pull out the appropriate columns of P
04841         if (iindex[jp[k]] == -2){
04842             iindex[jp[k]] = iistart;
04843             iistart = jp[k];
04844             ++length;
04845         }
04846     } // end for k
04847 } // end for j
04848
04849 // set B->IA
04850 iac[i1]=iac[i]+length;
04851
04852 if (iac[i1]>nB) {
04853     nB=nB*2;
04854     jac=(INT*)fasp_mem_realloc(jac, nB*sizeof(INT));
04855 }
04856
04857 // put the correct columns of p into the column list of the products
04858 begin_row=iac[i]; end_row=iac[i1];
04859 for (j=begin_row; j<end_row; ++j) {
04860     // put the value in B->JA
04861     jac[j] = iistart;
04862     // set iistart to the next value
04863     iistart = iindex[iistart];
04864     // set the iindex spot to 0
04865     iindex[jac[j]] = -2;
04866 } // end j
04867 } // end i: First loop
04868
04869 jac=(INT*)fasp_mem_realloc(jac, (iac[row])*sizeof(INT));
04870
04871 acj=(REAL*)fasp_mem_calloc(iac[row]*nb2,sizeof(REAL));
04872
04873 INT *BTindex=(INT*)fasp_mem_calloc(col,sizeof(INT));
04874
04875 // Second loop: compute entries of R*A*P
04876 for (i=0; i<row; ++i) {
04877     i1 = i+1;
04878     // each col of B
04879     begin_row=iac[i]; end_row=iac[i1];
04880     for (j=begin_row; j<end_row; ++j) {
04881         BTindex[jac[j]]=j;
04882     }
04883     // reset iistart and length at the beginning of each loop
04884     iistart = -1; length = 0;
04885
04886     // go across the rows in R
04887     begin_rowR=ir[i]; end_rowR=ir[i1];
04888     for ( jj=begin_rowR; jj<end_rowR; ++jj ) {
04889         j = jr[jj];
04890         // for each column in A
04891         begin_rowA=ia[j]; end_rowA=ia[j+1];
04892         for (k=begin_rowA; k<end_rowA; ++k) {
04893             if (index[ja[k]] == -2) {
04894                 index[ja[k]] = iistart;
04895                 iistart = ja[k];
04896                 ++length;
04897             }
04898             fasp_blas_smat_mul(&rj[jj*nb2],&aj[k*nb2],smat_tmp,nb);
04899             //fasp_darray_xpy(nb2,&temp[ja[k]*nb2], smat_tmp );
04900             fasp_blas_darray_axpy (nb2, 1.0, smat_tmp, &temp[ja[k]*nb2]);
04901
04902             //temp[ja[k]]+=rj[jj]*aj[k];
04903             // change to X = X+Y*Z
04904         }

```

```

04905     }
04906
04907     // book-keeping [resetting length and setting iistart]
04908     // use each column that would have resulted from R*A
04909     for (j=0; j<length; ++j) {
04910         jj = iistart;
04911         iistart = index[iistart];
04912         index[jj] = -2;
04913
04914         // go across the row of P
04915         begin_row=ip[jj]; end_row=ip[jj+1];
04916         for (k=begin_row; k<end_row; ++k) {
04917             // pull out the appropriate columns of P
04918             //acj[BTindex[jp[k]]]+=temp[jj]*pj[k];
04919             fasp_blas_smat_mul(&temp[jj*nb2], &pj[k*nb2], smat_tmp, nb);
04920             //fasp_darray_xpy(nb2, &acj[BTindex[jp[k]]*nb2], smat_tmp );
04921             fasp_blas_darray_axpy(nb2, 1.0, smat_tmp, &acj[BTindex[jp[k]]*nb2]);
04922
04923             // change to    X = X+Y*Z
04924         }
04925         //temp[jj]=0.0; // change to    X[nb,nb] = 0;
04926         fasp_darray_set(nb2, &temp[jj*nb2], 0.0);
04927     }
04928 } // end for i: Second loop
04929 // setup coarse matrix B
04930 B->ROW=row; B->COL=col;
04931 B->IA=iac; B->JA=jac; B->val=acj;
04932 B->NNZ=B->IA[B->ROW]-B->IA[0];
04933
04934 B->nb=A->nb;
04935 B->storage_manner = A->storage_manner;
04936
04937 fasp_mem_free(temp);      temp      = NULL;
04938 fasp_mem_free(index);    index     = NULL;
04939 fasp_mem_free(iindex);   iindex    = NULL;
04940 fasp_mem_free(BTindex);  BTindex   = NULL;
04941 fasp_mem_free(smat_tmp); smat_tmp  = NULL;
04942 }
04943
04961 void fasp_blas_dbsr_rap (const dBSRmat  *R,
04962                          const dBSRmat  *A,
04963                          const dBSRmat  *P,
04964                          dBSRmat        *B)
04965 {
04966     const INT row=R->ROW, col=P->COL, nb=A->nb, nb2=A->nb*A->nb;
04967
04968     const REAL *rj=R->val, *aj=A->val, *pj=P->val;
04969     const INT  *ir=R->IA,  *ia=A->IA,  *ip=P->IA;
04970     const INT  *jr=R->JA,  *ja=A->JA,  *jp=P->JA;
04971
04972     REAL      *acj;
04973     INT        *iac, *jac;
04974
04975     INT *Ps_marker = NULL;
04976     INT *As_marker = NULL;
04977
04978 #ifdef _OPENMP
04979     INT *P_marker = NULL;
04980     INT *A_marker = NULL;
04981     REAL *smat_tmp = NULL;
04982 #endif
04983
04984     INT i, i1, i2, i3, jj1, jj2, jj3;
04985     INT counter, jj_row_begining;
04986
04987     INT nthreads = 1;
04988
04989 #ifdef _OPENMP
04990     INT myid, mybegin, myend, Ctemp;
04991     nthreads = fasp_get_num_threads();
04992 #endif
04993
04994     INT n_coarse = row;
04995     INT n_fine   = A->ROW;
04996     INT coarse_mul_nthreads = n_coarse * nthreads;
04997     INT fine_mul_nthreads   = n_fine * nthreads;
04998     INT coarse_add_nthreads = n_coarse + nthreads;
04999     INT minus_one_length = coarse_mul_nthreads + fine_mul_nthreads;
05000     INT total_calloc = minus_one_length + coarse_add_nthreads + nthreads;
05001
05002     Ps_marker = (INT *)fasp_mem_calloc(total_calloc, sizeof(INT));

```



```

05003     As_marker = Ps_marker + coarse_mul_nthreads;
05004
05005     /*-----*
05006 * First Pass: Determine size of B and set up B_i *
05007 *-----*/
05008     iac = (INT *)fasp_mem_calloc(n_coarse+1, sizeof(INT));
05009
05010     fasp_iarray_set(minus_one_length, Ps_marker, -1);
05011
05012     REAL *tmp=(REAL *)fasp_mem_calloc(2*nthreads*nb2, sizeof(REAL));
05013
05014 #ifdef _OPENMP
05015     INT * RAP_temp = As_marker + fine_mul_nthreads;
05016     INT * part_end = RAP_temp + coarse_add_nthreads;
05017
05018     if (n_coarse > OPENMP_HOLDS) {
05019 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
05020 counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
05021         for (myid = 0; myid < nthreads; myid++) {
05022             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
05023             P_marker = Ps_marker + myid * n_coarse;
05024             A_marker = As_marker + myid * n_fine;
05025             counter = 0;
05026             for (i = mybegin; i < myend; ++i) {
05027                 P_marker[i] = counter;
05028                 jj_row_begining = counter;
05029                 counter ++;
05030                 for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05031                     i1 = jr[jj1];
05032                     for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05033                         i2 = ja[jj2];
05034                         if (A_marker[i2] != i) {
05035                             A_marker[i2] = i;
05036                             for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05037                                 i3 = jp[jj3];
05038                                 if (P_marker[i3] < jj_row_begining) {
05039                                     P_marker[i3] = counter;
05040                                     counter ++;
05041                                 }
05042                             }
05043                         }
05044                     }
05045                 }
05046                 RAP_temp[i+myid] = jj_row_begining;
05047             }
05048             RAP_temp[myend+myid] = counter;
05049             part_end[myid] = myend + myid + 1;
05050         }
05051         fasp_iarray_cp(part_end[0], RAP_temp, iac);
05052         counter = part_end[0];
05053         Ctemp = 0;
05054         for (i1 = 1; i1 < nthreads; i1++) {
05055             Ctemp += RAP_temp[part_end[i1]-1];
05056             for (jj1 = part_end[i1-1]+1; jj1 < part_end[i1]; jj1++) {
05057                 iac[counter] = RAP_temp[jj1] + Ctemp;
05058                 counter ++;
05059             }
05060         }
05061     }
05062     else {
05063 #endif
05064         counter = 0;
05065         for (i = 0; i < row; ++ i) {
05066             Ps_marker[i] = counter;
05067             jj_row_begining = counter;
05068             counter ++;
05069
05070             for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05071                 i1 = jr[jj1];
05072                 for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05073                     i2 = ja[jj2];
05074                     if (As_marker[i2] != i) {
05075                         As_marker[i2] = i;
05076                         for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05077                             i3 = jp[jj3];
05078                             if (Ps_marker[i3] < jj_row_begining) {
05079                                 Ps_marker[i3] = counter;
05080                                 counter ++;
05081                             }
05082                         }
05083                     }

```

```

05084         }
05085     }
05086     iac[i] = jj_row_begining;
05087 }
05088 #ifdef _OPENMP
05089 }
05090 #endif
05091
05092 iac[row] = counter;
05093
05094 jac=(INT*)fasp_mem_calloc(iac[row], sizeof(INT));
05095
05096 acj=(REAL*)fasp_mem_calloc(iac[row]*nb2, sizeof(REAL));
05097
05098 fasp_iarray_set(minus_one_length, Ps_marker, -1);
05099
05100 /*-----*
05101 * Second Pass: compute entries of B=R*A*P *
05102 *-----*/
05103 #ifdef _OPENMP
05104     if (n_coarse > OPENMP_HOLDS) {
05105 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
05106 counter, i, jj_row_begining, jj1, i1, jj2, i2, \
05107 jj3, i3, smat_tmp)
05108         for (myid = 0; myid < nthreads; myid++) {
05109             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
05110             P_marker = Ps_marker + myid * n_coarse;
05111             A_marker = As_marker + myid * n_fine;
05112             smat_tmp = tmp + myid*2*nb2;
05113             counter = iac[mybegin];
05114             for (i = mybegin; i < myend; ++i) {
05115                 P_marker[i] = counter;
05116                 jj_row_begining = counter;
05117                 jac[counter] = i;
05118                 fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
05119                 counter ++;
05120
05121                 for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05122                     i1 = jr[jj1];
05123                     for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05124                         fasp_blas_smat_mul(&r[jj1*nb2], &a[jj2*nb2], smat_tmp, nb);
05125                         i2 = ja[jj2];
05126                         if (A_marker[i2] != i) {
05127                             A_marker[i2] = i;
05128                             for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05129                                 i3 = jp[jj3];
05130                                 fasp_blas_smat_mul(smat_tmp, &p[jj3*nb2], smat_tmp+nb2, nb);
05131                                 if (P_marker[i3] < jj_row_begining) {
05132                                     P_marker[i3] = counter;
05133                                     fasp_darray_cp(nb2, smat_tmp+nb2, &acj[counter*nb2]);
05134                                     jac[counter] = i3;
05135                                     counter ++;
05136                                 }
05137                                 else {
05138                                     fasp_blas_darray_axpy(nb2, 1.0, smat_tmp+nb2,
05139                                                             &acj[P_marker[i3]*nb2]);
05140                                 }
05141                             }
05142                         }
05143                     }
05144                     else {
05145                         for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++ ) {
05146                             i3 = jp[jj3];
05147                             fasp_blas_smat_mul(smat_tmp, &p[jj3*nb2], smat_tmp+nb2, nb);
05148                             fasp_blas_darray_axpy(nb2, 1.0, smat_tmp+nb2,
05149                                                         &acj[P_marker[i3]*nb2]);
05150                         }
05151                     }
05152                 }
05153             }
05154         }
05155     }
05156     else {
05157 #endif
05158         counter = 0;
05159         for (i = 0; i < row; ++i) {
05160             Ps_marker[i] = counter;
05161             jj_row_begining = counter;
05162             jac[counter] = i;
05163             fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
05164             counter ++;

```

```

05165
05166         for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05167             i1 = jr[jj1];
05168             for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05169                 fasp_blas_smat_mul(&r[jj1*nb2], &aj[jj2*nb2], tmp, nb);
05170                 i2 = ja[jj2];
05171                 if (As_marker[i2] != i) {
05172                     As_marker[i2] = i;
05173                     for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05174                         i3 = jp[jj3];
05175                         fasp_blas_smat_mul(tmp, &pj[jj3*nb2], tmp+nb2, nb);
05176                         if (Ps_marker[i3] < jj_row_begining) {
05177                             Ps_marker[i3] = counter;
05178                             fasp_darray_cp(nb2, tmp+nb2, &acj[counter*nb2]);
05179                             jac[counter] = i3;
05180                             counter++;
05181                         }
05182                         else {
05183                             fasp_blas_darray_axpy(nb2, 1.0, tmp+nb2,
05184                                                     &acj[Ps_marker[i3]*nb2]);
05185                         }
05186                     }
05187                 }
05188                 else {
05189                     for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3++) {
05190                         i3 = jp[jj3];
05191                         fasp_blas_smat_mul(tmp, &pj[jj3*nb2], tmp+nb2, nb);
05192                         fasp_blas_darray_axpy(nb2, 1.0, tmp+nb2, &acj[Ps_marker[i3]*nb2]);
05193                     }
05194                 }
05195             }
05196         }
05197     }
05198 #ifdef _OPENMP
05199 }
05200 #endif
05201 // setup coarse matrix B
05202 B->ROW=row; B->COL=col;
05203 B->IA=iac; B->JA=jac; B->val=acj;
05204 B->NNZ=B->IA[B->ROW]-B->IA[0];
05205 B->nb=A->nb;
05206 B->storage_manner = A->storage_manner;
05207
05208 fasp_mem_free(Ps_marker); Ps_marker = NULL;
05209 fasp_mem_free(tmp); tmp = NULL;
05210 }
05211
05227 void fasp_blas_dbsr_rap_agg (const dBSRmat *R,
05228                             const dBSRmat *A,
05229                             const dBSRmat *P,
05230                             dBSRmat *B)
05231 {
05232     const INT row=R->ROW, col=P->COL, nb2=A->nb*A->nb;
05233
05234     const REAL *aj=A->val;
05235     const INT *ir=R->IA, *ia=A->IA, *ip=P->IA;
05236     const INT *jr=R->JA, *ja=A->JA, *jp=P->JA;
05237
05238     INT *iac, *jac;
05239     REAL *acj;
05240     INT *Ps_marker = NULL;
05241     INT *As_marker = NULL;
05242
05243 #ifdef _OPENMP
05244     INT *P_marker = NULL;
05245     INT *A_marker = NULL;
05246 #endif
05247
05248     INT i, i1, i2, i3, jj1, jj2, jj3;
05249     INT counter, jj_row_begining;
05250
05251     INT nthreads = 1;
05252
05253 #ifdef _OPENMP
05254     INT myid, mybegin, myend, Ctemp;
05255     nthreads = fasp_get_num_threads();
05256 #endif
05257
05258     INT n_coarse = row;
05259     INT n_fine = A->ROW;
05260     INT coarse_mul_nthreads = n_coarse * nthreads;

```

```

05261     INT fine_mul_nthreads = n_fine * nthreads;
05262     INT coarse_add_nthreads = n_coarse + nthreads;
05263     INT minus_one_length = coarse_mul_nthreads + fine_mul_nthreads;
05264     INT total_calloc = minus_one_length + coarse_add_nthreads + nthreads;
05265
05266     Ps_marker = (INT *)fasp_mem_calloc(total_calloc, sizeof(INT));
05267     As_marker = Ps_marker + coarse_mul_nthreads;
05268
05269     /*-----*
05270 * First Pass: Determine size of B and set up B_i *
05271 *-----*/
05272     iac = (INT *)fasp_mem_calloc(n_coarse+1, sizeof(INT));
05273
05274     fasp_iarray_set(minus_one_length, Ps_marker, -1);
05275
05276 #ifdef _OPENMP
05277     INT * RAP_temp = As_marker + fine_mul_nthreads;
05278     INT * part_end = RAP_temp + coarse_add_nthreads;
05279
05280     if (n_coarse > OPENMP_HOLDS) {
05281 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
05282 counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
05283         for (myid = 0; myid < nthreads; myid++) {
05284             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
05285             P_marker = Ps_marker + myid * n_coarse;
05286             A_marker = As_marker + myid * n_fine;
05287             counter = 0;
05288             for (i = mybegin; i < myend; ++i) {
05289                 P_marker[i] = counter;
05290                 jj_row_begining = counter;
05291                 counter ++;
05292                 for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05293                     i1 = jr[jj1];
05294                     for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05295                         i2 = ja[jj2];
05296                         if (A_marker[i2] != i) {
05297                             A_marker[i2] = i;
05298                             for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05299                                 i3 = jp[jj3];
05300                                 if (P_marker[i3] < jj_row_begining) {
05301                                     P_marker[i3] = counter;
05302                                     counter ++;
05303                                 }
05304                             }
05305                         }
05306                     }
05307                 }
05308                 RAP_temp[i+myid] = jj_row_begining;
05309             }
05310             RAP_temp[myend+myid] = counter;
05311             part_end[myid] = myend + myid + 1;
05312         }
05313         fasp_iarray_cp(part_end[0], RAP_temp, iac);
05314         counter = part_end[0];
05315         Ctemp = 0;
05316         for (i1 = 1; i1 < nthreads; i1++) {
05317             Ctemp += RAP_temp[part_end[i1-1]-1];
05318             for (jj1 = part_end[i1-1]+1; jj1 < part_end[i1]; jj1++) {
05319                 iac[counter] = RAP_temp[jj1] + Ctemp;
05320                 counter ++;
05321             }
05322         }
05323     }
05324     else {
05325 #endif
05326         counter = 0;
05327         for (i = 0; i < row; ++ i) {
05328             Ps_marker[i] = counter;
05329             jj_row_begining = counter;
05330             counter ++;
05331
05332             for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05333                 i1 = jr[jj1];
05334                 for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05335                     i2 = ja[jj2];
05336                     if (As_marker[i2] != i) {
05337                         As_marker[i2] = i;
05338                         for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05339                             i3 = jp[jj3];
05340                             if (Ps_marker[i3] < jj_row_begining) {
05341                                 Ps_marker[i3] = counter;

```

```

05342                                     counter ++;
05343                                     }
05344                                     }
05345                                     }
05346                                     }
05347                                     }
05348     iac[i] = jj_row_begining;
05349 }
05350 #ifdef _OPENMP
05351 }
05352 #endif
05353
05354     iac[row] = counter;
05355
05356     jac=(INT*) fasp_mem_calloc(iac[row], sizeof(INT));
05357
05358     acj=(REAL*) fasp_mem_calloc(iac[row]*nb2, sizeof(REAL));
05359
05360     fasp_iarray_set(minus_one_length, Ps_marker, -1);
05361
05362     /*-----*
05363 * Second Pass: compute entries of B=R*A*P *
05364 *-----*/
05365 #ifdef _OPENMP
05366     if (n_coarse > OPENMP_HOLDS) {
05367 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
05368 counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
05369         for (myid = 0; myid < nthreads; myid++) {
05370             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
05371             P_marker = Ps_marker + myid * n_coarse;
05372             A_marker = As_marker + myid * n_fine;
05373             counter = iac[mybegin];
05374             for (i = mybegin; i < myend; ++i) {
05375                 P_marker[i] = counter;
05376                 jj_row_begining = counter;
05377                 jac[counter] = i;
05378                 fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
05379                 counter ++;
05380
05381                 for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05382                     i1 = jr[jj1];
05383                     for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05384
05385                         i2 = ja[jj2];
05386                         if (A_marker[i2] != i) {
05387                             A_marker[i2] = i;
05388                             for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05389                                 i3 = jp[jj3];
05390
05391                                 if (P_marker[i3] < jj_row_begining) {
05392                                     P_marker[i3] = counter;
05393                                     fasp_darray_cp(nb2, &aj[jj2*nb2], &acj[counter*nb2]);
05394                                     jac[counter] = i3;
05395                                     counter ++;
05396                                 }
05397                                 else {
05398                                     fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
05399                                                             &acj[P_marker[i3]*nb2]);
05400                                 }
05401                             }
05402                         }
05403                     }
05404                     for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++ ) {
05405                         i3 = jp[jj3];
05406                         fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
05407                                                             &acj[P_marker[i3]*nb2]);
05408                     }
05409                 }
05410             }
05411         }
05412     }
05413 }
05414 }
05415 else {
05416 #endif
05417     counter = 0;
05418     for (i = 0; i < row; ++i) {
05419         Ps_marker[i] = counter;
05420         jj_row_begining = counter;
05421         jac[counter] = i;
05422         fasp_darray_set(nb2, &acj[counter*nb2], 0x0);

```

```

05423         counter ++;
05424
05425         for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
05426             i1 = jr[jj1];
05427             for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
05428
05429                 i2 = ja[jj2];
05430                 if (As_marker[i2] != i) {
05431                     As_marker[i2] = i;
05432                     for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
05433                         i3 = jp[jj3];
05434                         if (Ps_marker[i3] < jj_row_begining) {
05435                             Ps_marker[i3] = counter;
05436                             fasp_darray_cp(nb2, &aj[jj2*nb2], &acj[counter*nb2]);
05437                             jac[counter] = i3;
05438                             counter ++;
05439                         }
05440                         else {
05441                             fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
05442                                                     &acj[Ps_marker[i3]*nb2]);
05443                         }
05444                     }
05445                 }
05446                 else {
05447                     for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++ ) {
05448                         i3 = jp[jj3];
05449                         fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
05450                                                 &acj[Ps_marker[i3]*nb2]);
05451                     }
05452                 }
05453             }
05454         }
05455     }
05456 #ifdef _OPENMP
05457 }
05458 #endif
05459
05460     // setup coarse matrix B
05461     B->ROW=row; B->COL=col;
05462     B->IA=iac; B->JA=jac; B->val=acj;
05463     B->NNZ=B->IA[B->ROW]-B->IA[0];
05464     B->nb=A->nb;
05465     B->storage_manner = A->storage_manner;
05466
05467     fasp_mem_free(Ps_marker); Ps_marker = NULL;
05468 }
05469
05470 /*-----*/
05471 /*--      End of File      --*/
05472 /*-----*/

```

9.91 BlasSpmvCSR.c File Reference

Linear algebraic operations for [dCSRmat](#) matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_blas_dcsr_add](#) (const [dCSRmat](#) *A, const [REAL](#) alpha, const [dCSRmat](#) *B, const [REAL](#) beta, [dCSRmat](#) *C)
compute $C = \alpha * A + \beta * B$ in CSR format
- void [fasp_blas_dcsr_axm](#) ([dCSRmat](#) *A, const [REAL](#) alpha)
Multiply a sparse matrix A in CSR format by a scalar alpha.
- void [fasp_blas_dcsr_mxv](#) (const [dCSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
Matrix-vector multiplication $y = A * x$.

- void [fasp_blas_dcsr_mxv_agg](#) (const [dCSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = A*x$ (nonzeros of $A = 1$)*
- void [fasp_blas_dcsr_aApy](#) (const [REAL](#) alpha, const [dCSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha*A*x + y$.*
- void [fasp_blas_ldcsr_aApy](#) (const [REAL](#) alpha, const [dCSRmat](#) *A, const [LONGREAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha*A*x + y$.*
- void [fasp_blas_dcsr_aApy_agg](#) (const [REAL](#) alpha, const [dCSRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha*A*x + y$ (nonzeros of $A = 1$)*
- [REAL](#) [fasp_blas_dcsr_vmv](#) (const [dCSRmat](#) *A, const [REAL](#) *x, const [REAL](#) *y)
*vector-Matrix-vector multiplication $\alpha = y'*A*x$*
- void [fasp_blas_dcsr_mxm](#) (const [dCSRmat](#) *A, const [dCSRmat](#) *B, [dCSRmat](#) *C)
*Sparse matrix multiplication $C=A*B$.*
- void [fasp_blas_dcsr_rap](#) (const [dCSRmat](#) *R, const [dCSRmat](#) *A, const [dCSRmat](#) *P, [dCSRmat](#) *RAP)
*Triple sparse matrix multiplication $B=R*A*P$.*
- void [fasp_blas_dcsr_rap_agg](#) (const [dCSRmat](#) *R, const [dCSRmat](#) *A, const [dCSRmat](#) *P, [dCSRmat](#) *RAP)
*Triple sparse matrix multiplication $B=R*A*P$ (nonzeros of $R, P = 1$)*
- void [fasp_blas_dcsr_rap_agg1](#) (const [dCSRmat](#) *R, const [dCSRmat](#) *A, const [dCSRmat](#) *P, [dCSRmat](#) *B)
*Triple sparse matrix multiplication $B=R*A*P$ (nonzeros of $R, P = 1$)*
- void [fasp_blas_dcsr_ptap](#) (const [dCSRmat](#) *Pt, const [dCSRmat](#) *A, const [dCSRmat](#) *P, [dCSRmat](#) *Ac)
*Triple sparse matrix multiplication $B=P'*A*P$.*
- [dCSRmat](#) [fasp_blas_dcsr_rap2](#) ([INT](#) *ir, [INT](#) *jr, [REAL](#) *r, [INT](#) *ia, [INT](#) *ja, [REAL](#) *a, [INT](#) *ipt, [INT](#) *jpt, [REAL](#) *pt, [INT](#) n, [INT](#) nc, [INT](#) *maxrpout, [INT](#) *ipin, [INT](#) *jpin)
*Compute $R*A*P$.*
- void [fasp_blas_dcsr_rap4](#) ([dCSRmat](#) *R, [dCSRmat](#) *A, [dCSRmat](#) *P, [dCSRmat](#) *B, [INT](#) *icor_ysk)
*Triple sparse matrix multiplication $B=R*A*P$.*

Variables

- unsigned long [total_alloc_mem](#)
- unsigned long [total_alloc_count](#)

9.91.1 Detailed Description

Linear algebraic operations for [dCSRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaSparseCSR.c](#), [BlaSparseUtil.c](#), and [BlaArray.c](#)

Sparse functions usually contain three runs. The three runs are all the same but they serve different purpose.

Example: If you do $c=a+b$:

- first do a dry run to find the number of non-zeroes and form ic;
- allocate space (memory) for jc and form this one;
- if you only care about a "boolean" result of the addition, you stop here;
- you call another routine, which uses ic and jc to perform the addition.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSpmvCSR.c](#).

9.91.2 Function Documentation

9.91.2.1 fasp_blas_dcsr_aApy()

```
void fasp_blas_dcsr_aApy (
    const REAL alpha,
    const dCSRmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = \alpha * A * x + y$.

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/26/2012

Definition at line [493](#) of file [BlaSpmvCSR.c](#).

9.91.2.2 fasp_blas_dcsr_aApy_agg()

```
void fasp_blas_dcsr_aApy_agg (
    const REAL alpha,
    const dCSRmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = \alpha * A * x + y$ (nonzeros of $A = 1$)

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Xiaozhe Hu

Date

02/22/2011

Modified by Chunsheng Feng, Zheng Li on 08/29/2012
 Definition at line 724 of file [BlasSpmvCSR.c](#).

9.91.2.3 fasp_blas_dcsr_add()

```
SHORT fasp_blas_dcsr_add (
    const dCSRmat * A,
    const REAL alpha,
    const dCSRmat * B,
    const REAL beta,
    dCSRmat * C )
```

compute $C = \alpha * A + \beta * B$ in CSR format

Parameters

<i>A</i>	Pointer to dCSRmat matrix
<i>alpha</i>	REAL factor alpha
<i>B</i>	Pointer to dCSRmat matrix
<i>beta</i>	REAL factor beta
<i>C</i>	Pointer to dCSRmat matrix

Returns

FASP_SUCCESS if succeed, ERROR if not

Author

Xiaozhe Hu

Date

11/07/2009

Modified by Chunsheng Feng, Zheng Li on 06/29/2012
 Definition at line 60 of file [BlasSpmvCSR.c](#).

9.91.2.4 fasp_blas_dcsr_axm()

```
void fasp_blas_dcsr_axm (
    dCSRmat * A,
    const REAL alpha )
```

Multiply a sparse matrix A in CSR format by a scalar alpha.

Parameters

<i>A</i>	Pointer to dCSRmat matrix A
<i>alpha</i>	REAL factor alpha

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Zheng Li on 06/29/2012
 Definition at line 219 of file [BlaSpmvCSR.c](#).

9.91.2.5 fasp_blas_dcsr_mxm()

```
void fasp_blas_dcsr_mxm (
    const dCSRmat * A,
    const dCSRmat * B,
    dCSRmat * C )
```

Sparse matrix multiplication $C=A*B$.

Parameters

<i>A</i>	Pointer to the dCSRmat matrix A
<i>B</i>	Pointer to the dCSRmat matrix B
<i>C</i>	Pointer to dCSRmat matrix equal to $A*B$

Author

Xiaozhe Hu

Date

11/07/2009

Warning

This fct will be replaced! –Chensong

Definition at line 888 of file [BlaSpmvCSR.c](#).

9.91.2.6 fasp_blas_dcsr_mxv()

```
void fasp_blas_dcsr_mxv (
    const dCSRmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = A*x$.

Parameters

<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/26/2012
 Definition at line 241 of file [BlaSpmvCSR.c](#).

9.91.2.7 fasp_blas_dcsr_mxv_agg()

```
void fasp_blas_dcsr_mxv_agg (
    const dCSRmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = A*x$ (nonzeros of $A = 1$)

Parameters

<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Xiaozhe Hu

Date

02/22/2011

Modified by Chunsheng Feng, Zheng Li on 08/29/2012
 Definition at line 437 of file [BlaSpmvCSR.c](#).

9.91.2.8 fasp_blas_dcsr_ptap()

```
void fasp_blas_dcsr_ptap (
    const dCSRmat * Pt,
    const dCSRmat * A,
    const dCSRmat * P,
    dCSRmat * Ac )
```

Triple sparse matrix multiplication $B=P'*A*P$.

Parameters

<i>Pt</i>	Pointer to the restriction matrix
<i>A</i>	Pointer to the fine coefficient matrix
<i>P</i>	Pointer to the prolongation matrix
<i>Ac</i>	Pointer to the coarse coefficient matrix (output)

Author

Ludmil Zikatanov, Chensong Zhang

Date

05/10/2010

Modified by Chunsheng Feng, Zheng Li on 10/19/2012

Note

Driver to compute triple matrix product P^*A^*P using Itz CSR format. In Itz format: $ia[0]=1$, $ja[0]$ and $a[0]$ are used as usual. When called from Fortran, $ia[0]$, $ja[0]$ and $a[0]$ will be just $ia(1), ja(1), a(1)$. For the indices, $ia_itz[k] = ia_usual[k]+1$, $ja_itz[k] = ja_usual[k]+1$, $a_itz[k] = a_usual[k]$.

Definition at line 1734 of file [BlaSpmvCSR.c](#).**9.91.2.9 fasp_blas_dcsr_rap()**

```
void fasp_blas_dcsr_rap (
    const dCSRmat * R,
    const dCSRmat * A,
    const dCSRmat * P,
    dCSRmat * RAP )
```

Triple sparse matrix multiplication $B=R*A*P$.**Parameters**

<i>R</i>	Pointer to the dCSRmat matrix R
<i>A</i>	Pointer to the dCSRmat matrix A
<i>P</i>	Pointer to the dCSRmat matrix P
<i>RAP</i>	Pointer to dCSRmat matrix equal to $R*A*P$

Author

Xuehai Huang, Chensong Zhang

Date

05/10/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/26/2012

Note

Ref. R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. Advances in Computational Mathematics, 1 (1993), pp. 127-137.

Definition at line 994 of file [BlaSpmvCSR.c](#).**9.91.2.10 fasp_blas_dcsr_rap2()**

```
dCSRmat fasp_blas_dcsr_rap2 (
    INT * ir,
    INT * jr,
```

```

REAL * r,
INT * ia,
INT * ja,
REAL * a,
INT * ipt,
INT * jpt,
REAL * pt,
INT n,
INT nc,
INT * maxrpout,
INT * ipin,
INT * jpin )

```

Compute $R \cdot A \cdot P$.

Author

Ludmil Zikatanov

Date

04/08/2010

Note

It uses [dCSRmat](#) only. The functions called from here are in `sparse_util.c`. Not used for the moment!

Definition at line [1842](#) of file [BlasSpmvCSR.c](#).

9.91.2.11 fasp_blas_dcsr_rap4()

```

void fasp_blas_dcsr_rap4 (
    dCSRmat * R,
    dCSRmat * A,
    dCSRmat * P,
    dCSRmat * B,
    INT * icor_ysk )

```

Triple sparse matrix multiplication $B = R \cdot A \cdot P$.

Parameters

<i>R</i>	pointer to the dCSRmat matrix
<i>A</i>	pointer to the dCSRmat matrix
<i>P</i>	pointer to the dCSRmat matrix
<i>B</i>	pointer to dCSRmat matrix equal to $R \cdot A \cdot P$
<i>icor_ysk</i>	pointer to the array

Author

Feng Chunsheng, Yue Xiaoqiang

Date

08/02/2011

Note

Ref. R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. Advances in Computational Mathematics, 1 (1993), pp. 127-137.

Definition at line 1930 of file [BlaSpmvCSR.c](#).

9.91.2.12 fasp_blas_dcsr_rap_agg()

```
void fasp_blas_dcsr_rap_agg (
    const dCSRmat * R,
    const dCSRmat * A,
    const dCSRmat * P,
    dCSRmat * RAP )
```

Triple sparse matrix multiplication $B=R*A*P$ (nonzeros of R, $P = 1$)

Parameters

<i>R</i>	Pointer to the dCSRmat matrix R
<i>A</i>	Pointer to the dCSRmat matrix A
<i>P</i>	Pointer to the dCSRmat matrix P
<i>RAP</i>	Pointer to dCSRmat matrix equal to $R*A*P$

Author

Xiaozhe Hu

Date

05/10/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/26/2012

Definition at line 1269 of file [BlaSpmvCSR.c](#).

9.91.2.13 fasp_blas_dcsr_rap_agg1()

```
void fasp_blas_dcsr_rap_agg1 (
    const dCSRmat * R,
    const dCSRmat * A,
    const dCSRmat * P,
    dCSRmat * B )
```

Triple sparse matrix multiplication $B=R*A*P$ (nonzeros of R, $P = 1$)

Parameters

<i>R</i>	Pointer to the dCSRmat matrix R
<i>A</i>	Pointer to the dCSRmat matrix A
<i>P</i>	Pointer to the dCSRmat matrix P
<i>B</i>	Pointer to dCSRmat matrix equal to $R*A*P$

Author

Xiaozhe Hu

Date

02/21/2011

Note

Ref. R.E. Bank and C.C. Douglas. SMMP: Sparse Matrix Multiplication Package. Advances in Computational Mathematics, 1 (1993), pp. 127-137.

Definition at line 1530 of file BlasSpmvCSR.c.

9.91.2.14 fasp_blas_dcsr_vmv()

```
REAL fasp_blas_dcsr_vmv (
    const dCSRmat * A,
    const REAL * x,
    const REAL * y )
```

vector-Matrix-vector multiplication $y = y' * A * x$

Parameters

<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Chensong Zhang

Date

07/01/2009

Definition at line 834 of file BlasSpmvCSR.c.

9.91.2.15 fasp_blas_ldcsr_aAxy()

```
void fasp_blas_ldcsr_aAxy (
    const REAL alpha,
    const dCSRmat * A,
    const LONGREAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = \text{alpha} * A * x + y$.

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to dCSRmat matrix A
<i>x</i>	Pointer to array x
<i>y</i>	Pointer to array y

Author

Chensong Zhang

Date

07/01/2009

Modified by TingLai on 08/01/2022

Definition at line 608 of file [BlaSpmvCSR.c](#).**9.91.3 Variable Documentation****9.91.3.1 total_alloc_count**

unsigned long total_alloc_count [extern]

Total number of allocatations

9.91.3.2 total_alloc_mem

unsigned long total_alloc_mem [extern]

Total allocated memory

9.92 BlaSpmvCSR.c[Go to the documentation of this file.](#)

```

00001
00024 #include <math.h>
00025 #include <time.h>
00026
00027 #ifdef _OPENMP
00028 #include <omp.h>
00029 #endif
00030
00031 #include "fasp.h"
00032 #include "fasp_funcs.h"
00033
00034 extern unsigned long total_alloc_mem;
00035 extern unsigned long total_alloc_count;
00037 /*-----*/
00038 /*--      Public Functions      --*/
00039 /*-----*/
00040
00060 SHORT fasp_blas_dcsr_add(const dCSRmat* A, const REAL alpha, const dCSRmat* B,
00061                        const REAL beta, dCSRmat* C)
00062 {
00063     INT i, j, k, l;
00064     INT count = 0, added, countrow;
00065
00066     SHORT status = FASP_SUCCESS, use_omp = FALSE;
00067
00068 #ifdef _OPENMP
00069     INT mybegin, myend, myid, nthreads;
00070     if (A->nnz > OPENMP_HOLDS) {
00071         use_omp = TRUE;
00072         nthreads = fasp_get_num_threads();
00073     }
00074 #endif
00075
00076     if (A->row != B->row || A->col != B->col) {
00077         printf("### ERROR: Matrix sizes do not match!\n");
00078         status = ERROR_MAT_SIZE;
00079         goto FINISHED;
00080     }
00081
00082     if (A == NULL && B == NULL) {

```



```

00083     C->row = 0;
00084     C->col = 0;
00085     C->nnz = 0;
00086     status = FASP_SUCCESS;
00087     goto FINISHED;
00088 }
00089
00090 if (A->nnz == 0 && B->nnz == 0) {
00091     C->row = A->row;
00092     C->col = A->col;
00093     C->nnz = A->nnz;
00094     status = FASP_SUCCESS;
00095     goto FINISHED;
00096 }
00097
00098 // empty matrix A
00099 if (A->nnz == 0 || A == NULL) {
00100     fasp_dcsr_alloc(B->row, B->col, B->nnz, C);
00101     memcpy(C->IA, B->IA, (B->row + 1) * sizeof(INT));
00102     memcpy(C->JA, B->JA, (B->nnz) * sizeof(INT));
00103
00104     if (use_omp) {
00105 #ifdef _OPENMP
00106 #pragma omp parallel private(myid, mybegin, myend, i)
00107         {
00108             myid = omp_get_thread_num();
00109             fasp_get_start_end(myid, nthreads, A->nnz, &mybegin, &myend);
00110             for (i = mybegin; i < myend; ++i) C->val[i] = B->val[i] * beta;
00111         }
00112 #endif
00113     } else {
00114         for (i = 0; i < A->nnz; ++i) C->val[i] = B->val[i] * beta;
00115     }
00116
00117     status = FASP_SUCCESS;
00118     goto FINISHED;
00119 }
00120
00121 // empty matrix B
00122 if (B->nnz == 0 || B == NULL) {
00123     fasp_dcsr_alloc(A->row, A->col, A->nnz, C);
00124     memcpy(C->IA, A->IA, (A->row + 1) * sizeof(INT));
00125     memcpy(C->JA, A->JA, (A->nnz) * sizeof(INT));
00126
00127     if (use_omp) {
00128 #ifdef _OPENMP
00129         INT mybegin, myend, myid;
00130 #pragma omp parallel private(myid, mybegin, myend, i)
00131         {
00132             myid = omp_get_thread_num();
00133             fasp_get_start_end(myid, nthreads, A->nnz, &mybegin, &myend);
00134             for (i = mybegin; i < myend; ++i) C->val[i] = A->val[i] * alpha;
00135         }
00136 #endif
00137     } else {
00138         for (i = 0; i < A->nnz; ++i) C->val[i] = A->val[i] * alpha;
00139     }
00140
00141     status = FASP_SUCCESS;
00142     goto FINISHED;
00143 }
00144
00145 C->row = A->row;
00146 C->col = A->col;
00147
00148 C->IA = (INT*) fasp_mem_malloc(C->row + 1, sizeof(INT));
00149
00150 // allocate work space for C->JA and C->val
00151 C->JA = (INT*) fasp_mem_malloc(A->nnz + B->nnz, sizeof(INT));
00152
00153 C->val = (REAL*) fasp_mem_malloc(A->nnz + B->nnz, sizeof(REAL));
00154
00155 // initial C->IA
00156 memset(C->IA, 0, sizeof(INT) * (C->row + 1));
00157 memset(C->JA, -1, sizeof(INT) * (A->nnz + B->nnz));
00158
00159 for (i = 0; i < A->row; ++i) {
00160     countrow = 0;
00161     for (j = A->IA[i]; j < A->IA[i + 1]; ++j) {
00162         C->val[count] = alpha * A->val[j];
00163         C->JA[count] = A->JA[j];

```

```

00164         C->IA[i + 1]++;
00165         count++;
00166         countrow++;
00167     } // end for js
00168
00169     for (k = B->IA[i]; k < B->IA[i + 1]; ++k) {
00170         added = 0;
00171
00172         for (l = C->IA[i]; l < C->IA[i] + countrow + 1; l++) {
00173             if (B->JA[k] == C->JA[l]) {
00174                 C->val[l] = C->val[l] + beta * B->val[k];
00175                 added = 1;
00176                 break;
00177             }
00178         } // end for l
00179
00180         if (added == 0) {
00181             C->val[count] = beta * B->val[k];
00182             C->JA[count] = B->JA[k];
00183             C->IA[i + 1]++;
00184             count++;
00185         }
00186
00187     } // end for k
00188
00189     C->IA[i + 1] += C->IA[i];
00190 }
00191
00192 C->nnz = count;
00193 C->JA = (INT*)fasp_mem_realloc(C->JA, (count) * sizeof(INT));
00194 C->val = (REAL*)fasp_mem_realloc(C->val, (count) * sizeof(REAL));
00195
00196 #if MULTI_COLOR_ORDER
00197     C->color = 0;
00198     C->IC = NULL;
00199     C->ICMAP = NULL;
00200 #endif
00201
00202 FINISHED:
00203     return status;
00204 }
00205
00219 void fasp_blas_dcsr_axm(dCSRmat* A, const REAL alpha)
00220 {
00221     const INT nnz = A->nnz;
00222
00223     // A direct calculation can be written as:
00224     fasp_blas_darray_ax(nnz, alpha, A->val);
00225 }
00226
00241 void fasp_blas_dcsr_mxv(const dCSRmat* A, const REAL* x, REAL* y)
00242 {
00243     const INT m = A->row;
00244     const INT * ia = A->IA, *ja = A->JA;
00245     const REAL* aj = A->val;
00246
00247     INT i, k, begin_row, end_row, nnz_row;
00248     register REAL temp;
00249
00250     SHORT nthreads = 1, use_openmp = FALSE;
00251
00252 #ifdef _OPENMP
00253     if (m > OPENMP_HOLDS) {
00254         use_openmp = TRUE;
00255         nthreads = fasp_get_num_threads();
00256     }
00257 #endif
00258
00259     if (use_openmp) {
00260         INT myid, mybegin, myend;
00261
00262 #ifdef _OPENMP
00263 #pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, \
00264 nnz_row, k)
00265 #endif
00266         for (myid = 0; myid < nthreads; myid++) {
00267             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00268             for (i = mybegin; i < myend; ++i) {
00269                 temp = 0.0;
00270                 begin_row = ia[i];
00271                 end_row = ia[i + 1];

```

```

00272         nnz_row = end_row - begin_row;
00273         switch (nnz_row) {
00274             case 3:
00275                 k = begin_row;
00276                 temp += aj[k] * x[ja[k]];
00277                 k++;
00278                 temp += aj[k] * x[ja[k]];
00279                 k++;
00280                 temp += aj[k] * x[ja[k]];
00281                 break;
00282             case 4:
00283                 k = begin_row;
00284                 temp += aj[k] * x[ja[k]];
00285                 k++;
00286                 temp += aj[k] * x[ja[k]];
00287                 k++;
00288                 temp += aj[k] * x[ja[k]];
00289                 k++;
00290                 temp += aj[k] * x[ja[k]];
00291                 break;
00292             case 5:
00293                 k = begin_row;
00294                 temp += aj[k] * x[ja[k]];
00295                 k++;
00296                 temp += aj[k] * x[ja[k]];
00297                 k++;
00298                 temp += aj[k] * x[ja[k]];
00299                 k++;
00300                 temp += aj[k] * x[ja[k]];
00301                 k++;
00302                 temp += aj[k] * x[ja[k]];
00303                 break;
00304             case 6:
00305                 k = begin_row;
00306                 temp += aj[k] * x[ja[k]];
00307                 k++;
00308                 temp += aj[k] * x[ja[k]];
00309                 k++;
00310                 temp += aj[k] * x[ja[k]];
00311                 k++;
00312                 temp += aj[k] * x[ja[k]];
00313                 k++;
00314                 temp += aj[k] * x[ja[k]];
00315                 k++;
00316                 temp += aj[k] * x[ja[k]];
00317                 break;
00318             case 7:
00319                 k = begin_row;
00320                 temp += aj[k] * x[ja[k]];
00321                 k++;
00322                 temp += aj[k] * x[ja[k]];
00323                 k++;
00324                 temp += aj[k] * x[ja[k]];
00325                 k++;
00326                 temp += aj[k] * x[ja[k]];
00327                 k++;
00328                 temp += aj[k] * x[ja[k]];
00329                 k++;
00330                 temp += aj[k] * x[ja[k]];
00331                 k++;
00332                 temp += aj[k] * x[ja[k]];
00333                 break;
00334             default:
00335                 for (k = begin_row; k < end_row; ++k) {
00336                     temp += aj[k] * x[ja[k]];
00337                 }
00338                 break;
00339         }
00340         y[i] = temp;
00341     }
00342 }
00343 }
00344
00345 else {
00346     for (i = 0; i < m; ++i) {
00347         temp = 0.0;
00348         begin_row = ia[i];
00349         end_row = ia[i + 1];
00350         nnz_row = end_row - begin_row;
00351         switch (nnz_row) {
00352             case 3:

```

```

00353         k = begin_row;
00354         temp += aj[k] * x[ja[k]];
00355         k++;
00356         temp += aj[k] * x[ja[k]];
00357         k++;
00358         temp += aj[k] * x[ja[k]];
00359         break;
00360     case 4:
00361         k = begin_row;
00362         temp += aj[k] * x[ja[k]];
00363         k++;
00364         temp += aj[k] * x[ja[k]];
00365         k++;
00366         temp += aj[k] * x[ja[k]];
00367         k++;
00368         temp += aj[k] * x[ja[k]];
00369         break;
00370     case 5:
00371         k = begin_row;
00372         temp += aj[k] * x[ja[k]];
00373         k++;
00374         temp += aj[k] * x[ja[k]];
00375         k++;
00376         temp += aj[k] * x[ja[k]];
00377         k++;
00378         temp += aj[k] * x[ja[k]];
00379         k++;
00380         temp += aj[k] * x[ja[k]];
00381         break;
00382     case 6:
00383         k = begin_row;
00384         temp += aj[k] * x[ja[k]];
00385         k++;
00386         temp += aj[k] * x[ja[k]];
00387         k++;
00388         temp += aj[k] * x[ja[k]];
00389         k++;
00390         temp += aj[k] * x[ja[k]];
00391         k++;
00392         temp += aj[k] * x[ja[k]];
00393         k++;
00394         temp += aj[k] * x[ja[k]];
00395         break;
00396     case 7:
00397         k = begin_row;
00398         temp += aj[k] * x[ja[k]];
00399         k++;
00400         temp += aj[k] * x[ja[k]];
00401         k++;
00402         temp += aj[k] * x[ja[k]];
00403         k++;
00404         temp += aj[k] * x[ja[k]];
00405         k++;
00406         temp += aj[k] * x[ja[k]];
00407         k++;
00408         temp += aj[k] * x[ja[k]];
00409         k++;
00410         temp += aj[k] * x[ja[k]];
00411         break;
00412     default:
00413         for (k = begin_row; k < end_row; ++k) {
00414             temp += aj[k] * x[ja[k]];
00415         }
00416         break;
00417     }
00418     y[i] = temp;
00419 }
00420 }
00421 }
00422
00437 void fasp_blas_dcsr_mxv_agg(const dCSRmat* A, const REAL* x, REAL* y)
00438 {
00439     const INT m = A->row;
00440     const INT * ia = A->IA, *ja = A->JA;
00441     INT i, k, begin_row, end_row;
00442     register REAL temp;
00443
00444     #ifdef _OPENMP
00445         // variables for OpenMP
00446         INT myid, mybegin, myend;
00447         INT nthreads = fasp_get_num_threads();

```

```

00448 #endif
00449
00450 #ifdef _OPENMP
00451     if (m > OPENMP_HOLDS) {
00452         #pragma omp parallel for private(myid, i, mybegin, myend, temp, begin_row, end_row, k)
00453         for (myid = 0; myid < nthreads; myid++) {
00454             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00455             for (i = mybegin; i < myend; i++) {
00456                 temp = 0.0;
00457                 begin_row = ia[i];
00458                 end_row = ia[i + 1];
00459                 for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00460                 y[i] = temp;
00461             }
00462         }
00463     } else {
00464         #endif
00465         for (i = 0; i < m; ++i) {
00466             temp = 0.0;
00467             begin_row = ia[i];
00468             end_row = ia[i + 1];
00469             for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00470             y[i] = temp;
00471         }
00472     #ifdef _OPENMP
00473     }
00474 #endif
00475 }
00476
00493 void fasp_blas_dcsr_aAxy(const REAL alpha, const dCSRmat* A, const REAL* x, REAL* y)
00494 {
00495     const INT m = A->row;
00496     const INT * ia = A->IA, *ja = A->JA;
00497     const REAL* aj = A->val;
00498     INT i, k, begin_row, end_row;
00499     register REAL temp;
00500     SHORT nthreads = 1, use_omp = FALSE;
00501
00502 #ifdef _OPENMP
00503     if (m > OPENMP_HOLDS) {
00504         use_omp = TRUE;
00505         nthreads = fasp_get_num_threads();
00506     }
00507 #endif
00508     if (alpha == 1.0) {
00509         if (use_omp) {
00510             INT myid, mybegin, myend;
00511             #ifdef _OPENMP
00512             #pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
00513             #endif
00514             for (myid = 0; myid < nthreads; myid++) {
00515                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00516                 for (i = mybegin; i < myend; ++i) {
00517                     temp = 0.0;
00518                     begin_row = ia[i];
00519                     end_row = ia[i + 1];
00520                     for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00521                     y[i] += temp;
00522                 }
00523             }
00524         } else {
00525             for (i = 0; i < m; ++i) {
00526                 temp = 0.0;
00527                 begin_row = ia[i];
00528                 end_row = ia[i + 1];
00529                 for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00530                 y[i] += temp;
00531             }
00532         }
00533     }
00534
00535     else if (alpha == -1.0) {
00536         if (use_omp) {
00537             INT myid, mybegin, myend;
00538             INT S = 0;
00539             #ifdef _OPENMP
00540             #pragma omp parallel for private(myid, mybegin, myend, temp, i, begin_row, end_row, k)
00541             #endif
00542
00543             for (myid = 0; myid < nthreads; myid++) {
00544                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);

```

```

00545         for (i = mybegin; i < myend; ++i) {
00546             temp      = 0.0;
00547             begin_row  = ia[i];
00548             end_row    = ia[i + 1];
00549             for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00550             y[i] -= temp;
00551         }
00552     }
00553 } else {
00554     for (i = 0; i < m; ++i) {
00555         temp      = 0.0;
00556         begin_row  = ia[i];
00557         end_row    = ia[i + 1];
00558         for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00559         y[i] -= temp;
00560     }
00561 }
00562 }
00563
00564 else {
00565     if (use_omp) {
00566         INT myid, mybegin, myend;
00567 #ifdef _OPENMP
00568 #pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
00569 #endif
00570         for (myid = 0; myid < nthreads; myid++) {
00571             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00572             for (i = mybegin; i < myend; ++i) {
00573                 temp      = 0.0;
00574                 begin_row  = ia[i];
00575                 end_row    = ia[i + 1];
00576                 for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00577                 y[i] += temp * alpha;
00578             }
00579         }
00580     } else {
00581         for (i = 0; i < m; ++i) {
00582             temp      = 0.0;
00583             begin_row  = ia[i];
00584             end_row    = ia[i + 1];
00585             for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00586             y[i] += temp * alpha;
00587         }
00588     }
00589 }
00590 }
00591
00608 void fasp_blas_ldcsr_aAxy(const REAL alpha, const dCSRmat* A, const LONGREAL* x,
00609                          REAL* y)
00610 {
00611     const INT      m = A->row;
00612     const INT *    ia = A->IA, *ja = A->JA;
00613     const REAL*    aj = A->val;
00614     INT            i, k, begin_row, end_row;
00615     register LONGREAL temp;
00616
00617     SHORT nthreads = 1, use_omp = FALSE;
00618
00619 #ifdef _OPENMP
00620     if (m > OPENMP_HOLDS) {
00621         use_omp = TRUE;
00622         nthreads = fasp_get_num_threads();
00623     }
00624 #endif
00625
00626     if (alpha == 1.0) {
00627         if (use_omp) {
00628             INT myid, mybegin, myend;
00629 #ifdef _OPENMP
00630 #pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
00631 #endif
00632             for (myid = 0; myid < nthreads; myid++) {
00633                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00634                 for (i = mybegin; i < myend; ++i) {
00635                     temp      = 0.0;
00636                     begin_row  = ia[i];
00637                     end_row    = ia[i + 1];
00638                     for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00639                     y[i] += temp;
00640                 }
00641             }

```

```

00642         } else {
00643             for (i = 0; i < m; ++i) {
00644                 temp = 0.0;
00645                 begin_row = ia[i];
00646                 end_row = ia[i + 1];
00647                 for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00648                 y[i] += temp;
00649             }
00650         }
00651     }
00652
00653     else if (alpha == -1.0) {
00654         if (use_omp) {
00655             INT myid, mybegin, myend;
00656 #ifdef _OPENMP
00657 #pragma omp parallel for private(myid, mybegin, myend, temp, i, begin_row, end_row, k)
00658 #endif
00659             for (myid = 0; myid < nthreads; myid++) {
00660                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00661                 for (i = mybegin; i < myend; ++i) {
00662                     temp = 0.0;
00663                     begin_row = ia[i];
00664                     end_row = ia[i + 1];
00665                     for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00666                     y[i] -= temp;
00667                 }
00668             }
00669         } else {
00670             for (i = 0; i < m; ++i) {
00671                 temp = 0.0;
00672                 begin_row = ia[i];
00673                 end_row = ia[i + 1];
00674                 for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00675                 y[i] -= temp;
00676             }
00677         }
00678     }
00679
00680     else {
00681         if (use_omp) {
00682             INT myid, mybegin, myend;
00683 #ifdef _OPENMP
00684 #pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
00685 #endif
00686             for (myid = 0; myid < nthreads; myid++) {
00687                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00688                 for (i = mybegin; i < myend; ++i) {
00689                     temp = 0.0;
00690                     begin_row = ia[i];
00691                     end_row = ia[i + 1];
00692                     for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00693                     y[i] += temp * alpha;
00694                 }
00695             }
00696         } else {
00697             for (i = 0; i < m; ++i) {
00698                 temp = 0.0;
00699                 begin_row = ia[i];
00700                 end_row = ia[i + 1];
00701                 for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00702                 y[i] += temp * alpha;
00703             }
00704         }
00705     }
00706 }
00707
00724 void fasp_blas_dcsr_aApy_agg(const REAL alpha, const dCSRmat* A, const REAL* x,
00725                             REAL* y)
00726 {
00727     const INT m = A->row;
00728     const INT *ia = A->IA, *ja = A->JA;
00729
00730     INT i, k, begin_row, end_row;
00731     register REAL temp;
00732
00733     if (alpha == 1.0) {
00734 #ifdef _OPENMP
00735         if (m > OPENMP_HOLDS) {
00736             INT myid, mybegin, myend;
00737             INT nthreads = fasp_get_num_threads();
00738 #pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)

```

```

00739         for (myid = 0; myid < nthreads; myid++) {
00740             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00741             for (i = mybegin; i < myend; ++i) {
00742                 temp = 0.0;
00743                 begin_row = ia[i];
00744                 end_row = ia[i + 1];
00745                 for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00746                 y[i] += temp;
00747             }
00748         }
00749     } else {
00750 #endif
00751         for (i = 0; i < m; ++i) {
00752             temp = 0.0;
00753             begin_row = ia[i];
00754             end_row = ia[i + 1];
00755             for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00756             y[i] += temp;
00757         }
00758 #ifdef _OPENMP
00759     }
00760 #endif
00761     } else if (alpha == -1.0) {
00762 #ifdef _OPENMP
00763         if (m > OPENMP_HOLDS) {
00764             INT myid, mybegin, myend;
00765             INT nthreads = fasp_get_num_threads();
00766 #pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)
00767             for (myid = 0; myid < nthreads; myid++) {
00768                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00769                 for (i = mybegin; i < myend; ++i) {
00770                     temp = 0.0;
00771                     begin_row = ia[i];
00772                     end_row = ia[i + 1];
00773                     for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00774                     y[i] -= temp;
00775                 }
00776             }
00777         } else {
00778 #endif
00779             for (i = 0; i < m; ++i) {
00780                 temp = 0.0;
00781                 begin_row = ia[i];
00782                 end_row = ia[i + 1];
00783                 for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00784                 y[i] -= temp;
00785             }
00786 #ifdef _OPENMP
00787         }
00788 #endif
00789     }
00790
00791     else {
00792 #ifdef _OPENMP
00793         if (m > OPENMP_HOLDS) {
00794             INT myid, mybegin, myend;
00795             INT nthreads = fasp_get_num_threads();
00796 #pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)
00797             for (myid = 0; myid < nthreads; myid++) {
00798                 fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00799                 for (i = mybegin; i < myend; ++i) {
00800                     temp = 0.0;
00801                     begin_row = ia[i];
00802                     end_row = ia[i + 1];
00803                     for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00804                     y[i] += temp * alpha;
00805                 }
00806             }
00807         } else {
00808 #endif
00809             for (i = 0; i < m; ++i) {
00810                 temp = 0.0;
00811                 begin_row = ia[i];
00812                 end_row = ia[i + 1];
00813                 for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
00814                 y[i] += temp * alpha;
00815             }
00816 #ifdef _OPENMP
00817         }
00818 #endif
00819     }

```



```

00820 }
00821
00834 REAL fasp_blas_dcsr_vmv(const dCSRmat* A, const REAL* x, const REAL* y)
00835 {
00836     register REAL value = 0.0;
00837     const INT m = A->row;
00838     const INT * ia = A->IA, *ja = A->JA;
00839     const REAL* aj = A->val;
00840     INT i, k, begin_row, end_row;
00841     register REAL temp;
00842
00843     SHORT use_omp = FALSE;
00844
00845     #ifdef _OPENMP
00846     if (m > OPENMP_HOLDS) {
00847         use_omp = TRUE;
00848     }
00849     #endif
00850
00851     if (use_omp) {
00852     #ifdef _OPENMP
00853     #pragma omp parallel for reduction(+: value) private(i, temp, begin_row, end_row, k)
00854     #endif
00855         for (i = 0; i < m; ++i) {
00856             temp = 0.0;
00857             begin_row = ia[i];
00858             end_row = ia[i + 1];
00859             for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00860             value += y[i] * temp;
00861         }
00862     } else {
00863         for (i = 0; i < m; ++i) {
00864             temp = 0.0;
00865             begin_row = ia[i];
00866             end_row = ia[i + 1];
00867             for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
00868             value += y[i] * temp;
00869         }
00870     }
00871     return value;
00872 }
00873
00888 void fasp_blas_dcsr_mxm(const dCSRmat* A, const dCSRmat* B, dCSRmat* C)
00889 {
00890     INT i, j, k, l, count;
00891
00892     INT* JD = (INT*)fasp_mem_calloc(B->col, sizeof(INT));
00893
00894     C->row = A->row;
00895     C->col = B->col;
00896     C->val = NULL;
00897     C->JA = NULL;
00898     C->IA = (INT*)fasp_mem_calloc(C->row + 1, sizeof(INT));
00899
00900     for (i = 0; i < B->col; ++i) JD[i] = -1;
00901
00902     // step 1: Find first the structure IA of C
00903     for (i = 0; i < C->row; ++i) {
00904         count = 0;
00905
00906         for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
00907             for (j = B->IA[A->JA[k]]; j < B->IA[A->JA[k] + 1]; ++j) {
00908                 for (l = 0; l < count; ++l) {
00909                     if (JD[l] == B->JA[j]) break;
00910                 }
00911
00912                 if (l == count) {
00913                     JD[count] = B->JA[j];
00914                     count++;
00915                 }
00916             }
00917         }
00918         C->IA[i + 1] = count;
00919         for (j = 0; j < count; ++j) {
00920             JD[j] = -1;
00921         }
00922     }
00923
00924     for (i = 0; i < C->row; ++i) C->IA[i + 1] += C->IA[i];
00925
00926     // step 2: Find the structure JA of C

```

```

00927     INT countJD;
00928
00929     C->JA = (INT*) fasp_mem_calloc(C->IA[C->row], sizeof(INT));
00930
00931     for (i = 0; i < C->row; ++i) {
00932         countJD = 0;
00933         count = C->IA[i];
00934         for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
00935             for (j = B->IA[A->JA[k]]; j < B->IA[A->JA[k] + 1]; ++j) {
00936                 for (l = 0; l < countJD; l++) {
00937                     if (JD[l] == B->JA[j]) break;
00938                 }
00939
00940                 if (l == countJD) {
00941                     C->JA[count] = B->JA[j];
00942                     JD[countJD] = B->JA[j];
00943                     count++;
00944                     countJD++;
00945                 }
00946             }
00947         }
00948
00949         // for (j=0;j<countJD;++j) JD[j]=-1;
00950         fasp_iarray_set(countJD, JD, -1);
00951     }
00952
00953     fasp_mem_free(JD);
00954     JD = NULL;
00955
00956     // step 3: Find the structure A of C
00957     C->val = (REAL*) fasp_mem_calloc(C->IA[C->row], sizeof(REAL));
00958
00959     for (i = 0; i < C->row; ++i) {
00960         for (j = C->IA[i]; j < C->IA[i + 1]; ++j) {
00961             C->val[j] = 0;
00962             for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
00963                 for (l = B->IA[A->JA[k]]; l < B->IA[A->JA[k] + 1]; l++) {
00964                     if (B->JA[l] == C->JA[j]) {
00965                         C->val[j] += A->val[k] * B->val[l];
00966                     } // end if
00967                 } // end for l
00968             } // end for k
00969         } // end for j
00970     } // end for i
00971
00972     C->nnz = C->IA[C->row] - C->IA[0];
00973 }
00974
00994 void fasp_blas_dcsr_rap(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
00995                        dCSRmat* RAP)
00996 {
00997     const INT    n_coarse = R->row;
00998     const INT*   R_i      = R->IA;
00999     const INT*   R_j      = R->JA;
01000     const REAL*  R_data   = R->val;
01001
01002     const INT    n_fine = A->row;
01003     const INT*   A_i     = A->IA;
01004     const INT*   A_j     = A->JA;
01005     const REAL*  A_data  = A->val;
01006
01007     const INT*   P_i     = P->IA;
01008     const INT*   P_j     = P->JA;
01009     const REAL*  P_data  = P->val;
01010
01011     INT    RAP_size;
01012     INT*   RAP_i     = NULL;
01013     INT*   RAP_j     = NULL;
01014     REAL*  RAP_data  = NULL;
01015
01016 #ifdef _OPENMP
01017     INT* P_marker = NULL;
01018     INT* A_marker = NULL;
01019 #endif
01020
01021     INT* Ps_marker = NULL;
01022     INT* As_marker = NULL;
01023
01024     INT ic, il, i2, i3, jj1, jj2, jj3;
01025     INT jj_counter, jj_row_beginning;
01026     REAL r_entry, r_a_product, r_a_p_product;

```

```

01027
01028     INT nthreads = 1;
01029
01030 #ifdef _OPENMP
01031     INT myid, mybegin, myend, Ctemp;
01032     nthreads = fasp_get_num_threads();
01033 #endif
01034
01035     INT coarse_mul_nthreads = n_coarse * nthreads;
01036     INT fine_mul_nthreads   = n_fine * nthreads;
01037     INT coarse_add_nthreads = n_coarse + nthreads;
01038     INT minus_one_length    = coarse_mul_nthreads + fine_mul_nthreads;
01039     INT total_calloc        = minus_one_length + coarse_add_nthreads + nthreads;
01040
01041     Ps_marker = (INT*)fasp_mem_calloc(total_calloc, sizeof(INT));
01042     As_marker = Ps_marker + coarse_mul_nthreads;
01043
01044     /*-----*
01045  * First Pass: Determine size of RAP and set up RAP_i *
01046  *-----*/
01047     RAP_i = (INT*)fasp_mem_calloc(n_coarse + 1, sizeof(INT));
01048
01049     fasp_iarray_set(minus_one_length, Ps_marker, -1);
01050
01051 #ifdef _OPENMP
01052     INT* RAP_temp = As_marker + fine_mul_nthreads;
01053     INT* part_end = RAP_temp + coarse_add_nthreads;
01054
01055     if (n_coarse > OPENMP_HOLDS) {
01056 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
01057 jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2, \
01058 jj3, i3)
01059         for (myid = 0; myid < nthreads; myid++) {
01060             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
01061             P_marker = Ps_marker + myid * n_coarse;
01062             A_marker = As_marker + myid * n_fine;
01063             jj_counter = 0;
01064             for (ic = mybegin; ic < myend; ic++) {
01065                 P_marker[ic] = jj_counter;
01066                 jj_row_begining = jj_counter;
01067                 jj_counter++;
01068
01069                 for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01070                     i1 = R_j[jj1];
01071                     for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01072                         i2 = A_j[jj2];
01073                         if (A_marker[i2] != ic) {
01074                             A_marker[i2] = ic;
01075                             for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01076                                 i3 = P_j[jj3];
01077                                 if (P_marker[i3] < jj_row_begining) {
01078                                     P_marker[i3] = jj_counter;
01079                                     jj_counter++;
01080                                 }
01081                             }
01082                         }
01083                     }
01084                 }
01085
01086                 RAP_temp[ic + myid] = jj_row_begining;
01087             }
01088             RAP_temp[myend + myid] = jj_counter;
01089
01090             part_end[myid] = myend + myid + 1;
01091         }
01092         fasp_iarray_cp(part_end[0], RAP_temp, RAP_i);
01093         jj_counter = part_end[0];
01094         Ctemp = 0;
01095         for (i1 = 1; i1 < nthreads; i1++) {
01096             Ctemp += RAP_temp[part_end[i1] - 1] - 1;
01097             for (jj1 = part_end[i1] + 1; jj1 < part_end[i1+1]; jj1++) {
01098                 RAP_i[jj_counter] = RAP_temp[jj1] + Ctemp;
01099                 jj_counter++;
01100             }
01101         }
01102         RAP_size = RAP_i[n_coarse];
01103     }
01104     else {
01105 #endif
01106         jj_counter = 0;

```

```

01108     for (ic = 0; ic < n_coarse; ic++) {
01109         Ps_marker[ic] = jj_counter;
01110         jj_row_begining = jj_counter;
01111         jj_counter++;
01112
01113         for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01114             i1 = R_j[jj1];
01115
01116             for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01117                 i2 = A_j[jj2];
01118                 if (As_marker[i2] != ic) {
01119                     As_marker[i2] = ic;
01120                     for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01121                         i3 = P_j[jj3];
01122                         if (Ps_marker[i3] < jj_row_begining) {
01123                             Ps_marker[i3] = jj_counter;
01124                             jj_counter++;
01125                         }
01126                     }
01127                 }
01128             }
01129         }
01130
01131         RAP_i[ic] = jj_row_begining;
01132     }
01133
01134     RAP_i[n_coarse] = jj_counter;
01135     RAP_size = jj_counter;
01136 #ifdef _OPENMP
01137 }
01138 #endif
01139
01140     RAP_j = (INT*)fasp_mem_calloc(RAP_size, sizeof(INT));
01141     RAP_data = (REAL*)fasp_mem_calloc(RAP_size, sizeof(REAL));
01142
01143     fasp_iarray_set(minus_one_length, Ps_marker, -1);
01144
01145 #ifdef _OPENMP
01146     if (n_coarse > OPENMP_HOLDS) {
01147 #pragma omp parallel for private(myid, mybegin, myend, P_marker, A_marker, jj_counter, \
01148 ic, jj_row_begining, jj1, r_entry, i1, jj2, \
01149 r_a_product, i2, jj3, r_a_p_product, i3)
01150         for (myid = 0; myid < nthreads; myid++) {
01151             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
01152             P_marker = Ps_marker + myid * n_coarse;
01153             A_marker = As_marker + myid * n_fine;
01154             jj_counter = RAP_i[mybegin];
01155             for (ic = mybegin; ic < myend; ic++) {
01156                 P_marker[ic] = jj_counter;
01157                 jj_row_begining = jj_counter;
01158                 RAP_j[jj_counter] = ic;
01159                 RAP_data[jj_counter] = 0.0;
01160                 jj_counter++;
01161                 for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01162                     r_entry = R_data[jj1];
01163
01164                     i1 = R_j[jj1];
01165                     for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01166                         r_a_product = r_entry * A_data[jj2];
01167
01168                         i2 = A_j[jj2];
01169                         if (A_marker[i2] != ic) {
01170                             A_marker[i2] = ic;
01171                             for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01172                                 r_a_p_product = r_a_product * P_data[jj3];
01173
01174                                 i3 = P_j[jj3];
01175                                 if (P_marker[i3] < jj_row_begining) {
01176                                     P_marker[i3] = jj_counter;
01177                                     RAP_data[jj_counter] = r_a_p_product;
01178                                     RAP_j[jj_counter] = i3;
01179                                     jj_counter++;
01180                                 } else {
01181                                     RAP_data[P_marker[i3]] += r_a_p_product;
01182                                 }
01183                             }
01184                         } else {
01185                             for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01186                                 i3 = P_j[jj3];
01187                                 r_a_p_product = r_a_product * P_data[jj3];
01188                                 RAP_data[P_marker[i3]] += r_a_p_product;

```

```

01189     }
01190     }
01191     }
01192     }
01193     }
01194     }
01195     } else {
01196 #endif
01197         jj_counter = 0;
01198         for (ic = 0; ic < n_coarse; ic++) {
01199             Ps_marker[ic] = jj_counter;
01200             jj_row_begining = jj_counter;
01201             RAP_j[jj_counter] = ic;
01202             RAP_data[jj_counter] = 0.0;
01203             jj_counter++;
01204
01205             for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01206                 r_entry = R_data[jj1];
01207
01208                 i1 = R_j[jj1];
01209                 for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01210                     r_a_product = r_entry * A_data[jj2];
01211
01212                     i2 = A_j[jj2];
01213                     if (As_marker[i2] != ic) {
01214                         As_marker[i2] = ic;
01215                         for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01216                             r_a_p_product = r_a_product * P_data[jj3];
01217
01218                             i3 = P_j[jj3];
01219                             if (Ps_marker[i3] < jj_row_begining) {
01220                                 Ps_marker[i3] = jj_counter;
01221                                 RAP_data[jj_counter] = r_a_p_product;
01222                                 RAP_j[jj_counter] = i3;
01223                                 jj_counter++;
01224                             } else {
01225                                 RAP_data[Ps_marker[i3]] += r_a_p_product;
01226                             }
01227                         }
01228                     } else {
01229                         for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01230                             i3 = P_j[jj3];
01231                             r_a_p_product = r_a_product * P_data[jj3];
01232                             RAP_data[Ps_marker[i3]] += r_a_p_product;
01233                         }
01234                     }
01235                 }
01236             }
01237         }
01238 #ifdef _OPENMP
01239     }
01240 #endif
01241
01242     RAP->row = n_coarse;
01243     RAP->col = n_coarse;
01244     RAP->nnz = RAP_size;
01245     RAP->IA = RAP_i;
01246     RAP->JA = RAP_j;
01247     RAP->val = RAP_data;
01248
01249     fasp_mem_free(Ps_marker);
01250     Ps_marker = NULL;
01251 }
01252
01269 void fasp_blas_dcsr_rap_agg(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
01270                             dCSRmat* RAP)
01271 {
01272     const INT n_coarse = R->row;
01273     const INT* R_i = R->IA;
01274     const INT* R_j = R->JA;
01275
01276     const INT n_fine = A->row;
01277     const INT* A_i = A->IA;
01278     const INT* A_j = A->JA;
01279     const REAL* A_data = A->val;
01280
01281     const INT* P_i = P->IA;
01282     const INT* P_j = P->JA;
01283
01284     INT RAP_size;
01285     INT* RAP_i = NULL;

```

```

01286     INT* RAP_j      = NULL;
01287     REAL* RAP_data = NULL;
01288
01289 #ifdef _OPENMP
01290     INT* P_marker = NULL;
01291     INT* A_marker = NULL;
01292 #endif
01293
01294     INT* Ps_marker = NULL;
01295     INT* As_marker = NULL;
01296
01297     INT ic, i1, i2, i3, jj1, jj2, jj3;
01298     INT jj_counter, jj_row_begining;
01299
01300     INT nthreads = 1;
01301
01302 #ifdef _OPENMP
01303     INT myid, mybegin, myend, Ctemp;
01304     nthreads = fasp_get_num_threads();
01305 #endif
01306
01307     INT coarse_mul_nthreads = n_coarse * nthreads;
01308     INT fine_mul_nthreads   = n_fine * nthreads;
01309     INT coarse_add_nthreads = n_coarse + nthreads;
01310     INT minus_one_length    = coarse_mul_nthreads + fine_mul_nthreads;
01311     INT total_calloc        = minus_one_length + coarse_add_nthreads + nthreads;
01312
01313     Ps_marker = (INT*)fasp_mem_calloc(total_calloc, sizeof(INT));
01314     As_marker = Ps_marker + coarse_mul_nthreads;
01315
01316     /*-----*
01317  * First Pass: Determine size of RAP and set up RAP_i *
01318  *-----*/
01319     RAP_i = (INT*)fasp_mem_calloc(n_coarse + 1, sizeof(INT));
01320
01321     fasp_iarray_set(minus_one_length, Ps_marker, -1);
01322
01323 #ifdef _OPENMP
01324     INT* RAP_temp = As_marker + fine_mul_nthreads;
01325     INT* part_end = RAP_temp + coarse_add_nthreads;
01326
01327     if (n_coarse > OPENMP_HOLDS) {
01328 #pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
01329 jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2, \
01330 jj3, i3)
01331         for (myid = 0; myid < nthreads; myid++) {
01332             fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
01333             P_marker = Ps_marker + myid * n_coarse;
01334             A_marker = As_marker + myid * n_fine;
01335             jj_counter = 0;
01336             for (ic = mybegin; ic < myend; ic++) {
01337                 P_marker[ic] = jj_counter;
01338                 jj_row_begining = jj_counter;
01339                 jj_counter++;
01340
01341                 for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01342                     i1 = R_j[jj1];
01343                     for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01344                         i2 = A_j[jj2];
01345                         if (A_marker[i2] != ic) {
01346                             A_marker[i2] = ic;
01347                             for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01348                                 i3 = P_j[jj3];
01349                                 if (P_marker[i3] < jj_row_begining) {
01350                                     P_marker[i3] = jj_counter;
01351                                     jj_counter++;
01352                                 }
01353                             }
01354                         }
01355                     }
01356                 }
01357                 RAP_temp[ic + myid] = jj_row_begining;
01358             }
01359             RAP_temp[myend + myid] = jj_counter;
01360
01361             part_end[myid] = myend + myid + 1;
01362         }
01363     }
01364     fasp_iarray_cp(part_end[0], RAP_temp, RAP_i);
01365     jj_counter = part_end[0];
01366     Ctemp      = 0;

```

```

01367         for (i1 = 1; i1 < nthreads; i1++) {
01368             Ctemp += RAP_temp[part_end[i1] - 1];
01369             for (jj1 = part_end[i1] + 1; jj1 < part_end[i1]; jj1++) {
01370                 RAP_i[jj_counter] = RAP_temp[jj1] + Ctemp;
01371                 jj_counter++;
01372             }
01373         }
01374         RAP_size = RAP_i[n_coarse];
01375     }
01376     else {
01377         #endif
01378         jj_counter = 0;
01379         for (ic = 0; ic < n_coarse; ic++) {
01380             Ps_marker[ic] = jj_counter;
01381             jj_row_begining = jj_counter;
01382             jj_counter++;
01383             for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01384                 i1 = R_j[jj1];
01385                 for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01386                     i2 = A_j[jj2];
01387                     if (As_marker[i2] != ic) {
01388                         As_marker[i2] = ic;
01389                         for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01390                             i3 = P_j[jj3];
01391                             if (Ps_marker[i3] < jj_row_begining) {
01392                                 Ps_marker[i3] = jj_counter;
01393                                 jj_counter++;
01394                             }
01395                         }
01396                     }
01397                 }
01398             }
01399         }
01400     }
01401     RAP_i[ic] = jj_row_begining;
01402 }
01403 RAP_i[n_coarse] = jj_counter;
01404 RAP_size = jj_counter;
01405 #ifdef _OPENMP
01406 }
01407 #endif
01408 RAP_j = (INT*)fasp_mem_calloc(RAP_size, sizeof(INT));
01409 RAP_data = (REAL*)fasp_mem_calloc(RAP_size, sizeof(REAL));
01410 fasp_iarray_set(minus_one_length, Ps_marker, -1);
01411 #ifdef _OPENMP
01412 if (n_coarse > OPENMP_HOLDS) {
01413     #pragma omp parallel for private(myid, mybegin, myend, P_marker, A_marker, jj_counter, \
01414     ic, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
01415     for (myid = 0; myid < nthreads; myid++) {
01416         fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
01417         P_marker = Ps_marker + myid * n_coarse;
01418         A_marker = As_marker + myid * n_fine;
01419         jj_counter = RAP_i[mybegin];
01420         for (ic = mybegin; ic < myend; ic++) {
01421             P_marker[ic] = jj_counter;
01422             jj_row_begining = jj_counter;
01423             RAP_j[jj_counter] = ic;
01424             RAP_data[jj_counter] = 0.0;
01425             jj_counter++;
01426             for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01427                 i1 = R_j[jj1];
01428                 for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01429                     i2 = A_j[jj2];
01430                     if (A_marker[i2] != ic) {
01431                         A_marker[i2] = ic;
01432                         for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01433                             i3 = P_j[jj3];
01434                             if (P_marker[i3] < jj_row_begining) {
01435                                 P_marker[i3] = jj_counter;
01436                                 RAP_data[jj_counter] = A_data[jj2];
01437                                 RAP_j[jj_counter] = i3;
01438                                 jj_counter++;
01439                             }
01440                         }
01441                     }
01442                 }
01443             }
01444         }
01445     }
01446 }

```

```

01448             } else {
01449                 RAP_data[P_marker[i3]] += A_data[jj2];
01450             }
01451         }
01452     } else {
01453         for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01454             i3 = P_j[jj3];
01455             RAP_data[P_marker[i3]] += A_data[jj2];
01456         }
01457     }
01458 }
01459 }
01460 }
01461 }
01462 } else {
01463 #endif
01464     jj_counter = 0;
01465     for (ic = 0; ic < n_coarse; ic++) {
01466         Ps_marker[ic] = jj_counter;
01467         jj_row_begining = jj_counter;
01468         RAP_j[jj_counter] = ic;
01469         RAP_data[jj_counter] = 0.0;
01470         jj_counter++;
01471
01472         for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
01473             i1 = R_j[jj1];
01474             for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
01475                 i2 = A_j[jj2];
01476                 if (As_marker[i2] != ic) {
01477                     As_marker[i2] = ic;
01478                     for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01479                         i3 = P_j[jj3];
01480                         if (Ps_marker[i3] < jj_row_begining) {
01481                             Ps_marker[i3] = jj_counter;
01482                             RAP_data[jj_counter] = A_data[jj2];
01483                             RAP_j[jj_counter] = i3;
01484                             jj_counter++;
01485                         } else {
01486                             RAP_data[Ps_marker[i3]] += A_data[jj2];
01487                         }
01488                     }
01489                 } else {
01490                     for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
01491                         i3 = P_j[jj3];
01492                         RAP_data[Ps_marker[i3]] += A_data[jj2];
01493                     }
01494                 }
01495             }
01496         }
01497     }
01498 #ifdef _OPENMP
01499 }
01500 #endif
01501
01502 RAP->row = n_coarse;
01503 RAP->col = n_coarse;
01504 RAP->nnz = RAP_size;
01505 RAP->IA = RAP_i;
01506 RAP->JA = RAP_j;
01507 RAP->val = RAP_data;
01508
01509 fasp_mem_free(Ps_marker);
01510 Ps_marker = NULL;
01511 }
01512
01530 void fasp_blas_dcsr_rap_aggl(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
01531                             dCSRmat* B)
01532 {
01533     const INT row = R->row, col = P->col;
01534     const INT * ir = R->IA, *ia = A->IA, *ip = P->IA;
01535     const INT * jr = R->JA, *ja = A->JA, *jp = P->JA;
01536     const REAL* aj = A->val;
01537
01538     INT * iac, *jac;
01539     REAL* acj;
01540
01541     INT* index = (INT*)fasp_mem_malloc(A->col, sizeof(INT));
01542     INT* iindex = (INT*)fasp_mem_malloc(col, sizeof(INT));
01543
01544     INT nB = A->nnz;
01545     INT i, i1, j, jj, k, length;

```



```

01546     INT begin_row, end_row, begin_rowA, end_rowA, begin_rowR, end_rowR;
01547     INT istart, iistart, count;
01548
01549     // for (i=0; i<A->col; ++i) index[i] = -2;
01550     fasp_iarray_set(A->col, index, -2);
01551
01552     // memcpy(iindex,index,col*sizeof(INT));
01553     fasp_iarray_cp(col, index, iindex);
01554
01555     jac = (INT*)fasp_mem_calloc(nB, sizeof(INT));
01556
01557     iac = (INT*)fasp_mem_calloc(row + 1, sizeof(INT));
01558
01559     REAL* temp = (REAL*)fasp_mem_calloc(A->col, sizeof(REAL));
01560
01561     iac[0] = 0;
01562
01563     // First loop: form sparsity pattern of R*A*P
01564     for (i = 0; i < row; ++i) {
01565         // reset istart and length at the beginning of each loop
01566         istart = -1;
01567         length = 0;
01568         i1 = i + 1;
01569
01570         // go across the rows in R
01571         begin_rowR = ir[i];
01572         end_rowR = ir[i1];
01573         for (jj = begin_rowR; jj < end_rowR; ++jj) {
01574             j = jr[jj];
01575             // for each column in A
01576             begin_rowA = ia[j];
01577             end_rowA = ia[j + 1];
01578             for (k = begin_rowA; k < end_rowA; ++k) {
01579                 if (index[ja[k]] == -2) {
01580                     index[ja[k]] = istart;
01581                     istart = ja[k];
01582                     ++length;
01583                 }
01584             }
01585         }
01586
01587         // book-keeping [reseting length and setting iistart]
01588         count = length;
01589         iistart = -1;
01590         length = 0;
01591
01592         // use each column that would have resulted from R*A
01593         for (j = 0; j < count; ++j) {
01594             jj = istart;
01595             istart = index[istart];
01596             index[jj] = -2;
01597
01598             // go across the row of P
01599             begin_row = ip[jj];
01600             end_row = ip[jj + 1];
01601             for (k = begin_row; k < end_row; ++k) {
01602                 // pull out the appropriate columns of P
01603                 if (iindex[jp[k]] == -2) {
01604                     iindex[jp[k]] = iistart;
01605                     iistart = jp[k];
01606                     ++length;
01607                 }
01608             } // end for k
01609         } // end for j
01610
01611         // set B->IA
01612         iac[i1] = iac[i] + length;
01613
01614         if (iac[i1] > nB) { // Memory not enough!!!
01615             nB = nB * 2;
01616             jac = (INT*)fasp_mem_realloc(jac, nB * sizeof(INT));
01617         }
01618
01619         // put the correct columns of p into the column list of the products
01620         begin_row = iac[i];
01621         end_row = iac[i1];
01622         for (j = begin_row; j < end_row; ++j) {
01623             // put the value in B->JA
01624             jac[j] = iistart;
01625             // set istart to the next value
01626             iistart = iindex[iistart];

```

```

01627         // set the iindex spot to 0
01628         iindex[jac[j]] = -2;
01629     } // end j
01630
01631 } // end i: First loop
01632
01633 jac = (INT*)fasp_mem_realloc(jac, (iac[row]) * sizeof(INT));
01634
01635 acj = (REAL*)fasp_mem_calloc(iac[row], sizeof(REAL));
01636
01637 INT* BTindex = (INT*)fasp_mem_calloc(col, sizeof(INT));
01638
01639 // Second loop: compute entries of R*A*P
01640 for (i = 0; i < row; ++i) {
01641     il = i + 1;
01642
01643     // each col of B
01644     begin_row = iac[i];
01645     end_row = iac[il];
01646     for (j = begin_row; j < end_row; ++j) {
01647         BTindex[jac[j]] = j;
01648     }
01649
01650     // reset istart and length at the beginning of each loop
01651     istart = -1;
01652     length = 0;
01653
01654     // go across the rows in R
01655     begin_rowR = ir[i];
01656     end_rowR = ir[il];
01657     for (jj = begin_rowR; jj < end_rowR; ++jj) {
01658         j = jr[jj];
01659
01660         // for each column in A
01661         begin_rowA = ia[j];
01662         end_rowA = ia[j + 1];
01663         for (k = begin_rowA; k < end_rowA; ++k) {
01664             if (index[ja[k]] == -2) {
01665                 index[ja[k]] = istart;
01666                 istart = ja[k];
01667                 ++length;
01668             }
01669             temp[ja[k]] += aj[k];
01670         }
01671     }
01672
01673     // book-keeping [resetting length and setting iistart]
01674     // use each column that would have resulted from R*A
01675     for (j = 0; j < length; ++j) {
01676         jj = istart;
01677         istart = index[istart];
01678         index[jj] = -2;
01679
01680         // go across the row of P
01681         begin_row = ip[jj];
01682         end_row = ip[jj + 1];
01683         for (k = begin_row; k < end_row; ++k) {
01684             // pull out the appropriate columns of P
01685             acj[BTindex[jp[k]]] += temp[jj];
01686         }
01687         temp[jj] = 0.0;
01688     }
01689 } // end for i: Second loop
01690
01691 // setup coarse matrix B
01692 B->row = row;
01693 B->col = col;
01694 B->IA = iac;
01695 B->JA = jac;
01696 B->val = acj;
01697 B->nnz = B->IA[B->row] - B->IA[0];
01698
01699 fasp_mem_free(temp);
01700 temp = NULL;
01701 fasp_mem_free(index);
01702 index = NULL;
01703 fasp_mem_free(iindex);
01704 iindex = NULL;
01705 fasp_mem_free(BTindex);
01706 BTindex = NULL;

```

```

01708 }
01709
01734 void fasp_blas_dcsr_ptap(const dCSRmat* Pt, const dCSRmat* A, const dCSRmat* P,
01735                          dCSRmat* Ac)
01736 {
01737     const INT nc = Pt->row, n = Pt->col, nnzP = P->nnz, nnzA = A->nnz;
01738     INT i, maxrpout;
01739
01740     // shift A from usual to ltz format
01741 #ifdef _OPENMP
01742 #pragma omp parallel for if (n > OPENMP_HOLDS)
01743 #endif
01744     for (i = 0; i <= n; ++i) {
01745         A->IA[i]++;
01746         P->IA[i]++;
01747     }
01748
01749 #ifdef _OPENMP
01750 #pragma omp parallel for if (nnzA > OPENMP_HOLDS)
01751 #endif
01752     for (i = 0; i < nnzA; ++i) {
01753         A->JA[i]++;
01754     }
01755
01756 #ifdef _OPENMP
01757 #pragma omp parallel for if (nc > OPENMP_HOLDS)
01758 #endif
01759     for (i = 0; i <= nc; ++i) {
01760         Pt->IA[i]++;
01761     }
01762
01763 #ifdef _OPENMP
01764 #pragma omp parallel for if (nnzP > OPENMP_HOLDS)
01765 #endif
01766     for (i = 0; i < nnzP; ++i) {
01767         P->JA[i]++;
01768         Pt->JA[i]++;
01769     }
01770
01771     // compute P' A P
01772     dCSRmat PtAP =
01773         fasp_blas_dcsr_rap2(Pt->IA, Pt->JA, Pt->val, A->IA, A->JA, A->val, Pt->IA,
01774                             Pt->JA, Pt->val, n, nc, &maxrpout, P->IA, P->JA);
01775
01776     Ac->row = PtAP.row;
01777     Ac->col = PtAP.col;
01778     Ac->nnz = PtAP.nnz;
01779     Ac->IA = PtAP.IA;
01780     Ac->JA = PtAP.JA;
01781     Ac->val = PtAP.val;
01782
01783     // shift A back from ltz format
01784 #ifdef _OPENMP
01785 #pragma omp parallel for if (Ac->row > OPENMP_HOLDS)
01786 #endif
01787     for (i = 0; i <= Ac->row; ++i) Ac->IA[i]--;
01788
01789 #ifdef _OPENMP
01790 #pragma omp parallel for if (Ac->nnz > OPENMP_HOLDS)
01791 #endif
01792     for (i = 0; i < Ac->nnz; ++i) Ac->JA[i]--;
01793
01794 #ifdef _OPENMP
01795 #pragma omp parallel for if (n > OPENMP_HOLDS)
01796 #endif
01797     for (i = 0; i <= n; ++i) A->IA[i]--;
01798
01799 #ifdef _OPENMP
01800 #pragma omp parallel for if (nnzA > OPENMP_HOLDS)
01801 #endif
01802     for (i = 0; i < nnzA; ++i) A->JA[i]--;
01803
01804 #ifdef _OPENMP
01805 #pragma omp parallel for if (n > OPENMP_HOLDS)
01806 #endif
01807     for (i = 0; i <= n; ++i) P->IA[i]--;
01808
01809 #ifdef _OPENMP
01810 #pragma omp parallel for if (nnzP > OPENMP_HOLDS)
01811 #endif
01812     for (i = 0; i < nnzP; ++i) P->JA[i]--;

```

```

01813
01814 #ifdef _OPENMP
01815 #pragma omp parallel for if (nc > OPENMP_HOLDS)
01816 #endif
01817     for (i = 0; i <= nc; ++i) Pt->IA[i]--;
01818
01819 #ifdef _OPENMP
01820 #pragma omp parallel for if (nnzP > OPENMP_HOLDS)
01821 #endif
01822     for (i = 0; i < nnzP; ++i) Pt->JA[i]--;
01823
01824     return;
01825 }
01826
01842 dCSRmat fasp_blas_dcsr_rap2(INT* ir, INT* jr, REAL* r, INT* ia, INT* ja, REAL* a,
01843                             INT* ipt, INT* jpt, REAL* pt, INT n, INT nc, INT* maxrpout,
01844                             INT* ipin, INT* jpin)
01845 {
01846     dCSRmat ac;
01847     INT    n1, ncl, nnzp, maxrp;
01848     INT *   ip = NULL, *jp = NULL;
01849
01850     /*
01851     if ipin is null, this
01852     means that we need to do the transpose of p here; otherwise,
01853     these are considered to be input
01854     */
01855     maxrp = 0;
01856     nnzp = ipt[nc] - 1;
01857     n1 = n + 1;
01858
01859     if (!ipin) {
01860         ip = (INT*)calloc(n1, sizeof(INT));
01861         jp = (INT*)calloc(nnzp, sizeof(INT));
01862         /* these must be null anyway, so no need to assign null
01863         ipin=NULL;
01864         jpin=NULL;
01865         */
01866     } else {
01867         ip = ipin;
01868         jp = jpin;
01869     }
01870
01871     fasp_sparse_iit_(ipt, jpt, &nc, &n, ip, jp);
01872
01873     /* triple matrix product: R * A * transpose(P^T)=R*A*P.*/
01874     /* A is square n by n*/
01875     /* Note: to compute R*A* P the input are R, A and P^T */
01876     /* we need to transpose now the structure of P, because the input is P^T */
01877     /* end of transpose of the boolean corresponding to P */
01878     /* ic are the addresses of the rows of the output */
01879     ncl = nc + 1;
01880     ac.IA = (INT*)calloc(ncl, sizeof(INT));
01881
01882     /*
01883     First call is with jc=null so that we find the number of
01884     nonzeros in the result
01885     */
01886     ac.JA = NULL;
01887     fasp_sparse_rapms_(ir, jr, ia, ja, ip, jp, &n, &nc, ac.IA, ac.JA, &maxrp);
01888     ac.nnz = ac.IA[nc] - 1;
01889     ac.JA = (INT*)calloc(ac.nnz, sizeof(INT));
01890
01891     /*
01892     second call is to fill the column indexes array jc.
01893     */
01894     fasp_sparse_rapms_(ir, jr, ia, ja, ip, jp, &n, &nc, ac.IA, ac.JA, &maxrp);
01895     if (!ipin) {
01896         if (ip) free(ip);
01897         if (jp) free(jp);
01898     }
01899     ac.val = (REAL*)calloc(ac.nnz, sizeof(REAL));
01900     /* this is the compute with the entries */
01901     fasp_sparse_rapcmp_(ir, jr, r, ia, ja, a, ipt, jpt, pt, &n, &nc, ac.IA, ac.JA,
01902                        ac.val, &maxrp);
01903     ac.row = nc;
01904     ac.col = nc;
01905
01906     /*=====*/
01907     *maxrpout = maxrp;
01908

```

```

01909     return ac;
01910 }
01911
01930 void fasp_blas_dcsr_rap4(dCSRmat* R, dCSRmat* A, dCSRmat* P, dCSRmat* B, INT* icor_ysk)
01931 {
01932     SHORT nthreads = 1, use_omp = FALSE;
01933
01934     #ifdef _OPENMP
01935     if (R->row > OPENMP_HOLDS) {
01936         use_omp = TRUE;
01937         nthreads = fasp_get_num_threads();
01938     }
01939     #endif
01940
01941     if (use_omp) {
01942         const INT row = R->row, col = P->col;
01943         INT * ir = R->IA, *ia = A->IA, *ip = P->IA;
01944         INT * jr = R->JA, *ja = A->JA, *jp = P->JA;
01945         REAL * rj = R->val, *aj = A->val, *pj = P->val;
01946         INT istart, iistart;
01947         INT end_row, end_rowA, end_rowR;
01948         INT i, j, jj, k, length, myid, mybegin, myend;
01949         INT jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2, jj3, i3;
01950         INT* index = NULL;
01951         INT* iindex = NULL;
01952         INT* BTindex = NULL;
01953         REAL* temp = NULL;
01954         INT FiveMyid, min_A, min_P, A_pos, P_pos, FiveIc;
01955         INT minus_one_length_A = icor_ysk[5 * nthreads];
01956         INT minus_one_length_P = icor_ysk[5 * nthreads + 1];
01957         INT minus_one_length = minus_one_length_A + minus_one_length_P;
01958
01959         INT* iindexes =
01960             (INT*)fasp_mem_calloc(minus_one_length + minus_one_length_P, sizeof(INT));
01961
01962         #if DEBUG_MODE > 1
01963             total_alloc_mem += minus_one_length * sizeof(INT);
01964         #endif
01965         INT* indexes = iindexes + minus_one_length_P;
01966         INT* BTindexes = indexes + minus_one_length_A;
01967
01968         INT* iac = (INT*)fasp_mem_calloc(row + 1, sizeof(INT));
01969
01970         #if DEBUG_MODE > 1
01971             total_alloc_mem += (row + 1) * sizeof(INT);
01972         #endif
01973
01974         INT* part_end = (INT*)fasp_mem_calloc(2 * nthreads + row, sizeof(INT));
01975
01976         #if DEBUG_MODE > 1
01977             total_alloc_mem += (2 * nthreads + row) * sizeof(INT);
01978         #endif
01979
01980         INT* iac_temp = part_end + nthreads;
01981         INT** iindex_array = (INT**)fasp_mem_calloc(nthreads, sizeof(INT*));
01982         INT** index_array = (INT**)fasp_mem_calloc(nthreads, sizeof(INT*));
01983
01984         fasp_iarray_set(minus_one_length, iindexes, -2);
01985
01986         #ifdef _OPENMP
01987         #pragma omp parallel for private(myid, FiveMyid, mybegin, myend, min_A, min_P, index, \
01988             iindex, A_pos, P_pos, ic, FiveIc, jj_counter, \
01989             jj_row_begining, end_rowR, jj1, i1, end_rowA, jj2, \
01990             i2, end_row, jj3, i3)
01991         #endif
01992         for (myid = 0; myid < nthreads; myid++) {
01993             FiveMyid = myid * 5;
01994             mybegin = icor_ysk[FiveMyid];
01995             if (myid == nthreads - 1) {
01996                 myend = row;
01997             } else {
01998                 myend = icor_ysk[FiveMyid + 5];
01999             }
02000             min_A = icor_ysk[FiveMyid + 2];
02001             min_P = icor_ysk[FiveMyid + 4];
02002             A_pos = 0;
02003             P_pos = 0;
02004             for (ic = myid - 1; ic >= 0; ic--) {
02005                 FiveIc = ic * 5;
02006                 A_pos += icor_ysk[FiveIc + 1];
02007                 P_pos += icor_ysk[FiveIc + 3];

```

```

02008     }
02009     iindex_array[myid] = iindex = iindexs + P_pos - min_P;
02010     index_array[myid] = index = indexs + A_pos - min_A;
02011     jj_counter = 0;
02012     for (ic = mybegin; ic < myend; ic++) {
02013         iindex[ic] = jj_counter;
02014         jj_row_begining = jj_counter;
02015         jj_counter++;
02016         end_rowR = ir[ic + 1];
02017         for (jj1 = ir[ic]; jj1 < end_rowR; jj1++) {
02018             il = jr[jj1];
02019             end_rowA = ia[il + 1];
02020             for (jj2 = ia[il]; jj2 < end_rowA; jj2++) {
02021                 i2 = ja[jj2];
02022                 if (index[i2] != ic) {
02023                     index[i2] = ic;
02024                     end_row = ip[i2 + 1];
02025                     for (jj3 = ip[i2]; jj3 < end_row; jj3++) {
02026                         i3 = jp[jj3];
02027                         if (iindex[i3] < jj_row_begining) {
02028                             iindex[i3] = jj_counter;
02029                             jj_counter++;
02030                         }
02031                     }
02032                 }
02033             }
02034         }
02035         iac_temp[ic + myid] = jj_row_begining;
02036     }
02037     iac_temp[myend + myid] = jj_counter;
02038     part_end[myid] = myend + myid + 1;
02039 }
02040 fasp_iarray_cp(part_end[0], iac_temp, iac);
02041 jj_counter = part_end[0];
02042 INT Ctemp = 0;
02043 for (il = 1; il < nthreads; il++) {
02044     Ctemp += iac_temp[part_end[il - 1] - 1];
02045     for (jj1 = part_end[il - 1] + 1; jj1 < part_end[il]; jj1++) {
02046         iac[jj_counter] = iac_temp[jj1] + Ctemp;
02047         jj_counter++;
02048     }
02049 }
02050 INT* jac = (INT*)fasp_mem_calloc(iac[row], sizeof(INT));
02051 #if DEBUG_MODE > 1
02052     total_alloc_mem += iac[row] * sizeof(INT);
02053 #endif
02054     fasp_iarray_set(minus_one_length, iindexs, -2);
02055 #ifdef _OPENMP
02056 #pragma omp parallel for private(myid, index, iindex, FiveMyid, mybegin, myend, i, \
02057 istart, length, il, end_rowR, jj, j, end_rowA, k, \
02058 iistart, end_row)
02059 #endif
02060     for (myid = 0; myid < nthreads; myid++) {
02061         iindex = iindex_array[myid];
02062         index = index_array[myid];
02063         FiveMyid = myid * 5;
02064         mybegin = icor_ysk[FiveMyid];
02065         if (myid == nthreads - 1) {
02066             myend = row;
02067         } else {
02068             myend = icor_ysk[FiveMyid + 5];
02069         }
02070         for (i = mybegin; i < myend; ++i) {
02071             istart = -1;
02072             length = 0;
02073             il = i + 1;
02074             // go across the rows in R
02075             end_rowR = ir[il];
02076             for (jj = ir[i]; jj < end_rowR; ++jj) {
02077                 j = jr[jj];
02078                 // for each column in A
02079                 end_rowA = ia[j + 1];
02080                 for (k = ia[j]; k < end_rowA; ++k) {
02081                     if (index[ja[k]] == -2) {
02082                         index[ja[k]] = istart;
02083                         istart = ja[k];
02084                         ++length;
02085                     }
02086                 }
02087             }
02088             // book-keeping [resetting length and setting iistart]

```

```

02089         // count = length;
02090         iistart = -1;
02091         // length = 0;
02092         // use each column that would have resulted from R*A
02093         // for (j = 0; j < count; ++j) {
02094         for (j = 0; j < length; ++j) {
02095             jj = iistart;
02096             iistart = index[iistart];
02097             index[jj] = -2;
02098             // go across the row of P
02099             end_row = ip[jj + 1];
02100             for (k = ip[jj]; k < end_row; ++k) {
02101                 // pull out the appropriate columns of P
02102                 if (iindex[jp[k]] == -2) {
02103                     iindex[jp[k]] = iistart;
02104                     iistart = jp[k];
02105                     //++length;
02106                 }
02107             } // end for k
02108         } // end for j
02109         // put the correct columns of p into the column list of the products
02110         end_row = iac[i1];
02111         for (j = iac[i1]; j < end_row; ++j) {
02112             // put the value in B->JA
02113             jac[j] = iistart;
02114             // set iistart to the next value
02115             iistart = iindex[iistart];
02116             // set the iindex spot to 0
02117             iindex[jac[j]] = -2;
02118         } // end j
02119     }
02120 }
02121 // Third loop: compute entries of R*A*P
02122 REAL* acj = (REAL*)fasp_mem_malloc(iac[row], sizeof(REAL));
02123 #if DEBUG_MODE > 1
02124     total_alloc_mem += iac[row] * sizeof(REAL);
02125 #endif
02126 REAL* temps = (REAL*)fasp_mem_malloc(minus_one_length_A, sizeof(REAL));
02127 #if DEBUG_MODE > 1
02128     total_alloc_mem += minus_one_length_A * sizeof(REAL);
02129 #endif
02130
02131 #ifdef _OPENMP
02132 #pragma omp parallel for private(
02133     myid, index, FiveMyid, mybegin, myend, min_A, min_P, A_pos, P_pos, ic, FiveIc, \
02134     BTindex, temp, i, i1, end_row, j, iistart, length, end_rowR, jj, end_rowA, k)
02135 #endif
02136     for (myid = 0; myid < nthreads; myid++) {
02137         index = index_array[myid];
02138         FiveMyid = myid * 5;
02139         mybegin = icor_ysk[FiveMyid];
02140         if (myid == nthreads - 1) {
02141             myend = row;
02142         } else {
02143             myend = icor_ysk[FiveMyid + 5];
02144         }
02145         min_A = icor_ysk[FiveMyid + 2];
02146         min_P = icor_ysk[FiveMyid + 4];
02147         A_pos = 0;
02148         P_pos = 0;
02149         for (ic = myid - 1; ic >= 0; ic--) {
02150             FiveIc = ic * 5;
02151             A_pos += icor_ysk[FiveIc + 1];
02152             P_pos += icor_ysk[FiveIc + 3];
02153         }
02154         BTindex = BTindexs + P_pos - min_P;
02155         temp = temps + A_pos - min_A;
02156         for (i = mybegin; i < myend; ++i) {
02157             i1 = i + 1;
02158             // each col of B
02159             end_row = iac[i1];
02160             for (j = iac[i1]; j < end_row; ++j) {
02161                 BTindex[jac[j]] = j;
02162             }
02163             // reset iistart and length at the beginning of each loop
02164             iistart = -1;
02165             length = 0;
02166             // go across the rows in R
02167             end_rowR = ir[i1];
02168             for (jj = ir[i1]; jj < end_rowR; ++jj) {
02169                 j = jr[jj];

```

```

02170         // for each column in A
02171         end_rowA = ia[j + 1];
02172         for (k = ia[j]; k < end_rowA; ++k) {
02173             if (index[ja[k]] == -2) {
02174                 index[ja[k]] = istart;
02175                 istart
02176                     = ja[k];
02177                 ++length;
02178             }
02179             temp[ja[k]] += rj[jj] * aj[k];
02180         }
02181         // book-keeping [resetting length and setting iistart]
02182         // use each column that would have resulted from R*A
02183         for (j = 0; j < length; ++j) {
02184             jj
02185                 = istart;
02186             istart
02187                 = index[istart];
02188             index[jj] = -2;
02189             // go across the row of P
02190             end_row = ip[jj + 1];
02191             for (k = ip[jj]; k < end_row; ++k) {
02192                 // pull out the appropriate columns of P
02193                 acj[BIndex[jp[k]]] += temp[jj] * pj[k];
02194             }
02195             temp[jj] = 0.0;
02196         }
02197         // setup coarse matrix B
02198         B->row = row;
02199         B->col = col;
02200         B->IA = iac;
02201         B->JA = jac;
02202         B->val = acj;
02203         B->nnz = B->IA[B->row] - B->IA[0];
02204
02205         fasp_mem_free(temps);
02206         temps = NULL;
02207         fasp_mem_free(iindex);
02208         iindex = NULL;
02209         fasp_mem_free(part_end);
02210         part_end = NULL;
02211         fasp_mem_free(iindex_array);
02212         iindex_array = NULL;
02213         fasp_mem_free(index_array);
02214         index_array = NULL;
02215     } else {
02216         fasp_blas_dcsr_rap(R, A, P, B);
02217     }
02218 }
02219
02220 /*-----*/
02221 /*--      End of File      --*/
02222 /*-----*/

```

9.93 BlaSpmvCSRL.c File Reference

Linear algebraic operations for [dCSRLmat](#) matrices.

```
#include "fasp.h"
```

Functions

- void [fasp_blas_dcsr_l_mnv](#) (const [dCSRLmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
Compute $y = A*x$ for a sparse matrix in CSRL format.

9.93.1 Detailed Description

Linear algebraic operations for [dCSRLmat](#) matrices.

Note

This file contains Level-1 (Bla) functions.

Reference: John Mellor-Crummey and John Garvin Optimizaing sparse matrix vector product computations using unroll and jam, Tech Report Rice Univ, Aug 2002.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSpmvCSRL.c](#).

9.93.2 Function Documentation**9.93.2.1 fasp_blas_dcsrl_mxv()**

```
void fasp_blas_dcsrl_mxv (
    const dCSRLmat * A,
    const REAL * x,
    REAL * y )
```

Compute $y = A \cdot x$ for a sparse matrix in CSRL format.

Parameters

<i>A</i>	Pointer to dCSRLmat matrix A
<i>x</i>	Pointer to REAL array of vector x
<i>y</i>	Pointer to REAL array of vector y

Author

Zhiyang Zhou, Chensong Zhang

Date

2011/01/07

Definition at line 36 of file [BlaSpmvCSRL.c](#).

9.94 BlaSpmvCSRL.c

[Go to the documentation of this file.](#)

```
00001
00018 #include "fasp.h"
00019
00020 /*-----*/
00021 /*--      Public Functions      --*/
00022 /*-----*/
00023
00036 void fasp_blas_dcsrl_mxv (const dCSRLmat *A,
00037                          const REAL *x,
00038                          REAL *y)
00039 {
00040     const INT    dif      = A -> dif;
00041     const INT    *nz_diff = A -> nz_diff;
00042     const INT    *rowindex = A -> index;
00043     const INT    *rowstart = A -> start;
00044     const INT    *ja       = A -> ja;
00045     const REAL    *a        = A -> val;
00046
```

```

00047     INT i;
00048     INT row, col=0;
00049     INT len, rowlen;
00050     INT firstrow, lastrow;
00051
00052     REAL val0, val1;
00053
00054     for (len = 0; len < dif; len++) {
00055         firstrow = rowstart[len];
00056         lastrow  = rowstart[len+1] - 1;
00057         rowlen   = nz_diff[len];
00058
00059         if (lastrow > firstrow) {
00060             //-----
00061             // Fully-unrolled code for special case (i.g., rowlen = 5)
00062             // Note: you can also set other special case
00063             //-----
00064             if (rowlen == 5) {
00065                 for (row = firstrow; row < lastrow; row += 2) {
00066                     val0 = a[col]*x[ja[col]];
00067                     val1 = a[col+5]*x[ja[col+5]];
00068                     col++;
00069
00070                     val0 += a[col]*x[ja[col]];
00071                     val1 += a[col+5]*x[ja[col+5]];
00072                     col++;
00073
00074                     val0 += a[col]*x[ja[col]];
00075                     val1 += a[col+5]*x[ja[col+5]];
00076                     col++;
00077
00078                     val0 += a[col]*x[ja[col]];
00079                     val1 += a[col+5]*x[ja[col+5]];
00080                     col++;
00081
00082                     val0 += a[col]*x[ja[col]];
00083                     val1 += a[col+5]*x[ja[col+5]];
00084                     col++;
00085
00086                     y[rowindex[row]] = val0;
00087                     y[rowindex[row+1]] = val1;
00088
00089                     col += 5;
00090                 }
00091             }
00092             else {
00093                 //-----
00094                 // Unroll-and-jammed code for handling two rows at a time
00095                 //-----
00096
00097                 for (row = firstrow; row < lastrow; row += 2) {
00098                     val0 = 0.0;
00099                     val1 = 0.0;
00100                     for (i = 0; i < rowlen; i++) {
00101                         val0 += a[col]*x[ja[col]];
00102                         val1 += a[col+rowlen]*x[ja[col+rowlen]];
00103                         col++;
00104                     }
00105                     y[rowindex[row]] = val0;
00106                     y[rowindex[row+1]] = val1;
00107                     col += rowlen;
00108                 }
00109             }
00110             firstrow = row;
00111         }
00112
00113         //-----
00114         // Handle leftover rows that can't be handled in bundles
00115         // in the unroll-and-jammed loop
00116         //-----
00117
00118         for (row = firstrow; row <= lastrow; row++) {
00119             val0 = 0.0;
00120             for (i = 0; i < rowlen; i++) {
00121                 val0 += a[col]*x[ja[col]];
00122                 col++;
00123             }
00124             y[rowindex[row]] = val0;
00125         }
00126     }
00127 }

```

```

00128
00129 }
00130
00131 /*-----*/
00132 /*--      End of File      --*/
00133 /*-----*/

```

9.95 BlaSpmvSTR.c File Reference

Linear algebraic operations for [dSTRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_blas_dstr_aAxy](#) (const [REAL](#) alpha, const [dSTRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = \alpha * A * x + y$.*
- void [fasp_blas_dstr_mxy](#) (const [dSTRmat](#) *A, const [REAL](#) *x, [REAL](#) *y)
*Matrix-vector multiplication $y = A * x$.*
- [INT fasp_blas_dstr_diagscale](#) (const [dSTRmat](#) *A, [dSTRmat](#) *B)
 *$B = D^{-1} * A$.*

9.95.1 Detailed Description

Linear algebraic operations for [dSTRmat](#) matrices.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaSmallMatInv.c](#), [BlaSmallMat.c](#), and [BlaSparseSTR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaSpmvSTR.c](#).

9.95.2 Function Documentation

9.95.2.1 fasp_blas_dstr_aAxy()

```

void fasp_blas_dstr_aAxy (
    const REAL alpha,
    const dSTRmat * A,
    const REAL * x,
    REAL * y )

```

Matrix-vector multiplication $y = \alpha * A * x + y$.

Parameters

<i>alpha</i>	REAL factor alpha
<i>A</i>	Pointer to dSTRmat matrix

Parameters

<i>x</i>	Pointer to REAL array
<i>y</i>	Pointer to REAL array

Author

Zhiyang Zhou, Xiaozhe Hu, Shiquan Zhang

Date

2010/10/15

Definition at line 61 of file [BlaSpmvSTR.c](#).

9.95.2.2 fasp_blas_dstr_diagscale()

```
INT fasp_blas_dstr_diagscale (
    const dSTRmat * A,
    dSTRmat * B )
```

$B = D^{-1}A$.

Parameters

<i>A</i>	Pointer to a 'dSTRmat' type matrix A
<i>B</i>	Pointer to a 'dSTRmat' type matrix B

Author

Shiquan Zhang

Date

2010/10/15

Modified by Chunsheng Feng, Zheng Li on 08/30/2012

Definition at line 155 of file [BlaSpmvSTR.c](#).

9.95.2.3 fasp_blas_dstr_m xv()

```
void fasp_blas_dstr_m xv (
    const dSTRmat * A,
    const REAL * x,
    REAL * y )
```

Matrix-vector multiplication $y = Ax$.

Parameters

<i>A</i>	Pointer to dSTRmat matrix
<i>x</i>	Pointer to REAL array
<i>y</i>	Pointer to REAL array

Author

Chensong Zhang

Date

04/27/2013

Definition at line 131 of file BlaSpmvSTR.c.

9.96 BlaSpmvSTR.c

[Go to the documentation of this file.](#)

```

00001
00015 #include <math.h>
00016
00017 #ifdef _OPENMP
00018 #include <omp.h>
00019 #endif
00020
00021 #include "fasp.h"
00022 #include "fasp_functs.h"
00023
00024 /*-----*/
00025 /*--  Declare Private Functions  --*/
00026 /*-----*/
00027
00028 static inline void smat_amxv_nc3(const REAL, const REAL *, const REAL *, REAL *);
00029 static inline void smat_amxv_nc5(const REAL, const REAL *, const REAL *, REAL *);
00030 static inline void smat_amxv(const REAL, const REAL *, const REAL *, const INT, REAL *);
00031 static inline void str_spaAxy_2D_nc1(const REAL, const dSTRmat *, const REAL *, REAL *);
00032 static inline void str_spaAxy_2D_nc3(const REAL, const dSTRmat *, const REAL *, REAL *);
00033 static inline void str_spaAxy_2D_nc5(const REAL, const dSTRmat *, const REAL *, REAL *);
00034 static inline void str_spaAxy_2D_blk(const REAL, const dSTRmat *, const REAL *, REAL *);
00035 static inline void str_spaAxy_3D_nc1(const REAL, const dSTRmat *, const REAL *, REAL *);
00036 static inline void str_spaAxy_3D_nc3(const REAL, const dSTRmat *, const REAL *, REAL *);
00037 static inline void str_spaAxy_3D_nc5(const REAL, const dSTRmat *, const REAL *, REAL *);
00038 static inline void str_spaAxy_3D_blk(const REAL, const dSTRmat *, const REAL *, REAL *);
00039 static inline void str_spaAxy(const REAL, const dSTRmat *, const REAL *, REAL *);
00040 static inline void blkcontr_str(const INT, const INT, const INT, const INT,
00041                                const REAL *, const REAL *, REAL *);
00042
00043 /*-----*/
00044 /*--      Public Functions      --*/
00045 /*-----*/
00046
00061 void fasp_blas_dstr_aAxy (const REAL      alpha,
00062                          const dSTRmat  *A,
00063                          const REAL     *x,
00064                          REAL           *y)
00065 {
00066     switch (A->nband) {
00067     case 4:
00068         switch (A->nc) {
00069         case 1:
00070             str_spaAxy_2D_nc1(alpha, A, x, y);
00071             break;
00072         case 3:
00073             str_spaAxy_2D_nc3(alpha, A, x, y);
00074             break;
00075         case 5:
00076             str_spaAxy_2D_nc5(alpha, A, x, y);
00077             break;
00078         default:
00079             str_spaAxy_2D_blk(alpha, A, x, y);
00080             break;
00081         }
00082     }
00083     break;
00084 }
00085
00086
00087
00088
00089
00090

```

```

00091         case 6:
00092
00093             switch (A->nc) {
00094                 case 1:
00095                     str_spaAxy_3D_nc1(alpha, A, x, y);
00096                     break;
00097
00098                 case 3:
00099                     str_spaAxy_3D_nc3(alpha, A, x, y);
00100                     break;
00101
00102                 case 5:
00103                     str_spaAxy_3D_nc5(alpha, A, x, y);
00104                     break;
00105
00106                 default:
00107                     str_spaAxy_3D_blk(alpha, A, x, y);
00108                     break;
00109             }
00110             break;
00111
00112         default:
00113             str_spaAxy(alpha, A, x, y);
00114             break;
00115     }
00116 }
00117 }
00118
00131 void fasp_blas_dstr_mnv (const dSTRmat *A,
00132                         const REAL *x,
00133                         REAL *y)
00134 {
00135     int n = (A->ngrid)*(A->nc)*(A->nb);
00136     memset(y, 0, n*sizeof(REAL));
00137     fasp_blas_dstr_aAxy(1.0, A, x, y);
00138 }
00139
00155 INT fasp_blas_dstr_diagscale (const dSTRmat *A,
00156                             dSTRmat *B)
00157 {
00158     const INT ngrid=A->ngrid, nc=A->nc, nband=A->nband;
00159     const INT nc2=nc*nc, size=ngrid*nc2;
00160     INT i, j, ic2, nb, nbl;
00161
00162 #ifdef _OPENMP
00163     //variables for OpenMP
00164     INT myid, mybegin, myend;
00165     INT nthreads = fasp_get_num_threads();
00166 #endif
00167
00168     REAL *diag=(REAL *)fasp_mem_calloc(size, sizeof(REAL));
00169
00170     fasp_darray_cp(size, A->diag, diag);
00171
00172     fasp_dstr_alloc(A->nx, A->ny, A->nz, A->nxy, ngrid, nband, nc, A->offsets, B);
00173
00174     //compute diagonal elements of B
00175 #ifdef _OPENMP
00176     if (ngrid > OPENMP_HOLDS) {
00177 #pragma omp parallel for private(myid, mybegin, myend, i, ic2, j)
00178         for (myid=0; myid<nthreads; myid++) {
00179             fasp_get_start_end(myid, nthreads, ngrid, &mybegin, &myend);
00180             for (i=mybegin; i<myend; i++) {
00181                 ic2=i*nc2;
00182                 for (j=0; j<nc2; j++) {
00183                     if (j/nc == j%nc) B->diag[ic2+j]=1;
00184                     else B->diag[ic2+j]=0;
00185                 }
00186             }
00187         }
00188     }
00189     else {
00190 #endif
00191         for (i=0; i<ngrid; ++i) {
00192             ic2=i*nc2;
00193             for (j=0; j<nc2; ++j) {
00194                 if (j/nc == j%nc) B->diag[ic2+j]=1;
00195                 else B->diag[ic2+j]=0;
00196             }
00197         }
00198     }
00199 }

```

```

00197     }
00198 #ifdef _OPENMP
00199     }
00200 #endif
00201
00202     for (i=0; i<ngrid; ++i) fasp_smat_inv(&(diag[i*nc2]), nc);
00203
00204     for (i=0; i<nband; ++i) {
00205         nb=A->offsets[i];
00206         nb1=abs(nb);
00207         if (nb<0) {
00208             for (j=0; j<ngrid-nb1; ++j)
00209                 fasp_blas_smat_mul(&(diag[(j+nb1)*nc2]), &(A->offdiag[i][j*nc2]), &(B->offdiag[i][j*nc2]), nc);
00210         }
00211         else {
00212             for (j=0; j<ngrid-nb1; ++j)
00213                 fasp_blas_smat_mul(&(diag[j*nc2]), &(A->offdiag[i][j*nc2]), &(B->offdiag[i][j*nc2]), nc);
00214         }
00215     }
00216
00217     fasp_mem_free(diag); diag = NULL;
00218
00219     return (0);
00220 }
00221
00222 /*-----*/
00223 /*--      Private Functions      --*/
00224 /*-----*/
00225
00226 static inline void smat_amxv_nc3 (const REAL  alpha,
00227                                  const REAL  *a,
00228                                  const REAL  *b,
00229                                  REAL          *c)
00230 {
00231     c[0] += alpha*(a[0]*b[0] + a[1]*b[1] + a[2]*b[2]);
00232     c[1] += alpha*(a[3]*b[0] + a[4]*b[1] + a[5]*b[2]);
00233     c[2] += alpha*(a[6]*b[0] + a[7]*b[1] + a[8]*b[2]);
00234 }
00235
00236 static inline void smat_amxv_nc5 (const REAL  alpha,
00237                                  const REAL  *a,
00238                                  const REAL  *b,
00239                                  REAL          *c)
00240 {
00241     c[0] += alpha*(a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + a[3] * b[3] + a[4] * b[4]);
00242     c[1] += alpha*(a[5]*b[0] + a[6]*b[1] + a[7]*b[2] + a[8] * b[3] + a[9] * b[4]);
00243     c[2] += alpha*(a[10]*b[0] + a[11]*b[1] + a[12]*b[2] + a[13] * b[3] + a[14] * b[4]);
00244     c[3] += alpha*(a[15]*b[0] + a[16]*b[1] + a[17]*b[2] + a[18] * b[3] + a[19] * b[4]);
00245     c[4] += alpha*(a[20]*b[0] + a[21]*b[1] + a[22]*b[2] + a[23] * b[3] + a[24] * b[4]);
00246 }
00247
00248 static inline void smat_amxv (const REAL  alpha,
00249                               const REAL  *a,
00250                               const REAL  *b,
00251                               const INT    n,
00252                               REAL          *c)
00253 {
00254     INT i, j;
00255     INT in;
00256
00257 #ifdef _OPENMP
00258     // variables for OpenMP
00259     INT myid, mybegin, myend;
00260     INT nthreads = fasp_get_num_threads();
00261 #endif
00262
00263 #ifdef _OPENMP
00264     if (n > OPENMP_HOLDS) {
00265 #pragma omp parallel for private(myid, mybegin, myend, i, in, j)
00266         for (myid=0; myid<nthreads; myid++) {
00267             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00268             for (i=mybegin; i<myend; i++) {
00269                 in = i*n;
00270                 for (j=0; j<n; j++)
00271                     c[i] += alpha*a[in+j]*b[j];
00272             }
00273         }
00274     }
00275     else {

```

```

00322 #endif
00323     for (i=0;i<n;++i) {
00324         in = i*n;
00325         for (j=0;j<n;++j)
00326             c[i] += alpha*a[in+j]*b[j];
00327     }
00328 #ifdef _OPENMP
00329 }
00330 #endif
00331     return;
00332 }
00333
00356 static inline void blkcontr_str (const INT    start_data,
00357                                 const INT    start_vecx,
00358                                 const INT    start_vecy,
00359                                 const INT    nc,
00360                                 const REAL   *data,
00361                                 const REAL   *x,
00362                                 REAL   *y)
00363 {
00364     INT i,j,k,m;
00365
00366 #ifdef _OPENMP
00367     //variables for OpenMP
00368     INT myid, mybegin, myend;
00369     INT nthreads = fasp_get_num_threads();
00370 #endif
00371
00372 #ifdef _OPENMP
00373     if (nc > OPENMP_HOLDS) {
00374 #pragma omp parallel for private(myid, mybegin, myend, i, k, m, j)
00375         for (myid = 0; myid < nthreads; myid++) {
00376             fasp_get_start_end(myid, nthreads, nc, &mybegin, &myend);
00377             for (i = mybegin; i < myend; i++) {
00378                 k = start_data + i*nc;
00379                 m = start_vecy + i;
00380                 for (j = 0; j < nc; j++) {
00381                     y[m] += data[k+j]*x[start_vecx+j];
00382                 }
00383             }
00384         }
00385     }
00386     else {
00387 #endif
00388         for (i = 0; i < nc; i++) {
00389             k = start_data + i*nc;
00390             m = start_vecy + i;
00391             for (j = 0; j < nc; j++) {
00392                 y[m] += data[k+j]*x[start_vecx+j];
00393             }
00394         }
00395 #ifdef _OPENMP
00396     }
00397 #endif
00398 }
00399
00420 static inline void str_spaAxy_2D_nc1 (const REAL   alpha,
00421                                       const dSTRmat *A,
00422                                       const REAL   **x,
00423                                       REAL   **y)
00424 {
00425     INT i;
00426     INT idx1, idx2;
00427     INT endl, end2;
00428     INT nline;
00429
00430 #ifdef _OPENMP
00431     //variables for OpenMP
00432     INT myid, mybegin, myend, idx;
00433     INT nthreads = fasp_get_num_threads();
00434 #endif
00435
00436     // information of A
00437     INT nx = A->nx;
00438     INT ngrid = A->ngrid; // number of grids
00439     INT nband = A->nband;
00440
00441     REAL *diag = A->diag;
00442     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;
00443
00444     if (nx == 1) {

```



```

00445     nline = A->ny;
00446 }
00447 else {
00448     nline = nx;
00449 }
00450
00451 for (i=0; i<nband; ++i) {
00452     if (A->offsets[i] == -1) {
00453         offdiag0 = A->offdiag[i];
00454     }
00455     else if (A->offsets[i] == 1) {
00456         offdiag1 = A->offdiag[i];
00457     }
00458     else if (A->offsets[i] == -nline) {
00459         offdiag2 = A->offdiag[i];
00460     }
00461     else if (A->offsets[i] == nline) {
00462         offdiag3 = A->offdiag[i];
00463     }
00464     else {
00465         printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
00466         str_spaAxy(alpha, A, x, y);
00467         return;
00468     }
00469 }
00470
00471 end1 = ngrid-1;
00472 end2 = ngrid-nline;
00473
00474 y[0] += alpha*(diag[0]*x[0] + offdiag1[0]*x[1] + offdiag3[0]*x[nline]);
00475
00476 #ifdef _OPENMP
00477     if (nline-1 > OPENMP_HOLDS) {
00478 #pragma omp parallel for private(myid, mybegin, myend, i, idx1, idx)
00479         for (myid=0; myid<nthreads; myid++) {
00480             fasp_get_start_end(myid, nthreads, nline-1, &mybegin, &myend);
00481             for (i=mybegin; i<myend; i++) {
00482                 idx1 = i;
00483                 idx = i+1;
00484                 y[idx] += alpha*(offdiag0[idx1]*x[idx1] + diag[idx]*x[idx] +
00485                                offdiag1[idx]*x[idx+1] + offdiag3[idx]*x[idx+nline]);
00486             }
00487         }
00488     }
00489     else {
00490 #endif
00491         for (i=1; i<nline; ++i) {
00492             idx1 = i-1;
00493             y[i] += alpha*(offdiag0[idx1]*x[idx1] + diag[i]*x[i] +
00494                           offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nline]);
00495         }
00496 #ifdef _OPENMP
00497     }
00498 #endif
00499
00500 #ifdef _OPENMP
00501     if (end2-nline > OPENMP_HOLDS) {
00502 #pragma omp parallel for private(myid, i, mybegin, myend, idx1, idx2, idx)
00503         for (myid=0; myid<nthreads; myid++) {
00504             fasp_get_start_end(myid, nthreads, end2-nline, &mybegin, &myend);
00505             for (i=mybegin; i<myend; ++i) {
00506                 idx = i+nline;
00507                 idx1 = idx-1; //idx1 = i-1+nline;
00508                 idx2 = i;
00509                 y[idx] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
00510                                diag[idx]*x[idx] + offdiag1[idx]*x[idx+1] +
00511                                offdiag3[idx]*x[idx+nline]);
00512             }
00513         }
00514     }
00515     else {
00516 #endif
00517         for (i=nline; i<end2; ++i) {
00518             idx1 = i-1;
00519             idx2 = i-nline;
00520             y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
00521                           diag[i]*x[i] + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nline]);
00522         }
00523 #ifdef _OPENMP
00524     }
00525 #endif

```

```

00526
00527 #ifdef _OPENMP
00528     if (endl-end2 > OPENMP_HOLDS) {
00529 #pragma omp parallel for private(myid, i, mybegin, myend, idx1, idx2, idx)
00530         for (myid=0; myid<nthreads; myid++) {
00531             fasp_get_start_end(myid, nthreads, endl-end2, &mybegin, &myend);
00532             for (i=mybegin; i<myend; ++i) {
00533                 idx = i+endl;
00534                 idx1 = idx-1; //idx1 = i-1+endl;
00535                 idx2 = idx-nline; //idx2 = i-nline+endl;
00536                 y[idx] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
00537                                diag[idx]*x[idx] + offdiag1[idx]*x[idx+1]);
00538             }
00539         }
00540     }
00541     else {
00542 #endif
00543         for (i=endl; i<endl+1; ++i) {
00544             idx1 = i-1;
00545             idx2 = i-nline;
00546             y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
00547                           diag[i]*x[i] + offdiag1[i]*x[i+1]);
00548         }
00549 #ifdef _OPENMP
00550     }
00551 #endif
00552
00553     idx1 = endl-1;
00554     idx2 = endl-nline;
00555     y[endl] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] + diag[endl]*x[endl]);
00556
00557     return;
00558
00559 }
00560
00581 static inline void str_spaAxy_2D_nc3 (const REAL    alpha,
00582                                       const dSTRmat *A,
00583                                       const REAL    *x,
00584                                       REAL          *y)
00585 {
00586     INT i;
00587     INT idx,idx1,idx2;
00588     INT matidx, matidx1, matidx2;
00589     INT endl, end2;
00590     INT nline, nlinenc;
00591
00592     // information of A
00593     INT nx = A->nx;
00594     INT ngrid = A->ngrid; // number of grids
00595     INT nc = A->nc;
00596     INT nband = A->nband;
00597
00598 #ifdef _OPENMP
00599     // variables for OpenMP
00600     INT myid, mybegin, myend, up;
00601     INT nthreads = fasp_get_num_threads();
00602 #endif
00603
00604     REAL *diag = A->diag;
00605     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;
00606
00607     if (nx == 1) {
00608         nline = A->ny;
00609     }
00610     else {
00611         nline = nx;
00612     }
00613     nlinenc = nline*nc;
00614
00615     for (i=0; i<nband; ++i) {
00616
00617         if (A->offsets[i] == -1) {
00618             offdiag0 = A->offdiag[i];
00619         }
00620         else if (A->offsets[i] == 1) {
00621             offdiag1 = A->offdiag[i];
00622         }
00623         else if (A->offsets[i] == -nline) {
00624             offdiag2 = A->offdiag[i];
00625         }
00626         else if (A->offsets[i] == nline) {

```

```

00627         offdiag3 = A->offdiag[i];
00628     }
00629     else {
00630         printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
00631         str_spaAxy(alpha, A, x, y);
00632         return;
00633     }
00634 }
00635 }
00636
00637     endl = ngrid-1;
00638     end2 = ngrid-nline;
00639
00640     smat_amxv_nc3(alpha, diag, x, y);
00641     smat_amxv_nc3(alpha, offdiag1, x+nc, y);
00642     smat_amxv_nc3(alpha, offdiag3, x+nlinenc, y);
00643
00644 #ifdef _OPENMP
00645     up = nline - 1;
00646     if (up > OPENMP_HOLDS) {
00647 #pragma omp parallel for private(myid, mybegin, myend, i, idx, matidx, idx1, matidx1)
00648         for (myid=0; myid<nthreads; myid++) {
00649             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00650             for (i=mybegin; i<myend; i++) {
00651                 idx = (i+1)*nc;
00652                 matidx = idx*nc;
00653                 idx1 = i*nc;
00654                 matidx1 = idx1*nc;
00655                 smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00656                 smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00657                 smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00658                 smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00659             }
00660         }
00661     }
00662     else {
00663 #endif
00664         for (i=1; i<nline; ++i) {
00665             idx = i*nc;
00666             matidx = idx*nc;
00667             idx1 = idx - nc;
00668             matidx1 = idx1*nc;
00669             smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00670             smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00671             smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00672             smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00673         }
00674 #ifdef _OPENMP
00675     }
00676 #endif
00677
00678 #ifdef _OPENMP
00679     up = end2 - nx;
00680     if (up > OPENMP_HOLDS) {
00681 #pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
00682         for (myid=0; myid<nthreads; myid++) {
00683             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00684             for (i=mybegin; i<myend; i++) {
00685                 idx = (i+nx)*nc;
00686                 idx1 = idx-nc;
00687                 idx2 = idx-nlinenc;
00688                 matidx = idx*nc;
00689                 matidx1 = idx1*nc;
00690                 matidx2 = idx2*nc;
00691                 smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
00692                 smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00693                 smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00694                 smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00695                 smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00696             }
00697         }
00698     }
00699     else {
00700 #endif
00701         for (i=nx; i<end2; ++i) {
00702             idx = i*nc;
00703             idx1 = idx-nc;
00704             idx2 = idx-nlinenc;
00705             matidx = idx*nc;
00706             matidx1 = idx1*nc;
00707             matidx2 = idx2*nc;

```

```

00708         smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
00709         smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00710         smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00711         smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00712         smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00713     }
00714 #ifdef _OPENMP
00715     }
00716 #endif
00717 #ifdef _OPENMP
00718     up = endl - end2;
00719     if (up > OPENMP_HOLDS) {
00720 #pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
00721         for (myid=0; myid<nthreads; myid++) {
00722             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00723             for (i=mybegin; i<myend; i++) {
00724                 idx = (i+end2)*nc;
00725                 idx1 = idx-nc;
00726                 idx2 = idx-nlinenc;
00727                 matidx = idx*nc;
00728                 matidx1 = idx1*nc;
00729                 matidx2 = idx2*nc;
00730                 smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
00731                 smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00732                 smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00733                 smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00734             }
00735         }
00736     }
00737 }
00738 else {
00739 #endif
00740     for (i=end2; i<endl; ++i) {
00741         idx = i*nc;
00742         idx1 = idx-nc;
00743         idx2 = idx-nlinenc;
00744         matidx = idx*nc;
00745         matidx1 = idx1*nc;
00746         matidx2 = idx2*nc;
00747         smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
00748         smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00749         smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00750         smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00751     }
00752 #ifdef _OPENMP
00753     }
00754 #endif
00755     i=endl;
00756     idx = i*nc;
00757     idx1 = idx-nc;
00758     idx2 = idx-nlinenc;
00759     matidx = idx*nc;
00760     matidx1 = idx1*nc;
00761     matidx2 = idx2*nc;
00762     smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
00763     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
00764     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
00765
00766     return;
00767 }
00768
00787 static inline void str_spaApxy_2D_nc5 (const REAL    alpha,
00788                                         const dSTRmat *A,
00789                                         const REAL    *x,
00790                                         REAL          *y)
00791 {
00792     INT i;
00793     INT idx,idx1,idx2;
00794     INT matidx, matidx1, matidx2;
00795     INT endl, end2;
00796     INT nline, nlinenc;
00797
00798     // information of A
00799     INT nx = A->nx;
00800     INT ngrid = A->ngrid; // number of grids
00801     INT nc = A->nc;
00802     INT nband = A->nband;
00803
00804 #ifdef _OPENMP
00805     // variables for OpenMP
00806     INT myid, mybegin, myend, up;

```

```

00807     INT nthreads = fasp_get_num_threads();
00808 #endif
00809
00810     REAL *diag = A->diag;
00811     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;
00812
00813     if (nx == 1) {
00814         nline = A->ny;
00815     }
00816     else {
00817         nline = nx;
00818     }
00819     nlinenc = nline*nc;
00820
00821     for (i=0; i<nband; ++i) {
00822         if (A->offsets[i] == -1) {
00823             offdiag0 = A->offdiag[i];
00824         }
00825         else if (A->offsets[i] == 1) {
00826             offdiag1 = A->offdiag[i];
00827         }
00828         else if (A->offsets[i] == -nline) {
00829             offdiag2 = A->offdiag[i];
00830         }
00831         else if (A->offsets[i] == nline) {
00832             offdiag3 = A->offdiag[i];
00833         }
00834         else {
00835             printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
00836             str_spaAxy(alpha, A, x, y);
00837             return;
00838         }
00839     }
00840
00841     endl = ngrid-1;
00842     end2 = ngrid-nline;
00843
00844     smat_amxv_nc5(alpha, diag, x, y);
00845     smat_amxv_nc5(alpha, offdiag1, x+nc, y);
00846     smat_amxv_nc5(alpha, offdiag3, x+nlinenc, y);
00847
00848 #ifdef _OPENMP
00849     up = nline - 1;
00850     if (up > OPENMP_HOLDS) {
00851 #pragma omp parallel for private(myid, mybegin, myend, i, idx, matidx, idx1, matidx1)
00852         for (myid=0; myid<nthreads; myid++) {
00853             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00854             for (i=mybegin; i<myend; i++) {
00855                 idx = (i+1)*nc;
00856                 matidx = idx*nc;
00857                 idx1 = i*nc; // idx1 = idx - nc;
00858                 matidx1 = idx1*nc;
00859                 smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00860                 smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00861                 smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00862                 smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00863             }
00864         }
00865     }
00866     else {
00867 #endif
00868         for (i=1; i<nline; ++i) {
00869             idx = i*nc;
00870             matidx = idx*nc;
00871             idx1 = idx - nc;
00872             matidx1 = idx1*nc;
00873             smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00874             smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00875             smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00876             smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00877         }
00878 #ifdef _OPENMP
00879     }
00880 #endif
00881 #endif
00882
00883 #ifdef _OPENMP
00884     up = end2 - nx;
00885     if (up > OPENMP_HOLDS) {
00886 #pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
00887         for (myid=0; myid<nthreads; myid++) {

```

```

00888         fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00889     for (i=mybegin; i<myend; i++) {
00890         idx = (i+nx)*nc;
00891         idx1 = idx-nc;
00892         idx2 = idx-nlinenc;
00893         matidx = idx*nc;
00894         matidx1 = idx1*nc;
00895         matidx2 = idx2*nc;
00896         smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
00897         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00898         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00899         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00900         smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00901     }
00902 }
00903 }
00904     else {
00905 #endif
00906     for (i=nx; i<end2; ++i) {
00907         idx = i*nc;
00908         idx1 = idx-nc;
00909         idx2 = idx-nlinenc;
00910         matidx = idx*nc;
00911         matidx1 = idx1*nc;
00912         matidx2 = idx2*nc;
00913         smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
00914         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00915         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00916         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00917         smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
00918     }
00919 #ifdef _OPENMP
00920 }
00921 #endif
00922
00923 #ifdef _OPENMP
00924     up = endl - end2;
00925     if (up > OPENMP_HOLDS) {
00926 #pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
00927         for (myid=0; myid<nthreads; myid++) {
00928             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00929             for (i=mybegin; i<myend; i++) {
00930                 idx = (i+end2)*nc;
00931                 idx1 = idx-nc;
00932                 idx2 = idx-nlinenc;
00933                 matidx = idx*nc;
00934                 matidx1 = idx1*nc;
00935                 matidx2 = idx2*nc;
00936                 smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
00937                 smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00938                 smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00939                 smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00940             }
00941         }
00942     }
00943     else {
00944 #endif
00945     for (i=end2; i<endl; ++i) {
00946         idx = i*nc;
00947         idx1 = idx-nc;
00948         idx2 = idx-nlinenc;
00949         matidx = idx*nc;
00950         matidx1 = idx1*nc;
00951         matidx2 = idx2*nc;
00952         smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
00953         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00954         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00955         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
00956     }
00957 #ifdef _OPENMP
00958 }
00959 #endif
00960
00961     i=endl;
00962     idx = i*nc;
00963     idx1 = idx-nc;
00964     idx2 = idx-nlinenc;
00965     matidx = idx*nc;
00966     matidx1 = idx1*nc;
00967     matidx2 = idx2*nc;
00968     smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);

```

```

00969     smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
00970     smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
00971
00972     return;
00973 }
00974
00975 static inline void str_spaAxy_2D_blk (const REAL      alpha,
00994                                     const dSTRmat    *A,
00995                                     const REAL        *x,
00996                                     REAL              *y)
00997 {
00998     {
00999         INT i;
01000         INT idx,idx1,idx2;
01001         INT matidx, matidx1, matidx2;
01002         INT endl, end2;
01003         INT nline, nlinenc;
01004
01005         // information of A
01006         INT nx = A->nx;
01007         INT ngrid = A->ngrid; // number of grids
01008         INT nc = A->nc;
01009         INT nband = A->nband;
01010
01011         REAL *diag = A->diag;
01012         REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;
01013
01014         if (nx == 1) {
01015             nline = A->ny;
01016         }
01017         else {
01018             nline = nx;
01019         }
01020         nlinenc = nline*nc;
01021
01022         for (i=0; i<nband; ++i) {
01023             if (A->offsets[i] == -1) {
01024                 offdiag0 = A->offdiag[i];
01025             }
01026             else if (A->offsets[i] == 1) {
01027                 offdiag1 = A->offdiag[i];
01028             }
01029             else if (A->offsets[i] == -nline) {
01030                 offdiag2 = A->offdiag[i];
01031             }
01032             else if (A->offsets[i] == nline) {
01033                 offdiag3 = A->offdiag[i];
01034             }
01035             else {
01036                 printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
01037                 str_spaAxy(alpha, A, x, y);
01038                 return;
01039             }
01040         }
01041     }
01042 }
01043
01044 endl = ngrid-1;
01045 end2 = ngrid-nline;
01046
01047 smat_amxv(alpha, diag, x, nc, y);
01048 smat_amxv(alpha, offdiag1, x+nc, nc, y);
01049 smat_amxv(alpha, offdiag3, x+nlinenc, nc, y);
01050
01051 for (i=1; i<nline; ++i) {
01052     idx = i*nc;
01053     matidx = idx*nc;
01054     idx1 = idx - nc;
01055     matidx1 = idx1*nc;
01056     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01057     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01058     smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01059     smat_amxv(alpha, offdiag3+matidx, x+idx+nlinenc, nc, y+idx);
01060 }
01061
01062 for (i=nx; i<end2; ++i) {
01063     idx = i*nc;
01064     idx1 = idx-nc;
01065     idx2 = idx-nlinenc;
01066     matidx = idx*nc;
01067     matidx1 = idx1*nc;

```

```

01068         matidx2 = idx2*nc;
01069         smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01070         smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01071         smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01072         smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01073         smat_amxv(alpha, offdiag3+matidx, x+idx+nlinenc, nc, y+idx);
01074     }
01075
01076     for (i=end2; i<endl; ++i) {
01077         idx = i*nc;
01078         idx1 = idx-nc;
01079         idx2 = idx-nlinenc;
01080         matidx = idx*nc;
01081         matidx1 = idx1*nc;
01082         matidx2 = idx2*nc;
01083         smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01084         smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01085         smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01086         smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01087     }
01088
01089     i=endl;
01090     idx = i*nc;
01091     idx1 = idx-nc;
01092     idx2 = idx-nlinenc;
01093     matidx = idx*nc;
01094     matidx1 = idx1*nc;
01095     matidx2 = idx2*nc;
01096     smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01097     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01098     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01099
01100     return;
01101 }
01102
01121 static inline void str_spaAxy_3D_nc1 (const REAL      alpha,
01122                                       const dSTRmat    *A,
01123                                       const REAL        *x,
01124                                       REAL              *y)
01125 {
01126     INT i;
01127     INT idx1,idx2,idx3;
01128     INT endl, end2, end3;
01129     // information of A
01130     INT nx = A->nx;
01131     INT nxy = A->nxy;
01132     INT ngrid = A->ngrid; // number of grids
01133     INT nband = A->nband;
01134
01135     REAL *diag = A->diag;
01136     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
01137     *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;
01138
01139     for (i=0; i<nband; ++i) {
01140
01141         if (A->offsets[i] == -1) {
01142             offdiag0 = A->offdiag[i];
01143         }
01144         else if (A->offsets[i] == 1) {
01145             offdiag1 = A->offdiag[i];
01146         }
01147         else if (A->offsets[i] == -nx) {
01148             offdiag2 = A->offdiag[i];
01149         }
01150         else if (A->offsets[i] == nx) {
01151             offdiag3 = A->offdiag[i];
01152         }
01153         else if (A->offsets[i] == -nxy) {
01154             offdiag4 = A->offdiag[i];
01155         }
01156         else if (A->offsets[i] == nxy) {
01157             offdiag5 = A->offdiag[i];
01158         }
01159         else {
01160             printf("### WARNING: offsets for 3D scalar is illegal!  %s\n", __FUNCTION__);
01161             str_spaAxy(alpha, A, x, y);
01162             return;
01163         }
01164     }
01165
01166     endl = ngrid-1;

```



```

01167     end2 = ngrid-nx;
01168     end3 = ngrid-nxy;
01169
01170     y[0] += alpha*(diag[0]*x[0] + offdiag1[0]*x[1] + offdiag3[0]*x[nx] + offdiag5[0]*x[nxy]);
01171
01172     for (i=1; i<nx; ++i) {
01173         idx1 = i-1;
01174         y[i] += alpha*(offdiag0[idx1]*x[idx1] + diag[i]*x[i] + offdiag1[i]*x[i+1] +
01175             offdiag3[i]*x[i+nx] + offdiag5[i]*x[i+nxy]);
01176     }
01177
01178     for (i=nx; i<nxy; ++i) {
01179         idx1 = i-1;
01180         idx2 = i-nx;
01181         y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1]
01182             + diag[i]*x[i] + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nx]
01183             + offdiag5[i]*x[i+nxy]);
01184     }
01185
01186     for (i=nxy; i<end3; ++i) {
01187         idx1 = i-1;
01188         idx2 = i-nx;
01189         idx3 = i-nxy;
01190         y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
01191             + offdiag0[idx1]*x[idx1] + diag[i]*x[i] + offdiag1[i]*x[i+1]
01192             + offdiag3[i]*x[i+nx] + offdiag5[i]*x[i+nxy]);
01193     }
01194
01195     for (i=end3; i<end2; ++i) {
01196         idx1 = i-1;
01197         idx2 = i-nx;
01198         idx3 = i-nxy;
01199         y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
01200             + offdiag0[idx1]*x[idx1] + diag[i]*x[i]
01201             + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nx]);
01202     }
01203
01204     for (i=end2; i<end1; ++i) {
01205         idx1 = i-1;
01206         idx2 = i-nx;
01207         idx3 = i-nxy;
01208         y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
01209             + offdiag0[idx1]*x[idx1] + diag[i]*x[i]
01210             + offdiag1[i]*x[i+1]);
01211     }
01212
01213     idx1 = end1-1;
01214     idx2 = end1-nx;
01215     idx3 = end1-nxy;
01216     y[end1] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2] +
01217         offdiag0[idx1]*x[idx1] + diag[end1]*x[end1]);
01218
01219     return;
01220 }
01221
01240 static inline void str_spaApy_3D_nc3 (const REAL      alpha,
01241     const dSTRmat  *A,
01242     const REAL      *x,
01243     REAL            *y)
01244 {
01245     INT i;
01246     INT idx,idx1,idx2,idx3;
01247     INT matidx, matidx1, matidx2, matidx3;
01248     INT end1, end2, end3;
01249     // information of A
01250     INT nx = A->nx;
01251     INT nxy = A->nxy;
01252     INT ngrid = A->ngrid; // number of grids
01253     INT nc = A->nc;
01254     INT nxnc = nx*nc;
01255     INT nxync = nxy*nc;
01256     INT nband = A->nband;
01257
01258     REAL *diag = A->diag;
01259     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
01260     *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;
01261
01262     for (i=0; i<nband; ++i) {
01263
01264         if (A->offsets[i] == -1) {
01265             offdiag0 = A->offdiag[i];

```

```

01266     }
01267     else if (A->offsets[i] == 1) {
01268         offdiag1 = A->offdiag[i];
01269     }
01270     else if (A->offsets[i] == -nx) {
01271         offdiag2 = A->offdiag[i];
01272     }
01273     else if (A->offsets[i] == nx) {
01274         offdiag3 = A->offdiag[i];
01275     }
01276     else if (A->offsets[i] == -nxy) {
01277         offdiag4 = A->offdiag[i];
01278     }
01279     else if (A->offsets[i] == nxy) {
01280         offdiag5 = A->offdiag[i];
01281     }
01282     else {
01283         printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
01284         str_spaAxy(alpha, A, x, y);
01285         return;
01286     }
01287 }
01288
01289 end1 = ngrid-1;
01290 end2 = ngrid-nx;
01291 end3 = ngrid-nxy;
01292
01293 smat_amxv_nc3(alpha, diag, x, y);
01294 smat_amxv_nc3(alpha, offdiag1, x+nc, y);
01295 smat_amxv_nc3(alpha, offdiag3, x+nxnc, y);
01296 smat_amxv_nc3(alpha, offdiag5, x+nxync, y);
01297
01298 for (i=1; i<nx; ++i) {
01299     idx = i*nc;
01300     matidx = idx*nc;
01301     idx1 = idx - nc;
01302     matidx1 = idx1*nc;
01303     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01304     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01305     smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01306     smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01307     smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01308 }
01309
01310 for (i=nx; i<nxy; ++i) {
01311     idx = i*nc;
01312     idx1 = idx-nc;
01313     idx2 = idx-nxnc;
01314     matidx = idx*nc;
01315     matidx1 = idx1*nc;
01316     matidx2 = idx2*nc;
01317     smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
01318     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01319     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01320     smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01321     smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01322     smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01323 }
01324
01325
01326 for (i=nxy; i<end3; ++i) {
01327     idx = i*nc;
01328     idx1 = idx-nc;
01329     idx2 = idx-nxnc;
01330     idx3 = idx-nxync;
01331     matidx = idx*nc;
01332     matidx1 = idx1*nc;
01333     matidx2 = idx2*nc;
01334     matidx3 = idx3*nc;
01335     smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
01336     smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
01337     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01338     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01339     smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01340     smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01341     smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01342 }
01343
01344 for (i=end3; i<end2; ++i) {
01345     idx = i*nc;
01346     idx1 = idx-nc;

```

```

01347     idx2 = idx-nxnc;
01348     idx3 = idx-nxync;
01349     matidx = idx*nc;
01350     matidx1 = idx1*nc;
01351     matidx2 = idx2*nc;
01352     matidx3 = idx3*nc;
01353     smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
01354     smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
01355     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01356     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01357     smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01358     smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01359 }
01360
01361 for (i=end2; i<endl; ++i) {
01362     idx = i*nc;
01363     idx1 = idx-nc;
01364     idx2 = idx-nxnc;
01365     idx3 = idx-nxync;
01366     matidx = idx*nc;
01367     matidx1 = idx1*nc;
01368     matidx2 = idx2*nc;
01369     matidx3 = idx3*nc;
01370     smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
01371     smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
01372     smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01373     smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01374     smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01375 }
01376
01377 i=endl;
01378 idx = i*nc;
01379 idx1 = idx-nc;
01380 idx2 = idx-nxnc;
01381 idx3 = idx-nxync;
01382 matidx = idx*nc;
01383 matidx1 = idx1*nc;
01384 matidx2 = idx2*nc;
01385 matidx3 = idx3*nc;
01386 smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
01387 smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
01388 smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
01389 smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
01390
01391 return;
01392
01393 }
01394
01413 static inline void str_spaAxy3D_nc5 (const REAL      alpha,
01414                                     const dSTRmat    *A,
01415                                     const REAL      *x,
01416                                     REAL            *y)
01417 {
01418     INT i;
01419     INT idx,idx1,idx2,idx3;
01420     INT matidx, matidx1, matidx2, matidx3;
01421     INT endl, end2, end3;
01422     // information of A
01423     INT nx = A->nx;
01424     INT nxy = A->nxy;
01425     INT ngrid = A->ngrid; // number of grids
01426     INT nc = A->nc;
01427     INT nxnc = nx*nc;
01428     INT nxync = nxy*nc;
01429     INT nband = A->nband;
01430
01431     REAL *diag = A->diag;
01432     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
01433     *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;
01434
01435     for (i=0; i<nband; ++i) {
01436
01437         if (A->offsets[i] == -1) {
01438             offdiag0 = A->offdiag[i];
01439         }
01440         else if (A->offsets[i] == 1) {
01441             offdiag1 = A->offdiag[i];
01442         }
01443         else if (A->offsets[i] == -nx) {
01444             offdiag2 = A->offdiag[i];
01445         }

```

```

01446         else if (A->offsets[i] == nx) {
01447             offdiag3 = A->offdiag[i];
01448         }
01449         else if (A->offsets[i] == -nxy) {
01450             offdiag4 = A->offdiag[i];
01451         }
01452         else if (A->offsets[i] == nxy) {
01453             offdiag5 = A->offdiag[i];
01454         }
01455         else {
01456             printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
01457             str_spaAxy(alpha, A, x, y);
01458             return;
01459         }
01460     }
01461
01462     end1 = ngrid-1;
01463     end2 = ngrid-nx;
01464     end3 = ngrid-nxy;
01465
01466     smat_amxv_nc5(alpha, diag, x, y);
01467     smat_amxv_nc5(alpha, offdiag1, x+nc, y);
01468     smat_amxv_nc5(alpha, offdiag3, x+nxnc, y);
01469     smat_amxv_nc5(alpha, offdiag5, x+nxync, y);
01470
01471     for (i=1; i<nx; ++i) {
01472         idx = i*nc;
01473         matidx = idx*nc;
01474         idx1 = idx - nc;
01475         matidx1 = idx1*nc;
01476         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01477         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01478         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01479         smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01480         smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01481     }
01482
01483     for (i=nx; i<nxy; ++i) {
01484         idx = i*nc;
01485         idx1 = idx-nc;
01486         idx2 = idx-nxnc;
01487         matidx = idx*nc;
01488         matidx1 = idx1*nc;
01489         matidx2 = idx2*nc;
01490         smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
01491         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01492         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01493         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01494         smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01495         smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01496     }
01497
01498
01499     for (i=nxy; i<end3; ++i) {
01500         idx = i*nc;
01501         idx1 = idx-nc;
01502         idx2 = idx-nxnc;
01503         idx3 = idx-nxync;
01504         matidx = idx*nc;
01505         matidx1 = idx1*nc;
01506         matidx2 = idx2*nc;
01507         matidx3 = idx3*nc;
01508         smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
01509         smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
01510         smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01511         smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01512         smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01513         smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01514         smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
01515     }
01516
01517     for (i=end3; i<end2; ++i) {
01518         idx = i*nc;
01519         idx1 = idx-nc;
01520         idx2 = idx-nxnc;
01521         idx3 = idx-nxync;
01522         matidx = idx*nc;
01523         matidx1 = idx1*nc;
01524         matidx2 = idx2*nc;
01525         matidx3 = idx3*nc;
01526         smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);

```

```

01527     smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
01528     smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01529     smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01530     smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01531     smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
01532 }
01533
01534 for (i=end2; i<endl; ++i) {
01535     idx = i+nc;
01536     idx1 = idx-nc;
01537     idx2 = idx-nxnc;
01538     idx3 = idx-nxync;
01539     matidx = idx*nc;
01540     matidx1 = idx1*nc;
01541     matidx2 = idx2*nc;
01542     matidx3 = idx3*nc;
01543     smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
01544     smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
01545     smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01546     smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01547     smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
01548 }
01549
01550 i=end1;
01551 idx = i+nc;
01552 idx1 = idx-nc;
01553 idx2 = idx-nxnc;
01554 idx3 = idx-nxync;
01555 matidx = idx*nc;
01556 matidx1 = idx1*nc;
01557 matidx2 = idx2*nc;
01558 matidx3 = idx3*nc;
01559 smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
01560 smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
01561 smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
01562 smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
01563
01564 return;
01565 }
01566
01567 static inline void str_spaAxy_3D_blk (const REAL      alpha,
01568                                     const dSTRmat    *A,
01569                                     const REAL      *x,
01570                                     REAL            *y)
01571 {
01572     INT i;
01573     INT idx,idx1,idx2,idx3;
01574     INT matidx, matidx1, matidx2, matidx3;
01575     INT endl, end2, end3;
01576     // information of A
01577     INT nx = A->nx;
01578     INT nxy = A->nxy;
01579     INT ngrid = A->ngrid; // number of grids
01580     INT nc = A->nc;
01581     INT nxnc = nx*nc;
01582     INT nxync = nxy*nc;
01583     INT nband = A->nband;
01584
01585     REAL *diag = A->diag;
01586     REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
01587     *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;
01588
01589     for (i=0; i<nband; ++i) {
01590         if (A->offsets[i] == -1) {
01591             offdiag0 = A->offdiag[i];
01592         }
01593         else if (A->offsets[i] == 1) {
01594             offdiag1 = A->offdiag[i];
01595         }
01596         else if (A->offsets[i] == -nx) {
01597             offdiag2 = A->offdiag[i];
01598         }
01599         else if (A->offsets[i] == nx) {
01600             offdiag3 = A->offdiag[i];
01601         }
01602         else if (A->offsets[i] == -nxy) {
01603             offdiag4 = A->offdiag[i];
01604         }
01605         else if (A->offsets[i] == nxy) {
01606

```

```

01626         offdiag5 = A->offdiag[i];
01627     }
01628     else {
01629         printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
01630         str_spaAxy(alpha, A, x, y);
01631         return;
01632     }
01633 }
01634
01635 end1 = ngrid-1;
01636 end2 = ngrid-nx;
01637 end3 = ngrid-nxy;
01638
01639 smat_amxv(alpha, diag, x, nc, y);
01640 smat_amxv(alpha, offdiag1, x+nc, nc, y);
01641 smat_amxv(alpha, offdiag3, x+nxnc, nc, y);
01642 smat_amxv(alpha, offdiag5, x+nxync, nc, y);
01643
01644 for (i=1; i<nx; ++i) {
01645     idx = i*nc;
01646     matidx = idx*nc;
01647     idx1 = idx - nc;
01648     matidx1 = idx1*nc;
01649     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01650     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01651     smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01652     smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
01653     smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);
01654 }
01655
01656 for (i=nx; i<nxy; ++i) {
01657     idx = i*nc;
01658     idx1 = idx-nc;
01659     idx2 = idx-nxnc;
01660     matidx = idx*nc;
01661     matidx1 = idx1*nc;
01662     matidx2 = idx2*nc;
01663     smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01664     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01665     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01666     smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01667     smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
01668     smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);
01669 }
01670
01671 for (i=nxy; i<end3; ++i) {
01672     idx = i*nc;
01673     idx1 = idx-nc;
01674     idx2 = idx-nxnc;
01675     idx3 = idx-nxync;
01676     matidx = idx*nc;
01677     matidx1 = idx1*nc;
01678     matidx2 = idx2*nc;
01679     matidx3 = idx3*nc;
01680     smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
01681     smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01682     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01683     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01684     smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01685     smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
01686     smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);
01687 }
01688
01689 for (i=end3; i<end2; ++i) {
01690     idx = i*nc;
01691     idx1 = idx-nc;
01692     idx2 = idx-nxnc;
01693     idx3 = idx-nxync;
01694     matidx = idx*nc;
01695     matidx1 = idx1*nc;
01696     matidx2 = idx2*nc;
01697     matidx3 = idx3*nc;
01698     smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
01699     smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01700     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01701     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01702     smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01703     smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
01704 }
01705 }
01706

```

```

01707     for (i=end2; i<endl; ++i) {
01708         idx = i*nc;
01709         idx1 = idx-nc;
01710         idx2 = idx-nxnc;
01711         idx3 = idx-nxync;
01712         matidx = idx*nc;
01713         matidx1 = idx1*nc;
01714         matidx2 = idx2*nc;
01715         matidx3 = idx3*nc;
01716         smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
01717         smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01718         smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01719         smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01720         smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
01721     }
01722
01723     i=endl;
01724     idx = i*nc;
01725     idx1 = idx-nc;
01726     idx2 = idx-nxnc;
01727     idx3 = idx-nxync;
01728     matidx = idx*nc;
01729     matidx1 = idx1*nc;
01730     matidx2 = idx2*nc;
01731     matidx3 = idx3*nc;
01732     smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
01733     smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
01734     smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
01735     smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
01736
01737     return;
01738 }
01739
01740 static inline void str_spaAxy (const REAL    alpha,
01741                               const dSTRmat *A,
01742                               const REAL    *x,
01743                               REAL          *y)
01744 {
01745     // information of A
01746     INT ngrid = A->ngrid; // number of grids
01747     INT nc = A->nc;        // size of each block (number of components)
01748     INT nband = A->nband; // number of off-diag band
01749     INT *offsets = A->offsets; // offsets of the off-diagonals
01750     REAL *diag = A->diag; // Diagonal entries
01751     REAL **offdiag = A->offdiag; // Off-diagonal entries
01752
01753     // local variables
01754     INT k;
01755     INT block = 0;
01756     INT point = 0;
01757     INT band = 0;
01758     INT width = 0;
01759     INT size = nc*ngrid;
01760     INT nc2 = nc*nc;
01761     INT ncw = 0;
01762     INT start_data = 0;
01763     INT start_vecx = 0;
01764     INT start_vecy = 0;
01765     INT start_vect = 0;
01766     REAL beta = 0.0;
01767
01768     if (alpha == 0) {
01769         return; // nothing should be done
01770     }
01771
01772     beta = 1.0/alpha;
01773
01774     // y: = beta*y
01775     for (k = 0; k < size; ++k) {
01776         y[k] *= beta;
01777     }
01778
01779     // y: = y + A*x
01780     if (nc > 1) {
01781         // Deal with the diagonal band
01782         for (block = 0; block < ngrid; ++block) {
01783             start_data = nc2*block;
01784             start_vect = nc*block;
01785             blkcontr_str(start_data, start_vect, start_vect, nc, diag, x, y);
01786         }
01787     }

```

```

01802
01803 // Deal with the off-diagonal bands
01804 for (band = 0; band < nband; band++) {
01805     width = offsets[band];
01806     ncw = nc*width;
01807     if (width < 0) {
01808         for (block = 0; block < ngrid+width; ++block) {
01809             start_data = nc2*block;
01810             start_vecx = nc*block;
01811             start_vecy = start_vecx - ncw;
01812             blkcontr_str(start_data, start_vecx, start_vecy, nc, offdiag[band], x, y);
01813         }
01814     }
01815     else {
01816         for (block = 0; block < ngrid-width; ++block) {
01817             start_data = nc2*block;
01818             start_vecy = nc*block;
01819             start_vecx = start_vecy + ncw;
01820             blkcontr_str(start_data, start_vecx, start_vecy, nc, offdiag[band], x, y);
01821         }
01822     }
01823 }
01824 }
01825 else if (nc == 1) {
01826     // Deal with the diagonal band
01827     for (point = 0; point < ngrid; point++) {
01828         y[point] += diag[point]*x[point];
01829     }
01830
01831     // Deal with the off-diagonal bands
01832     for (band = 0; band < nband; band++) {
01833         width = offsets[band];
01834         if (width < 0) {
01835             for (point = 0; point < ngrid+width; point++) {
01836                 y[point-width] += offdiag[band][point]*x[point];
01837             }
01838         }
01839         else {
01840             for (point = 0; point < ngrid-width; point++) {
01841                 y[point] += offdiag[band][point]*x[point+width];
01842             }
01843         }
01844     }
01845 }
01846 else {
01847     printf("### WARNING: nc is illegal! %s\n", __FUNCTION__);
01848     return;
01849 }
01850
01851 // y: = alpha*y
01852 for (k = 0; k < size; ++k) {
01853     y[k] *= alpha;
01854 }
01855 }
01856
01857 /*-----*/
01858 /*--          End of File          --*/
01859 /*-----*/

```

9.97 BlaVector.c File Reference

BLAS1 operations for vectors.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_blas_dvec_axpy` (const `REAL` a, const `dvector` *x, `dvector` *y)

$$y = a*x + y$$
- void `fasp_blas_dvec_axpyz` (const `REAL` a, const `dvector` *x, const `dvector` *y, `dvector` *z)

$z = a*x + y$, z is a third vector (z is cleared)

- [REAL fasp_blas_dvec_norm1](#) (const [dvector](#) *x)
L1 norm of dvector x.
- [REAL fasp_blas_dvec_norm2](#) (const [dvector](#) *x)
L2 norm of dvector x.
- [REAL fasp_blas_dvec_norminf](#) (const [dvector](#) *x)
Linf norm of dvector x.
- [REAL fasp_blas_dvec_dotprod](#) (const [dvector](#) *x, const [dvector](#) *y)
Inner product of two vectors (x,y)
- [REAL fasp_blas_dvec_relerr](#) (const [dvector](#) *x, const [dvector](#) *y)
Relative difference between two dvector x and y.

9.97.1 Detailed Description

BLAS1 operations for vectors.

Note

This file contains Level-1 (Bla) functions. It requires: [AuxMessage.c](#), [AuxThreads.c](#), and [BlaArray.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [BlaVector.c](#).

9.97.2 Function Documentation

9.97.2.1 fasp_blas_dvec_axpy()

```
void fasp_blas_dvec_axpy (
    const REAL a,
    const dvector * x,
    dvector * y )
```

$y = a*x + y$

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to dvector x
<i>y</i>	Pointer to dvector y

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 41 of file [BlaVector.c](#).

9.97.2.2 fasp_blas_dvec_axpyz()

```
void fasp_blas_dvec_axpyz (
    const REAL a,
    const dvector * x,
    const dvector * y,
    dvector * z )
z = a*x + y, z is a third vector (z is cleared)
```

Parameters

<i>a</i>	REAL factor a
<i>x</i>	Pointer to dvector x
<i>y</i>	Pointer to dvector y
<i>z</i>	Pointer to dvector z

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 96 of file [BlaVector.c](#).

9.97.2.3 fasp_blas_dvec_dotprod()

```
REAL fasp_blas_dvec_dotprod (
    const dvector * x,
    const dvector * y )
Inner product of two vectors (x,y)
```

Parameters

<i>x</i>	Pointer to dvector x
<i>y</i>	Pointer to dvector y

Returns

Inner product

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012
Definition at line 236 of file [BlaVector.c](#).

9.97.2.4 fasp_blas_dvec_norm1()

```
REAL fasp_blas_dvec_norm1 (  
    const dvector * x )
```

L1 norm of dvector x.

Parameters

x	Pointer to dvector x
---	----------------------

Returns

L1 norm of x

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 130 of file [BlaVector.c](#).

9.97.2.5 fasp_blas_dvec_norm2()

```
REAL fasp_blas_dvec_norm2 (  
    const dvector * x )
```

L2 norm of dvector x.

Parameters

x	Pointer to dvector x
---	----------------------

Returns

L2 norm of x

Author

Chensong Zhang

Date

07/01/2009

Definition at line 170 of file [BlaVector.c](#).

9.97.2.6 fasp_blas_dvec_norminf()

```
REAL fasp_blas_dvec_norminf (  
    const dvector * x )
```

Linf norm of dvector x.

Parameters

<i>x</i>	Pointer to dvector x
----------	----------------------

Returns

L_{∞} norm of x

Author

Chensong Zhang

Date

07/01/2009

Definition at line 208 of file [BlaVector.c](#).

9.97.2.7 fasp_blas_dvec_relerr()

```
REAL fasp_blas_dvec_relerr (  
    const dvector * x,  
    const dvector * y )
```

Relative difference between two dvector x and y.

Parameters

<i>x</i>	Pointer to dvector x
<i>y</i>	Pointer to dvector y

Returns

Relative difference $\|x-y\|/\|x\|$

Author

Chensong Zhang

Date

07/01/2009

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/23/2012

Definition at line 278 of file [BlaVector.c](#).

9.98 BlaVector.c

[Go to the documentation of this file.](#)

```
00001  
00014 #include <math.h>  
00015  
00016 #ifdef _OPENMP  
00017 #include <omp.h>  
00018 #endif  
00019
```

```

00020 #include "fasp.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00041 void fasp_blas_dvec_axpy (const REAL    a,
00042                          const dvector *x,
00043                          dvector      *y)
00044 {
00045     const INT    m = x->row;
00046     const REAL *xpt = x->val;
00047     REAL      *ypt = y->val;
00048
00049     SHORT use_omp = FALSE;
00050     INT    i;
00051
00052 #ifdef _OPENMP
00053     INT myid, mybegin, myend, nthreads;
00054     if ( m > OPENMP_HOLDS ) {
00055         use_omp = TRUE;
00056         nthreads = fasp_get_num_threads();
00057     }
00058 #endif
00059
00060     if ( y->row != m ) {
00061         printf("### ERROR: Vectors have different dimensions!\n");
00062         fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00063     }
00064
00065     if (use_omp) {
00066 #ifdef _OPENMP
00067 #pragma omp parallel private(myid,mybegin,myend,i) num_threads(nthreads)
00068     {
00069         myid = omp_get_thread_num();
00070         fasp_get_start_end (myid, nthreads, m, &mybegin, &myend);
00071         for ( i = mybegin; i < myend; ++i ) ypt[i] += a*xpt[i];
00072     }
00073 #endif
00074     }
00075     else {
00076         for ( i = 0; i < m; ++i ) ypt[i] += a*xpt[i];
00077     }
00078 }
00079
00096 void fasp_blas_dvec_axpyz (const REAL    a,
00097                           const dvector *x,
00098                           const dvector *y,
00099                           dvector      *z)
00100 {
00101     const INT    m = x->row;
00102     const REAL *xpt = x->val, *ypt = y->val;
00103     REAL      *zpt = z->val;
00104
00105     if ( y->row != m ) {
00106         printf("### ERROR: Vectors have different dimensions!\n");
00107         fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
00108     }
00109
00110     z->row = m;
00111
00112     memcpy (zpt, ypt, m*sizeof(REAL));
00113     fasp_blas_darray_axpy (m, a, xpt, zpt);
00114 }
00115
00130 REAL fasp_blas_dvec_norm1 (const dvector *x)
00131 {
00132     const INT    length = x->row;
00133     const REAL *xpt = x->val;
00134
00135     register REAL onenorm = 0.0;
00136     SHORT use_omp = FALSE;
00137     INT    i;
00138
00139 #ifdef _OPENMP
00140     if ( length > OPENMP_HOLDS ) {
00141         use_omp = TRUE;
00142     }
00143 #endif
00144 }

```

```

00145     if ( use_omp ) {
00146 #ifdef _OPENMP
00147 #pragma omp parallel for reduction(+:onenorm) private(i)
00148 #endif
00149         for ( i = 0; i < length; ++i ) onenorm += ABS(xpt[i]);
00150     }
00151     else {
00152         for ( i = 0; i < length; ++i ) onenorm += ABS(xpt[i]);
00153     }
00154     return onenorm;
00155 }
00156
00157
00170 REAL fasp_blas_dvec_norm2 (const dvector *x)
00171 {
00172     const INT    length = x->row;
00173     const REAL *xpt = x->val;
00174
00175     register REAL twonorm = 0.0;
00176     SHORT use_omp = FALSE;
00177     INT i;
00178
00179 #ifdef _OPENMP
00180     if ( length > OPENMP_HOLDS ) use_omp = TRUE;
00181 #endif
00182
00183     if ( use_omp ) {
00184 #ifdef _OPENMP
00185 #pragma omp parallel for reduction(+:twonorm) private(i)
00186 #endif
00187         for ( i = 0; i < length; ++i ) twonorm += xpt[i]*xpt[i];
00188     }
00189     else {
00190         for ( i = 0; i < length; ++i ) twonorm += xpt[i]*xpt[i];
00191     }
00192
00193     return sqrt(twonorm);
00194 }
00195
00208 REAL fasp_blas_dvec_norminf (const dvector *x)
00209 {
00210     const INT    length=x->row;
00211     const REAL *xpt=x->val;
00212
00213     register REAL infnorm = 0.0;
00214     register INT i;
00215
00216     for ( i = 0; i < length; ++i ) infnorm = MAX(infnorm, ABS(xpt[i]));
00217
00218     return infnorm;
00219 }
00220
00236 REAL fasp_blas_dvec_dotprod (const dvector *x,
00237                             const dvector *y)
00238 {
00239     const INT    length = x->row;
00240     const REAL *xpt = x->val, *ypt = y->val;
00241
00242     register REAL value = 0.0;
00243     SHORT use_omp = FALSE;
00244     INT i;
00245
00246 #ifdef _OPENMP
00247     if ( length > OPENMP_HOLDS ) use_omp = TRUE;
00248 #endif
00249
00250     if (use_omp) {
00251 #ifdef _OPENMP
00252 #pragma omp parallel for reduction(+:value) private(i)
00253 #endif
00254         for ( i = 0; i < length; ++i ) value += xpt[i] * ypt[i];
00255     }
00256     else {
00257         for ( i = 0; i < length; ++i ) value += xpt[i] * ypt[i];
00258     }
00259
00260     return value;
00261 }
00262
00278 REAL fasp_blas_dvec_relerr (const dvector *x,
00279                             const dvector *y)

```

```

00280 {
00281     const INT    length = x->row;
00282     const REAL *xpt = x->val, *ypt = y->val;
00283
00284     SHORT use_openmp = FALSE;
00285     REAL diff = 0.0, temp = 0.0;
00286     INT    i;
00287
00288     if ( length != y->row ) {
00289         printf("### ERROR: Vectors have different dimensions!\n");
00290         fasp_chkerf(ERROR_DATA_STRUCTURE, __FUNCTION__);
00291     }
00292
00293 #ifdef _OPENMP
00294     if ( length > OPENMP_HOLDS ) use_openmp = TRUE;
00295 #endif
00296
00297     if ( use_openmp ) {
00298 #ifdef _OPENMP
00299 #pragma omp parallel for reduction(+:temp,diff) private(i)
00300 #endif
00301         for ( i = 0; i < length; ++i ) {
00302             temp += xpt[i]*xpt[i];
00303             diff += pow(xpt[i]-ypt[i],2);
00304         }
00305     }
00306     else {
00307         for ( i = 0; i < length; ++i ) {
00308             temp += xpt[i]*xpt[i];
00309             diff += pow(xpt[i]-ypt[i],2);
00310         }
00311     }
00312
00313     return sqrt(diff/temp);
00314 }
00315
00316 /*-----*/
00317 /*--      End of File      --*/
00318 /*-----*/

```

9.99 ItrSmootherBSR.c File Reference

Smoothers for [dBSRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_smoother_dbsr_jacobi](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u)
Jacobi relaxation.
- void [fasp_smoother_dbsr_jacobi_setup](#) ([dBSRmat](#) *A, [REAL](#) *diaginv)
Setup for jacobi relaxation, fetch the diagonal sub-block matrixes and make them inverse first.
- void [fasp_smoother_dbsr_jacobi1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)
Jacobi relaxation.
- void [fasp_smoother_dbsr_gs](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [INT](#) order, [INT](#) *mark)
Gauss-Seidel relaxation.
- void [fasp_smoother_dbsr_gs1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [INT](#) order, [INT](#) *mark, [REAL](#) *diaginv)
Gauss-Seidel relaxation.
- void [fasp_smoother_dbsr_gs_ascend](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)
Gauss-Seidel relaxation in the ascending order.
- void [fasp_smoother_dbsr_gs_ascend1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u)
Gauss-Seidel relaxation in the ascending order.

- void [fasp_smoother_dbsr_gs_descend](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)
Gauss-Seidel relaxation in the descending order.
- void [fasp_smoother_dbsr_gs_descend1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u)
Gauss-Seidel relaxation in the descending order.
- void [fasp_smoother_dbsr_gs_order1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark)
Gauss-Seidel relaxation in the user-defined order.
- void [fasp_smoother_dbsr_gs_order2](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [INT](#) *mark, [REAL](#) *work)
Gauss-Seidel relaxation in the user-defined order.
- void [fasp_smoother_dbsr_sor](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [INT](#) order, [INT](#) *mark, [REAL](#) weight)
SOR relaxation.
- void [fasp_smoother_dbsr_sor1](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [INT](#) order, [INT](#) *mark, [REAL](#) *diaginv, [REAL](#) weight)
SOR relaxation.
- void [fasp_smoother_dbsr_sor_ascend](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [REAL](#) weight)
SOR relaxation in the ascending order.
- void [fasp_smoother_dbsr_sor_descend](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [REAL](#) weight)
SOR relaxation in the descending order.
- void [fasp_smoother_dbsr_sor_order](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark, [REAL](#) weight)
SOR relaxation in the user-defined order.
- void [fasp_smoother_dbsr_ilu](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, void *data)
ILU method as the smoother in solving $Au=b$ with multigrid method.

Variables

- [REAL](#) [ilu_solve_time](#) = 0.0

9.99.1 Detailed Description

Smoothers for [dBSRmat](#) matrices.

Note

This file contains Level-2 (ltr) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), [AuxTiming.c](#), [BlaSmallMatInv.c](#), [BlaSmallMat.c](#), [BlaArray.c](#), [BlaSpmvBSR.c](#), and [PreBSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

// TODO: Need to optimize routines here! –Chensong

Definition in file [ltrSmootherBSR.c](#).

9.99.2 Function Documentation

9.99.2.1 fasp_smoother_dbsr_gs()

```
void fasp_smoother_dbsr_gs (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    INT order,
    INT * mark )
```

Gauss-Seidel relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending order DESCEND 21: in descending order If mark != NULL: in the user-defined order
<i>mark</i>	Pointer to NULL or to the user-defined ordering

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 08/03/2012

Definition at line [428](#) of file [ltrSmootherBSR.c](#).

9.99.2.2 fasp_smoother_dbsr_gs1()

```
void fasp_smoother_dbsr_gs1 (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    INT order,
    INT * mark,
    REAL * diaginv )
```

Gauss-Seidel relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending order DESCEND 21: in descending order If mark != NULL: in the user-defined order
<i>mark</i>	Pointer to NULL or to the user-defined ordering
<i>diaginv</i>	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 545 of file [ltrSmootherBSR.c](#).**9.99.2.3 fasp_smoother_dbsr_gs_ascend()**

```
void fasp_smoother_dbsr_gs_ascend (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )
```

Gauss-Seidel relaxation in the ascending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 582 of file [ltrSmootherBSR.c](#).**9.99.2.4 fasp_smoother_dbsr_gs_ascend1()**

```
void fasp_smoother_dbsr_gs_ascend1 (
    dBSRmat * A,
    dvector * b,
    dvector * u )
```

Gauss-Seidel relaxation in the ascending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)

Author

Xiaozhe Hu

Date

01/01/2014

Note

The only difference between the functions 'fasp_smoother_dbsr_gs_ascend1' and 'fasp_smoother_dbsr_gs_ascend' is that we don't have to multiply by the inverses of the diagonal blocks in each ROW since matrix A has been such scaled that all the diagonal blocks become identity matrices.

Definition at line 655 of file [ltrSmootherBSR.c](#).

9.99.2.5 fasp_smoother_dbsr_gs_descend()

```
void fasp_smoother_dbsr_gs_descend (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )
```

Gauss-Seidel relaxation in the descending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 724 of file [ltrSmootherBSR.c](#).

9.99.2.6 fasp_smoother_dbsr_gs_descend1()

```
void fasp_smoother_dbsr_gs_descend1 (
    dBSRmat * A,
    dvector * b,
    dvector * u )
```

Gauss-Seidel relaxation in the descending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)

Author

Xiaozhe Hu

Date

01/01/2014

Note

The only difference between the functions 'fasp_smoother_dbsr_gs_ascend1' and 'fasp_smoother_dbsr_gs_ascend' is that we don't have to multiply by the inverses of the diagonal blocks in each ROW since matrix A has been such scaled that all the diagonal blocks become identity matrices.

Definition at line [798](#) of file [ltrSmootherBSR.c](#).

9.99.2.7 fasp_smoother_dbsr_gs_order1()

```
void fasp_smoother_dbsr_gs_order1 (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginvs,
    INT * mark )
```

Gauss-Seidel relaxation in the user-defined order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A
<i>mark</i>	Pointer to the user-defined ordering

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line [868](#) of file [ltrSmootherBSR.c](#).

9.99.2.8 fasp_smoother_dbsr_gs_order2()

```
void fasp_smoother_dbsr_gs_order2 (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    INT * mark,
    REAL * work )
```

Gauss-Seidel relaxation in the user-defined order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>mark</i>	Pointer to the user-defined ordering
<i>work</i>	Work temp array

Author

Zhiyang Zhou

Date

2010/11/08

Note

The only difference between the functions 'fasp_smoother_dbsr_gs_order2' and 'fasp_smoother_dbsr_gs_order1' lies in that we don't have to multiply by the inverses of the diagonal blocks in each ROW since matrix A has been such scaled that all the diagonal blocks become identity matrices.

Definition at line [946](#) of file [ltrSmootherBSR.c](#).

9.99.2.9 fasp_smoother_dbsr_ilu()

```
void fasp_smoother_dbsr_ilu (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    void * data )
```

ILU method as the smoother in solving $Au=b$ with multigrid method.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>x</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>data</i>	Pointer to user defined data

Author

Zhiyang Zhou, Zheng Li

Date

2010/10/25

NOTE: Add multi-threads parallel ILU block by Zheng Li 12/04/2016. form residual $zr = b - A x$
solve LU $z=zr$

$x=x+z$

Definition at line [1566](#) of file [ltrSmootherBSR.c](#).

9.99.2.10 fasp_smoother_dbsr_jacobi()

```
void fasp_smoother_dbsr_jacobi (
    dBSRmat * A,
    dvector * b,
    dvector * u )
```

Jacobi relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 08/02/2012

Definition at line [59](#) of file [ltrSmootherBSR.c](#).

9.99.2.11 fasp_smoother_dbsr_jacobi1()

```
void fasp_smoother_dbsr_jacobi1 (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )
```

Jacobi relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 08/03/2012

Definition at line [274](#) of file [ltrSmootherBSR.c](#).

9.99.2.12 fasp_smoother_dbsr_jacobi_setup()

```
void fasp_smoother_dbsr_jacobi_setup (
    dBSRmat * A,
    REAL * diaginvs )
```

Setup for jacobi relaxation, fetch the diagonal sub-block matrixes and make them inverse first.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>diaginv</i>	Inverse of the diagonal entries

Author

Zhiyang Zhou

Date

10/25/2010

Modified by Chunsheng Feng, Zheng Li on 08/02/2012

Definition at line [168](#) of file [ltrSmootherBSR.c](#).

9.99.2.13 fasp_smoother_dbsr_sor()

```
void fasp_smoother_dbsr_sor (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    INT order,
    INT * mark,
    REAL weight )
```

SOR relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending order DESCEND 21: in descending order If mark != NULL: in the user-defined order
<i>mark</i>	Pointer to NULL or to the user-defined ordering
<i>weight</i>	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 08/03/2012

Definition at line 1023 of file [ltrSmootherBSR.c](#).**9.99.2.14 fasp_smoother_dbsr_sor1()**

```
void fasp_smoother_dbsr_sor1 (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    INT order,
    INT * mark,
    REAL * diaginvs,
    REAL weight )
```

SOR relaxation.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending order DESCEND 21: in descending order If mark != NULL: in the user-defined order
<i>mark</i>	Pointer to NULL or to the user-defined ordering
<i>diaginv</i>	Inverses for all the diagonal blocks of A
<i>weight</i>	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 1146 of file [ltrSmootherBSR.c](#).**9.99.2.15 fasp_smoother_dbsr_sor_ascend()**

```
void fasp_smoother_dbsr_sor_ascend (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginvs,
    REAL weight )
```

SOR relaxation in the ascending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
----------	---

Parameters

<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A
<i>weight</i>	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 2012/09/04

Definition at line 1187 of file [ltrSmootherBSR.c](#).**9.99.2.16 fasp_smoother_dbsr_sor_descend()**

```
void fasp_smoother_dbsr_sor_descend (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    REAL weight )
```

SOR relaxation in the descending order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A
<i>weight</i>	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 2012/09/04

Definition at line 1310 of file [ltrSmootherBSR.c](#).**9.99.2.17 fasp_smoother_dbsr_sor_order()**

```
void fasp_smoother_dbsr_sor_order (
    dBSRmat * A,
```

```

dvector * b,
dvector * u,
REAL * diaginv,
INT * mark,
REAL weight )

```

SOR relaxation in the user-defined order.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>diaginv</i>	Inverses for all the diagonal blocks of A
<i>mark</i>	Pointer to the user-defined ordering
<i>weight</i>	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Modified by Chunsheng Feng, Zheng Li on 2012/09/04

Definition at line [1438](#) of file [ltrSmootherBSR.c](#).

9.99.3 Variable Documentation

9.99.3.1 ilu_solve_time

```
REAL ilu_solve_time = 0.0
```

ILU time for the SOLVE phase

Definition at line [39](#) of file [ltrSmootherBSR.c](#).

9.100 ltrSmootherBSR.c

[Go to the documentation of this file.](#)

```

00001
00017 #include <math.h>
00018
00019 #ifdef _OPENMP
00020 #include <omp.h>
00021 #endif
00022
00023 #include "fasp.h"
00024 #include "fasp_funcs.h"
00025
00026 /*-----*/
00027 /*--  Declare Private Functions  --*/
00028 /*-----*/
00029
00030 #ifdef _OPENMP
00031
00032 #if ILU_MC_OMP
00033 static inline void perm (const INT, const INT, const REAL *, const INT *, REAL *);
00034 static inline void invperm (const INT, const INT, const REAL *, const INT *, REAL *);

```

```

00035 #endif
00036
00037 #endif
00038
00039 REAL ilu_solve_time = 0.0;
00041 /*-----*/
00042 /*--      Public Functions      --*/
00043 /*-----*/
00044
00059 void fasp_smoother_dbsr_jacobi (dBSRmat *A,
00060                               dvector *b,
00061                               dvector *u)
00062 {
00063     // members of A
00064     const INT    ROW = A->ROW;
00065     const INT    nb  = A->nb;
00066     const INT    nb2 = nb*nb;
00067     const INT    size = ROW*nb2;
00068     const INT    *IA  = A->IA;
00069     const INT    *JA  = A->JA;
00070     const REAL    *val = A->val;
00071
00072     // local variables
00073     INT    i,k;
00074     SHORT nthreads = 1, use_omp = FALSE;
00075     REAL *diaginv = (REAL *)fasp_mem_malloc(size, sizeof(REAL));
00076
00077 #ifdef _OPENMP
00078     if ( ROW > OPENMP_HOLDS ) {
00079         use_omp = TRUE;
00080         nthreads = fasp_get_num_threads();
00081     }
00082 #endif
00083
00084     // get all the diagonal sub-blocks
00085     if (use_omp) {
00086         INT mybegin,myend,myid;
00087 #ifdef _OPENMP
00088         #pragma omp parallel for private(myid,mybegin,myend,i,k)
00089 #endif
00090         for (myid=0; myid<nthreads; myid++) {
00091             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00092             for (i=mybegin; i<myend; i++) {
00093                 for (k=IA[i]; k<IA[i+1]; ++k)
00094                     if (JA[k] == i)
00095                         memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00096             }
00097         }
00098     }
00099     else {
00100         for (i = 0; i < ROW; ++i) {
00101             for (k = IA[i]; k < IA[i+1]; ++k) {
00102                 if (JA[k] == i)
00103                     memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00104             }
00105         }
00106     }
00107
00108     // compute the inverses of all the diagonal sub-blocks
00109     if (nb > 1) {
00110         if (use_omp) {
00111             INT mybegin,myend,myid;
00112 #ifdef _OPENMP
00113             #pragma omp parallel for private(myid,mybegin,myend,i)
00114 #endif
00115             for (myid=0; myid<nthreads; myid++) {
00116                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00117                 for (i=mybegin; i<myend; i++) {
00118                     fasp_smat_inv(diaginv+i*nb2, nb);
00119                 }
00120             }
00121         }
00122         else {
00123             for (i = 0; i < ROW; ++i) {
00124                 fasp_smat_inv(diaginv+i*nb2, nb);
00125             }
00126         }
00127     }
00128     else {
00129         if (use_omp) {
00130             INT mybegin, myend, myid;

```

```

00131 #ifdef _OPENMP
00132 #pragma omp parallel for private(myid,mybegin,myend,i)
00133 #endif
00134         for (myid=0; myid<nthreads; myid++) {
00135             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00136             for (i=mybegin; i<myend; i++) {
00137                 diaginv[i] = 1.0 / diaginv[i];
00138             }
00139         }
00140     }
00141     else {
00142         for (i = 0; i < ROW; ++i) {
00143             // zero-diagonal should be tested previously
00144             diaginv[i] = 1.0 / diaginv[i];
00145         }
00146     }
00147 }
00148
00149 fasp_smoother_dbsr_jacobi(A, b, u, diaginv);
00150
00151 fasp_mem_free(diaginv); diaginv = NULL;
00152 }
00153
00168 void fasp_smoother_dbsr_jacobi_setup (dBSRmat *A,
00169                                     REAL *diaginv)
00170 {
00171     // members of A
00172     const INT ROW = A->ROW;
00173     const INT nb = A->nb;
00174     const INT nb2 = nb*nb;
00175     const INT *IA = A->IA;
00176     const INT *JA = A->JA;
00177     const REAL *val = A->val;
00178
00179     // local variables
00180     INT i,k;
00181
00182     SHORT nthreads = 1, use_omp = FALSE;
00183
00184 #ifdef _OPENMP
00185     if ( ROW > OPENMP_HOLDS ) {
00186         use_omp = TRUE;
00187         nthreads = fasp_get_num_threads();
00188     }
00189 #endif
00190
00191     // get all the diagonal sub-blocks
00192     if (use_omp) {
00193         INT mybegin,myend,myid;
00194 #ifdef _OPENMP
00195         #pragma omp parallel for private(myid,mybegin,myend,i,k)
00196 #endif
00197         for (myid=0; myid<nthreads; myid++) {
00198             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00199             for (i=mybegin; i<myend; i++) {
00200                 for (k=IA[i]; k<IA[i+1]; ++k)
00201                     if (JA[k] == i)
00202                         memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00203             }
00204         }
00205     }
00206     else {
00207         for (i = 0; i < ROW; ++i) {
00208             for (k = IA[i]; k < IA[i+1]; ++k) {
00209                 if (JA[k] == i)
00210                     memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00211             }
00212         }
00213     }
00214
00215     // compute the inverses of all the diagonal sub-blocks
00216     if (nb > 1) {
00217         if (use_omp) {
00218             INT mybegin,myend,myid;
00219 #ifdef _OPENMP
00220             #pragma omp parallel for private(myid,mybegin,myend,i)
00221 #endif
00222             for (myid=0; myid<nthreads; myid++) {
00223                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00224                 for (i=mybegin; i<myend; i++) {
00225                     fasp_smat_inv(diaginv+i*nb2, nb);

```

```

00226     }
00227     }
00228     }
00229     else {
00230         for (i = 0; i < ROW; ++i) {
00231             fasp_smat_inv(diaginv+i*nb2, nb);
00232         }
00233     }
00234 }
00235 else {
00236     if (use_omp) {
00237         INT mybegin, myend, myid;
00238 #ifdef _OPENMP
00239 #pragma omp parallel for private(myid,mybegin,myend,i)
00240 #endif
00241         for (myid=0; myid<nthreads; myid++) {
00242             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00243             for (i=mybegin; i<myend; i++) {
00244                 diaginv[i] = 1.0 / diaginv[i];
00245             }
00246         }
00247     }
00248     else {
00249         for (i = 0; i < ROW; ++i) {
00250             // zero-diagonal should be tested previously
00251             diaginv[i] = 1.0 / diaginv[i];
00252         }
00253     }
00254 }
00255 }
00256 }
00257
00274 void fasp_smoother_dbsr_jacobi1 (dBSRmat *A,
00275                                dvector *b,
00276                                dvector *u,
00277                                REAL *diaginv)
00278 {
00279     // members of A
00280     const INT ROW = A->ROW;
00281     const INT nb = A->nb;
00282     const INT nb2 = nb*nb;
00283     const INT size = ROW*nb;
00284     const INT *IA = A->IA;
00285     const INT *JA = A->JA;
00286     REAL *val = A->val;
00287
00288     SHORT nthreads = 1, use_omp = FALSE;
00289
00290 #ifdef _OPENMP
00291     if ( ROW > OPENMP_HOLDS ) {
00292         use_omp = TRUE;
00293         nthreads = fasp_get_num_threads();
00294     }
00295 #endif
00296
00297     // values of dvector b and u
00298     REAL *b_val = b->val;
00299     REAL *u_val = u->val;
00300
00301     // auxiliary array
00302     REAL *b_tmp = NULL;
00303
00304     // local variables
00305     INT i,j,k;
00306     INT pb;
00307
00308     // b_tmp = b_val
00309     b_tmp = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00310     memcpy(b_tmp, b_val, size*sizeof(REAL));
00311
00312     // No need to assign the smoothing order since the result doesn't depend on it
00313     if (nb == 1) {
00314         if (use_omp) {
00315             INT mybegin, myend, myid;
00316 #ifdef _OPENMP
00317 #pragma omp parallel for private(myid,mybegin,myend,i,j,k)
00318 #endif
00319             for (myid=0; myid<nthreads; myid++) {
00320                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00321                 for (i=mybegin; i<myend; i++) {
00322                     for (k = IA[i]; k < IA[i+1]; ++k) {

```

```

00323         j = JA[k];
00324         if (j != i)
00325             b_tmp[i] -= val[k]*u_val[j];
00326     }
00327 }
00328 }
00329 #ifdef _OPENMP
00330 #pragma omp parallel for private(myid,mybegin,myend,i)
00331 #endif
00332     for (myid=0; myid<nthreads; myid++) {
00333         fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00334         for (i=mybegin; i<myend; i++) {
00335             u_val[i] = b_tmp[i]*diaginv[i];
00336         }
00337     }
00338 }
00339 else {
00340     for (i = 0; i < ROW; ++i) {
00341         for (k = IA[i]; k < IA[i+1]; ++k) {
00342             j = JA[k];
00343             if (j != i)
00344                 b_tmp[i] -= val[k]*u_val[j];
00345         }
00346     }
00347     for (i = 0; i < ROW; ++i) {
00348         u_val[i] = b_tmp[i]*diaginv[i];
00349     }
00350 }
00351
00352     fasp_mem_free(b_tmp); b_tmp = NULL;
00353 }
00354 else if (nb > 1) {
00355     if (use_omp) {
00356         INT mybegin, myend, myid;
00357 #ifdef _OPENMP
00358 #pragma omp parallel for private(myid,mybegin,myend,i,pb,k,j)
00359 #endif
00360         for (myid=0; myid<nthreads; myid++) {
00361             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00362             for (i=mybegin; i<myend; i++) {
00363                 pb = i*nb;
00364                 for (k = IA[i]; k < IA[i+1]; ++k) {
00365                     j = JA[k];
00366                     if (j != i)
00367                         fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+pb, nb);
00368                 }
00369             }
00370         }
00371 #ifdef _OPENMP
00372 #pragma omp parallel for private(myid,mybegin,myend,i,pb)
00373 #endif
00374         for (myid=0; myid<nthreads; myid++) {
00375             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00376             for (i=mybegin; i<myend; i++) {
00377                 pb = i*nb;
00378                 fasp_blas_smat_mnv(diaginv+nb2*i, b_tmp+pb, u_val+pb, nb);
00379             }
00380         }
00381     }
00382     else {
00383         for (i = 0; i < ROW; ++i) {
00384             pb = i*nb;
00385             for (k = IA[i]; k < IA[i+1]; ++k) {
00386                 j = JA[k];
00387                 if (j != i)
00388                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+pb, nb);
00389             }
00390         }
00391     }
00392     for (i = 0; i < ROW; ++i) {
00393         pb = i*nb;
00394         fasp_blas_smat_mnv(diaginv+nb2*i, b_tmp+pb, u_val+pb, nb);
00395     }
00396 }
00397
00398     fasp_mem_free(b_tmp); b_tmp = NULL;
00399 }
00400 else {
00401     printf("### ERROR: nb is illegal! [%s:%d]\n", __FILE__, __LINE__);
00402     fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00403 }

```

```

00404
00405 }
00406
00428 void fasp_smoother_dbsr_gs (dBSRmat *A,
00429                             dvector *b,
00430                             dvector *u,
00431                             INT      order,
00432                             INT      *mark )
00433 {
00434     // members of A
00435     const INT    ROW = A->ROW;
00436     const INT    nb  = A->nb;
00437     const INT    nb2 = nb*nb;
00438     const INT    size = ROW*nb2;
00439     const INT    *IA  = A->IA;
00440     const INT    *JA  = A->JA;
00441     const REAL    *val = A->val;
00442
00443     // local variables
00444     INT    i,k;
00445     SHORT  nthreads = 1, use_omp = FALSE;
00446     REAL *diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00447
00448 #ifdef _OPENMP
00449     if ( ROW > OPENMP_HOLDS ) {
00450         use_omp = TRUE;
00451         nthreads = fasp_get_num_threads();
00452     }
00453 #endif
00454
00455     // get all the diagonal sub-blocks
00456     if (use_omp) {
00457         INT mybegin,myend,myid;
00458 #ifdef _OPENMP
00459 #pragma omp parallel for private(myid,mybegin,myend,i,k)
00460 #endif
00461         for (myid=0; myid<nthreads; myid++) {
00462             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00463             for (i=mybegin; i<myend; i++) {
00464                 for (k=IA[i]; k<IA[i+1]; ++k)
00465                     if (JA[k] == i)
00466                         memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00467             }
00468         }
00469     }
00470     else {
00471         for (i = 0; i < ROW; ++i) {
00472             for (k = IA[i]; k < IA[i+1]; ++k) {
00473                 if (JA[k] == i)
00474                     memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
00475             }
00476         }
00477     }
00478
00479     // compute the inverses of all the diagonal sub-blocks
00480     if (nb > 1) {
00481         if (use_omp) {
00482             INT mybegin,myend,myid;
00483 #ifdef _OPENMP
00484 #pragma omp parallel for private(myid,mybegin,myend,i)
00485 #endif
00486             for (myid=0; myid<nthreads; myid++) {
00487                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00488                 for (i=mybegin; i<myend; i++) {
00489                     fasp_smat_inv(diaginv+i*nb2, nb);
00490                 }
00491             }
00492         }
00493         else {
00494             for (i = 0; i < ROW; ++i) {
00495                 fasp_smat_inv(diaginv+i*nb2, nb);
00496             }
00497         }
00498     }
00499     else {
00500         if (use_omp) {
00501             INT mybegin, myend, myid;
00502 #ifdef _OPENMP
00503 #pragma omp parallel for private(myid,mybegin,myend,i)
00504 #endif
00505             for (myid=0; myid<nthreads; myid++) {

```

```

00506         fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00507         for (i=mybegin; i<myend; i++) {
00508             diaginv[i] = 1.0 / diaginv[i];
00509         }
00510     }
00511 }
00512 else {
00513     for (i = 0; i < ROW; ++i) {
00514         // zero-diagonal should be tested previously
00515         diaginv[i] = 1.0 / diaginv[i];
00516     }
00517 }
00518 }
00519
00520 fasp_smoother_dbsr_gsl(A, b, u, order, mark, diaginv);
00521
00522 fasp_mem_free(diaginv); diaginv = NULL;
00523 }
00524
00545 void fasp_smoother_dbsr_gsl (dBSRmat *A,
00546                             dvector *b,
00547                             dvector *u,
00548                             INT     order,
00549                             INT     *mark,
00550                             REAL    *diaginv)
00551 {
00552     if (!mark) {
00553         if (order == ASCEND) // smooth ascendingly
00554         {
00555             fasp_smoother_dbsr_gs_ascend(A, b, u, diaginv);
00556         }
00557         else if (order == DESCEND) // smooth descendingly
00558         {
00559             fasp_smoother_dbsr_gs_descend(A, b, u, diaginv);
00560         }
00561     }
00562     // smooth according to the order 'mark' defined by user
00563     else {
00564         fasp_smoother_dbsr_gs_order1(A, b, u, diaginv, mark);
00565     }
00566 }
00567
00582 void fasp_smoother_dbsr_gs_ascend (dBSRmat *A,
00583                                    dvector *b,
00584                                    dvector *u,
00585                                    REAL    *diaginv)
00586 {
00587     // members of A
00588     const INT    ROW = A->ROW;
00589     const INT    nb  = A->nb;
00590     const INT    nb2 = nb*nb;
00591     const INT    *IA  = A->IA;
00592     const INT    *JA  = A->JA;
00593     REAL         *val = A->val;
00594
00595     // values of dvector b and u
00596     REAL *b_val = b->val;
00597     REAL *u_val = u->val;
00598
00599     // local variables
00600     INT    i, j, k;
00601     INT    pb;
00602     REAL    rhs = 0.0;
00603
00604     if (nb == 1) {
00605         for (i = 0; i < ROW; ++i) {
00606             rhs = b_val[i];
00607             for (k = IA[i]; k < IA[i+1]; ++k) {
00608                 j = JA[k];
00609                 if (j != i)
00610                     rhs -= val[k]*u_val[j];
00611             }
00612             u_val[i] = rhs*diaginv[i];
00613         }
00614     }
00615     else if (nb > 1) {
00616         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
00617
00618         for (i = 0; i < ROW; ++i) {
00619             pb = i*nb;
00620             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));

```



```

00621         for (k = IA[i]; k < IA[i+1]; ++k) {
00622             j = JA[k];
00623             if (j != i)
00624                 fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00625         }
00626         fasp_blas_smat_m xv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
00627     }
00628
00629     fasp_mem_free(b_tmp); b_tmp = NULL;
00630 }
00631 else {
00632     printf("### ERROR: nb is illegal! [%s:%d]\n", __FILE__, __LINE__);
00633     fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00634 }
00635
00636 }
00637
00655 void fasp_smoother_dbsr_gs_ascend1 (dBSRmat *A,
00656                                     dvector *b,
00657                                     dvector *u)
00658 {
00659     // members of A
00660     const INT ROW = A->ROW;
00661     const INT nb = A->nb;
00662     const INT nb2 = nb*nb;
00663     const INT *IA = A->IA;
00664     const INT *JA = A->JA;
00665     REAL *val = A->val;
00666
00667     // values of dvector b and u
00668     REAL *b_val = b->val;
00669     REAL *u_val = u->val;
00670
00671     // local variables
00672     INT i, j, k;
00673     INT pb;
00674     REAL rhs = 0.0;
00675
00676     if (nb == 1) {
00677         for (i = 0; i < ROW; ++i) {
00678             rhs = b_val[i];
00679             for (k = IA[i]; k < IA[i+1]; ++k) {
00680                 j = JA[k];
00681                 if (j != i)
00682                     rhs -= val[k]*u_val[j];
00683             }
00684             u_val[i] = rhs;
00685         }
00686     }
00687     else if (nb > 1) {
00688         REAL *b_tmp = (REAL *)fasp_mem_malloc(nb, sizeof(REAL));
00689
00690         for (i = 0; i < ROW; ++i) {
00691             pb = i*nb;
00692             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
00693             for (k = IA[i]; k < IA[i+1]; ++k) {
00694                 j = JA[k];
00695                 if (j != i)
00696                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00697             }
00698             memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
00699         }
00700
00701         fasp_mem_free(b_tmp); b_tmp = NULL;
00702     }
00703     else {
00704         printf("### ERROR: nb is illegal! [%s:%d]\n", __FILE__, __LINE__);
00705         fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00706     }
00707 }
00708 }
00709
00724 void fasp_smoother_dbsr_gs_descend (dBSRmat *A,
00725                                     dvector *b,
00726                                     dvector *u,
00727                                     REAL *diaginv )
00728 {
00729     // members of A
00730     const INT ROW = A->ROW;
00731     const INT nb = A->nb;
00732     const INT nb2 = nb*nb;

```

```

00733     const INT      *IA  = A->IA;
00734     const INT      *JA  = A->JA;
00735     REAL           *val  = A->val;
00736
00737     // values of dvector b and u
00738     REAL *b_val = b->val;
00739     REAL *u_val = u->val;
00740
00741     // local variables
00742     INT i,j,k;
00743     INT pb;
00744     REAL rhs = 0.0;
00745
00746     if (nb == 1) {
00747         for (i = ROW-1; i >= 0; i--) {
00748             rhs = b_val[i];
00749             for (k = IA[i]; k < IA[i+1]; ++k) {
00750                 j = JA[k];
00751                 if (j != i)
00752                     rhs -= val[k]*u_val[j];
00753             }
00754             u_val[i] = rhs*diaginv[i];
00755         }
00756     }
00757     else if (nb > 1) {
00758         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
00759
00760         for (i = ROW-1; i >= 0; i--) {
00761             pb = i*nb;
00762             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
00763             for (k = IA[i]; k < IA[i+1]; ++k) {
00764                 j = JA[k];
00765                 if (j != i)
00766                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00767             }
00768             fasp_blas_smat_m xv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
00769         }
00770
00771         fasp_mem_free(b_tmp); b_tmp = NULL;
00772     }
00773     else {
00774         printf("### ERROR: nb is illegal! [%s:%d]\n", __FILE__, __LINE__);
00775         fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00776     }
00777 }
00778
00779 void fasp_smoother_dbsr_gs_descend1 (dBSRmat *A,
00780                                     dvector *b,
00781                                     dvector *u)
00782 {
00783     // members of A
00784     const INT      ROW = A->ROW;
00785     const INT      nb  = A->nb;
00786     const INT      nb2 = nb*nb;
00787     const INT      *IA  = A->IA;
00788     const INT      *JA  = A->JA;
00789     REAL           *val  = A->val;
00790
00791     // values of dvector b and u
00792     REAL *b_val = b->val;
00793     REAL *u_val = u->val;
00794
00795     // local variables
00796     INT i,j,k;
00797     INT pb;
00798     REAL rhs = 0.0;
00799
00800     if (nb == 1) {
00801         for (i = ROW-1; i >= 0; i--) {
00802             rhs = b_val[i];
00803             for (k = IA[i]; k < IA[i+1]; ++k) {
00804                 j = JA[k];
00805                 if (j != i)
00806                     rhs -= val[k]*u_val[j];
00807             }
00808             u_val[i] = rhs;
00809         }
00810     }
00811     else if (nb > 1) {
00812         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

```

```

00832
00833     for (i = ROW-1; i >= 0; i--) {
00834         pb = i*nb;
00835         memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
00836         for (k = IA[i]; k < IA[i+1]; ++k) {
00837             j = JA[k];
00838             if (j != i)
00839                 fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00840         }
00841         memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
00842     }
00843
00844     fasp_mem_free(b_tmp); b_tmp = NULL;
00845 }
00846 else {
00847     printf("### ERROR: nb is illegal! [%s:%d]\n", __FILE__, __LINE__);
00848     fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00849 }
00850
00851 }
00852
00868 void fasp_smoother_dbsr_gs_order1 (dBSRmat *A,
00869                                     dvector *b,
00870                                     dvector *u,
00871                                     REAL *diaginv,
00872                                     INT *mark)
00873 {
00874     // members of A
00875     const INT ROW = A->ROW;
00876     const INT nb = A->nb;
00877     const INT nb2 = nb*nb;
00878     const INT *IA = A->IA;
00879     const INT *JA = A->JA;
00880     REAL *val = A->val;
00881
00882     // values of dvector b and u
00883     REAL *b_val = b->val;
00884     REAL *u_val = u->val;
00885
00886     // local variables
00887     INT i, j, k;
00888     INT I, pb;
00889     REAL rhs = 0.0;
00890
00891     if (nb == 1) {
00892         for (I = 0; I < ROW; ++I) {
00893             i = mark[I];
00894             rhs = b_val[i];
00895             for (k = IA[i]; k < IA[i+1]; ++k) {
00896                 j = JA[k];
00897                 if (j != i)
00898                     rhs -= val[k]*u_val[j];
00899             }
00900             u_val[i] = rhs*diaginv[i];
00901         }
00902     }
00903     else if (nb > 1) {
00904         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
00905
00906         for (I = 0; I < ROW; ++I) {
00907             i = mark[I];
00908             pb = i*nb;
00909             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
00910             for (k = IA[i]; k < IA[i+1]; ++k) {
00911                 j = JA[k];
00912                 if (j != i)
00913                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00914             }
00915             fasp_blas_smat_m xv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
00916         }
00917
00918         fasp_mem_free(b_tmp); b_tmp = NULL;
00919     }
00920     else {
00921         fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00922     }
00923 }
00924 }
00925
00946 void fasp_smoother_dbsr_gs_order2 (dBSRmat *A,
00947                                     dvector *b,

```

```

00948                                     dvector *u,
00949                                     INT      *mark,
00950                                     REAL     *work)
00951 {
00952     // members of A
00953     const INT    ROW = A->ROW;
00954     const INT    nb  = A->nb;
00955     const INT    nb2 = nb*nb;
00956     const INT    *IA  = A->IA;
00957     const INT    *JA  = A->JA;
00958     REAL         *val = A->val;
00959
00960     // values of dvector b and u
00961     REAL *b_val = b->val;
00962     REAL *u_val = u->val;
00963
00964     // auxiliary array
00965     REAL *b_tmp = work;
00966
00967     // local variables
00968     INT i,j,k,I,pb;
00969     REAL rhs = 0.0;
00970
00971     if (nb == 1) {
00972         for (I = 0; I < ROW; ++I) {
00973             i = mark[I];
00974             rhs = b_val[i];
00975             for (k = IA[i]; k < IA[i+1]; ++k) {
00976                 j = JA[k];
00977                 if (j != i)
00978                     rhs -= val[k]*u_val[j];
00979             }
00980             u_val[i] = rhs;
00981         }
00982     }
00983     else if (nb > 1) {
00984         for (I = 0; I < ROW; ++I) {
00985             i = mark[I];
00986             pb = i*nb;
00987             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
00988             for (k = IA[i]; k < IA[i+1]; ++k) {
00989                 j = JA[k];
00990                 if (j != i)
00991                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
00992             }
00993             memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
00994         }
00995     }
00996     else {
00997         fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
00998     }
00999 }
01000
01023 void fasp_smoother_dbsr_sor (dBSRmat *A,
01024                             dvector *b,
01025                             dvector *u,
01026                             INT      order,
01027                             INT      *mark,
01028                             REAL     weight)
01029 {
01030     // members of A
01031     const INT    ROW = A->ROW;
01032     const INT    nb  = A->nb;
01033     const INT    nb2 = nb*nb;
01034     const INT    size = ROW*nb2;
01035     const INT    *IA  = A->IA;
01036     const INT    *JA  = A->JA;
01037     const REAL    *val = A->val;
01038
01039     // local variables
01040     INT i,k;
01041     REAL *diagin = NULL;
01042
01043     SHORT nthreads = 1, use_openmp = FALSE;
01044
01045     #ifdef _OPENMP
01046         if ( ROW > OPENMP_HOLDS ) {
01047             use_openmp = TRUE;
01048             nthreads = fasp_get_num_threads();
01049         }
01050     #endif

```

```

01051
01052 // allocate memory
01053 diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
01054
01055 // get all the diagonal sub-blocks
01056 if (use_omp) {
01057     INT mybegin,myend,myid;
01058 #ifdef _OPENMP
01059 #pragma omp parallel for private(myid,mybegin,myend,i,k)
01060 #endif
01061     for (myid=0; myid<nthreads; myid++) {
01062         fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01063         for (i=mybegin; i<myend; i++) {
01064             for (k=IA[i]; k<IA[i+1]; ++k)
01065                 if (JA[k] == i)
01066                     memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
01067         }
01068     }
01069 }
01070 else {
01071     for (i = 0; i < ROW; ++i) {
01072         for (k = IA[i]; k < IA[i+1]; ++k) {
01073             if (JA[k] == i)
01074                 memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
01075         }
01076     }
01077 }
01078
01079 // compute the inverses of all the diagonal sub-blocks
01080 if (nb > 1) {
01081     if (use_omp) {
01082         INT mybegin,myend,myid;
01083 #ifdef _OPENMP
01084 #pragma omp parallel for private(myid,mybegin,myend,i)
01085 #endif
01086         for (myid=0; myid<nthreads; myid++) {
01087             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01088             for (i=mybegin; i<myend; i++) {
01089                 fasp_smat_inv(diaginv+i*nb2, nb);
01090             }
01091         }
01092     }
01093     else {
01094         for (i = 0; i < ROW; ++i) {
01095             fasp_smat_inv(diaginv+i*nb2, nb);
01096         }
01097     }
01098 }
01099 else {
01100     if (use_omp) {
01101         INT mybegin, myend, myid;
01102 #ifdef _OPENMP
01103 #pragma omp parallel for private(myid,mybegin,myend,i)
01104 #endif
01105         for (myid=0; myid<nthreads; myid++) {
01106             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01107             for (i=mybegin; i<myend; i++) {
01108                 diaginv[i] = 1.0 / diaginv[i];
01109             }
01110         }
01111     }
01112     else {
01113         for (i = 0; i < ROW; ++i) {
01114             // zero-diagonal should be tested previously
01115             diaginv[i] = 1.0 / diaginv[i];
01116         }
01117     }
01118 }
01119
01120 fasp_smoother_dbsr_sor1(A, b, u, order, mark, diaginv, weight);
01121
01122 fasp_mem_free(diaginv); diaginv = NULL;
01123 }
01124
01146 void fasp_smoother_dbsr_sor1 (dBSRmat *A,
01147                               dvector *b,
01148                               dvector *u,
01149                               INT order,
01150                               INT *mark,
01151                               REAL *diaginv,
01152                               REAL weight)

```

```

01153 {
01154     if (!mark) {
01155         if (order == ASCEND)          // smooth ascendingly
01156         {
01157             fasp_smoother_dbsr_sor_ascend(A, b, u, diaginv, weight);
01158         }
01159         else if (order == DESCEND) // smooth descendingly
01160         {
01161             fasp_smoother_dbsr_sor_descend(A, b, u, diaginv, weight);
01162         }
01163     }
01164     // smooth according to the order 'mark' defined by user
01165     else {
01166         fasp_smoother_dbsr_sor_order(A, b, u, diaginv, mark, weight);
01167     }
01168 }
01169
01187 void fasp_smoother_dbsr_sor_ascend (dBSRmat *A,
01188                                     dvector *b,
01189                                     dvector *u,
01190                                     REAL *diaginv,
01191                                     REAL weight)
01192 {
01193     // members of A
01194     const INT ROW = A->ROW;
01195     const INT nb = A->nb;
01196     const INT *IA = A->IA;
01197     const INT *JA = A->JA;
01198     const REAL *val = A->val;
01199
01200     // values of dvector b and u
01201     const REAL *b_val = b->val;
01202     REAL *u_val = u->val;
01203
01204     // local variables
01205     const INT nb2 = nb*nb;
01206     INT i, j, k;
01207     INT pb;
01208     REAL rhs = 0.0;
01209     REAL one_minus_weight = 1.0 - weight;
01210
01211 #ifdef _OPENMP
01212     // variables for OpenMP
01213     INT myid, mybegin, myend;
01214     INT nthreads = fasp_get_num_threads();
01215 #endif
01216
01217     if (nb == 1) {
01218 #ifdef _OPENMP
01219         if (ROW > OPENMP_HOLDS) {
01220             #pragma omp parallel for private(myid, mybegin, myend, i, rhs, k, j)
01221             for (myid = 0; myid < nthreads; myid++) {
01222                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01223                 for (i = mybegin; i < myend; i++) {
01224                     rhs = b_val[i];
01225                     for (k = IA[i]; k < IA[i+1]; ++k) {
01226                         j = JA[k];
01227                         if (j != i)
01228                             rhs -= val[k]*u_val[j];
01229                     }
01230                     u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01231                 }
01232             }
01233         }
01234     }
01235 #endif
01236     else {
01237         for (i = 0; i < ROW; ++i) {
01238             rhs = b_val[i];
01239             for (k = IA[i]; k < IA[i+1]; ++k) {
01240                 j = JA[k];
01241                 if (j != i)
01242                     rhs -= val[k]*u_val[j];
01243             }
01244             u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01245         }
01246 #ifdef _OPENMP
01247 #endif
01248     }
01249     else if (nb > 1) {
01250 #ifdef _OPENMP

```

```

01251         if (ROW > OPENMP_HOLDS) {
01252             REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
01253 #pragma omp parallel for private(myid, mybegin, myend, i, pb, k, j)
01254             for (myid = 0; myid < nthreads; myid++) {
01255                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01256                 for (i = mybegin; i < myend; i++) {
01257                     pb = i*nb;
01258                     memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
01259                     for (k = IA[i]; k < IA[i+1]; ++k) {
01260                         j = JA[k];
01261                         if (j != i)
01262                             fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
01263                     }
01264                     fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb, one_minus_weight,
u_val+pb, nb);
01265                 }
01266             }
01267             fasp_mem_free(b_tmp); b_tmp = NULL;
01268         }
01269         else {
01270 #endif
01271             REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
01272             for (i = 0; i < ROW; ++i) {
01273                 pb = i*nb;
01274                 memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
01275                 for (k = IA[i]; k < IA[i+1]; ++k) {
01276                     j = JA[k];
01277                     if (j != i)
01278                         fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
01279                 }
01280                 fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight, u_val+pb, nb);
01281             }
01282             fasp_mem_free(b_tmp); b_tmp = NULL;
01283 #ifdef _OPENMP
01284         }
01285 #endif
01286     }
01287     else {
01288         fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
01289     }
01290 }
01291 }
01292
01310 void fasp_smoother_dbsr_sor_descend (dBSRmat *A,
01311                                     dvector *b,
01312                                     dvector *u,
01313                                     REAL *diaginv,
01314                                     REAL weight)
01315 {
01316     // members of A
01317     const INT ROW = A->ROW;
01318     const INT nb = A->nb;
01319     const INT nb2 = nb*nb;
01320     const INT *IA = A->IA;
01321     const INT *JA = A->JA;
01322     REAL *val = A->val;
01323     const REAL one_minus_weight = 1.0 - weight;
01324
01325     // values of dvector b and u
01326     REAL *b_val = b->val;
01327     REAL *u_val = u->val;
01328
01329     // local variables
01330     INT i, j, k;
01331     INT pb;
01332     REAL rhs = 0.0;
01333
01334 #ifdef _OPENMP
01335     // variables for OpenMP
01336     INT myid, mybegin, myend;
01337     INT nthreads = fasp_get_num_threads();
01338 #endif
01339
01340     if (nb == 1) {
01341 #ifdef _OPENMP
01342         if (ROW > OPENMP_HOLDS) {
01343 #pragma omp parallel for private(myid, mybegin, myend, i, rhs, k, j)
01344             for (myid = 0; myid < nthreads; myid++) {
01345                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01346                 mybegin = ROW-1-mybegin; myend = ROW-1-myend;
01347                 for (i = mybegin; i > myend; i--) {

```

```

01348         rhs = b_val[i];
01349         for (k = IA[i]; k < IA[i+1]; ++k) {
01350             j = JA[k];
01351             if (j != i)
01352                 rhs -= val[k]*u_val[j];
01353         }
01354         u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01355     }
01356 }
01357 }
01358 else {
01359 #endif
01360     for (i = ROW-1; i >= 0; i--) {
01361         rhs = b_val[i];
01362         for (k = IA[i]; k < IA[i+1]; ++k) {
01363             j = JA[k];
01364             if (j != i)
01365                 rhs -= val[k]*u_val[j];
01366         }
01367         u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01368     }
01369 #ifdef _OPENMP
01370 }
01371 #endif
01372 }
01373 else if (nb > 1) {
01374 #ifdef _OPENMP
01375     if (ROW > OPENMP_HOLDS) {
01376         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
01377 #pragma omp parallel for private(myid, mybegin, myend, i, pb, k, j)
01378         for (myid = 0; myid < nthreads; myid++) {
01379             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01380             mybegin = ROW-1-mybegin; myend = ROW-1-myend;
01381             for (i = mybegin; i > myend; i--) {
01382                 pb = i*nb;
01383                 memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
01384                 for (k = IA[i]; k < IA[i+1]; ++k) {
01385                     j = JA[k];
01386                     if (j != i)
01387                         fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+myid*nb, nb);
01388                 }
01389                 fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb,
01390                                     one_minus_weight, u_val+pb, nb);
01391             }
01392         }
01393         fasp_mem_free(b_tmp); b_tmp = NULL;
01394     }
01395     else {
01396 #endif
01397         REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
01398         for (i = ROW-1; i >= 0; i--) {
01399             pb = i*nb;
01400             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
01401             for (k = IA[i]; k < IA[i+1]; ++k) {
01402                 j = JA[k];
01403                 if (j != i)
01404                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
01405             }
01406             fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight,
01407                                 u_val+pb, nb);
01408         }
01409         fasp_mem_free(b_tmp); b_tmp = NULL;
01410 #ifdef _OPENMP
01411     }
01412 #endif
01413 }
01414 else {
01415     fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
01416 }
01417 }
01418 }
01419
01438 void fasp_smoother_dbsr_sor_order (dBSRmat *A,
01439                                   dvector *b,
01440                                   dvector *u,
01441                                   REAL *diaginv,
01442                                   INT *mark,
01443                                   REAL weight)
01444 {
01445     // members of A
01446     const INT ROW = A->ROW;

```



```

01447     const INT      nb  = A->nb;
01448     const INT      nb2 = nb*nb;
01449     const INT      *IA  = A->IA;
01450     const INT      *JA  = A->JA;
01451     REAL           *val  = A->val;
01452     const REAL     one_minus_weight = 1.0 - weight;
01453
01454     // values of dvector b and u
01455     REAL *b_val = b->val;
01456     REAL *u_val = u->val;
01457
01458     // local variables
01459     INT i,j,k;
01460     INT I,pb;
01461     REAL rhs = 0.0;
01462
01463 #ifdef _OPENMP
01464     // variables for OpenMP
01465     INT myid, mybegin, myend;
01466     INT nthreads = fasp_get_num_threads();
01467 #endif
01468
01469     if (nb == 1) {
01470 #ifdef _OPENMP
01471         if (ROW > OPENMP_HOLDS) {
01472 #pragma omp parallel for private(myid, mybegin, myend, I, i, rhs, k, j)
01473             for (myid = 0; myid < nthreads; myid++) {
01474                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01475                 for (I = mybegin; I < myend; ++I) {
01476                     i = mark[I];
01477                     rhs = b_val[i];
01478                     for (k = IA[i]; k < IA[i+1]; ++k) {
01479                         j = JA[k];
01480                         if (j != i)
01481                             rhs -= val[k]*u_val[j];
01482                     }
01483                     u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01484                 }
01485             }
01486         }
01487     }
01488 #endif
01489     else {
01490         for (I = 0; I < ROW; ++I) {
01491             i = mark[I];
01492             rhs = b_val[i];
01493             for (k = IA[i]; k < IA[i+1]; ++k) {
01494                 j = JA[k];
01495                 if (j != i)
01496                     rhs -= val[k]*u_val[j];
01497             }
01498             u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
01499         }
01500 #ifdef _OPENMP
01501     }
01502 #endif
01503     else if (nb > 1) {
01504 #ifdef _OPENMP
01505         if (ROW > OPENMP_HOLDS) {
01506             REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
01507 #pragma omp parallel for private(myid, mybegin, myend, I, i, pb, k, j)
01508             for (myid = 0; myid < nthreads; myid++) {
01509                 fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
01510                 for (I = mybegin; I < myend; ++I) {
01511                     i = mark[I];
01512                     pb = i*nb;
01513                     memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
01514                     for (k = IA[i]; k < IA[i+1]; ++k) {
01515                         j = JA[k];
01516                         if (j != i)
01517                             fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+myid*nb, nb);
01518                     }
01519                     fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb,
01520                                           one_minus_weight, u_val+pb, nb);
01521                 }
01522             }
01523             fasp_mem_free(b_tmp); b_tmp = NULL;
01524         }
01525     }
01526 #endif
01527     REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

```

```

01528         for (I = 0; I < ROW; ++I) {
01529             i = mark[I];
01530             pb = i*nb;
01531             memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
01532             for (k = IA[i]; k < IA[i+1]; ++k) {
01533                 j = JA[k];
01534                 if (j != i)
01535                     fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
01536             }
01537             fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight,
01538                                 u_val+pb, nb);
01539         }
01540         fasp_mem_free(b_tmp); b_tmp = NULL;
01541 #ifdef _OPENMP
01542     }
01543 #endif
01544 }
01545 else {
01546     fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
01547 }
01548 }
01549 }
01550
01566 void fasp_smoother_dbsr_ilu (dBSRmat *A,
01567                             dvector *b,
01568                             dvector *x,
01569                             void *data)
01570 {
01571     ILU_data *iludata=(ILU_data *)data;
01572     const INT nb=iludata->nb,m=A->ROW*nb, memneed=5*m;
01573
01574     REAL *xval = x->val, *bval = b->val;
01575     REAL *zr = iludata->work + 3*m;
01576     REAL *z = zr + m;
01577
01578     double start, end;
01579
01580     if (iludata->nwork<memneed) goto MEMERR;
01581
01582     fasp_darray_cp(m,bval,zr); fasp_blas_dbsr_aAxy(-1.0,A,xval,zr);
01583
01584 #ifdef __OPENMP
01585 #if ILU_MC_OMP
01586     REAL *tz = (REAL*)fasp_mem_calloc(A->ROW*A->nb, sizeof(REAL));
01587     REAL *tzr = (REAL*)fasp_mem_calloc(A->ROW*A->nb, sizeof(REAL));
01588     perm(A->ROW, A->nb, zr, iludata->jlevL, tzr);
01589
01590     fasp_gettime(&start);
01591     fasp_precond_dbsr_ilu_mc_omp(tzr,tz,iludata);
01592     fasp_gettime(&end);
01593
01594     invperm(A->ROW, A->nb, tz, iludata->jlevL, z);
01595     fasp_mem_free(tzr); tzr = NULL;
01596     fasp_mem_free(tz); tz = NULL;
01597 #else
01598     fasp_gettime(&start);
01599     fasp_precond_dbsr_ilu_ls_omp(zr,z,iludata);
01600     fasp_gettime(&end);
01601 #endif
01602 #endif
01603
01604     ilu_solve_time += end-start;
01605
01606 #else
01607     fasp_gettime(&start);
01608     fasp_precond_dbsr_ilu(zr,z,iludata);
01609     fasp_gettime(&end);
01610     ilu_solve_time += end-start;
01611 #endif
01612
01613     fasp_blas_darray_axpy(m,1,z,xval);
01614
01615     return;
01616
01622 MEMERR:
01623     printf("### ERROR: ILU needs %d memory, only %d available! [%s:%d]\n",
01624           memneed, iludata->nwork, __FILE__, __LINE__);
01625     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01626 }

```

```

01627
01628 /*-----*/
01629 /*--      Private Functions      --*/
01630 /*-----*/
01631
01632 #ifdef _OPENMP
01633 #if ILU_MC_OMP
01635
01651 static inline void perm (const INT    n,
01652                          const INT    nb,
01653                          const REAL    *x,
01654                          const INT    *p,
01655                          REAL          *y)
01656 {
01657     INT i, j, indx, indy;
01658
01659 #ifdef _OPENMP
01660 #pragma omp parallel for private(i, j, indx, indy)
01661 #endif
01662     for (i=0; i<n; ++i) {
01663         indx = p[i]*nb;
01664         indy = i*nb;
01665         for (j=0; j<nb; ++j) {
01666             y[indy+j] = x[indx+j];
01667         }
01668     }
01669 }
01670
01686 static inline void invperm (const INT    n,
01687                             const INT    nb,
01688                             const REAL    *x,
01689                             const INT    *p,
01690                             REAL          *y)
01691 {
01692     INT i, j, indx, indy;
01693
01694 #ifdef _OPENMP
01695 #pragma omp parallel for private(i, j, indx, indy)
01696 #endif
01697     for (i=0; i<n; ++i) {
01698         indx = i*nb;
01699         indy = p[i]*nb;
01700         for (j=0; j<nb; ++j) {
01701             y[indy+j] = x[indx+j];
01702         }
01703     }
01704 }
01705
01706 #endif // end of ILU_MC_OMP
01707
01708 #endif // end of _OPENMP
01709
01710 /*-----*/
01711 /*--      End of File      --*/
01712 /*-----*/

```

9.101 ItrSmootherCSR.c File Reference

Smoothers for [dCSRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void [fasp_smoother_dcsr_jacobi](#) ([dvector](#) *u, const [INT](#) i_1, const [INT](#) i_n, const [INT](#) s, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L, const [REAL](#) w)
Weighted Jacobi method as a smoother.
- void [fasp_smoother_dcsr_gs](#) ([dvector](#) *u, const [INT](#) i_1, const [INT](#) i_n, const [INT](#) s, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L)

Gauss-Seidel method as a smoother.

- void [fasp_smoother_dcsr_gs_cf](#) ([dvector](#) *u, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L, [INT](#) *mark, const [INT](#) order)

Gauss-Seidel smoother with C/F ordering for Au=b.

- void [fasp_smoother_dcsr_sgs](#) ([dvector](#) *u, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L)

Symmetric Gauss-Seidel method as a smoother.

- void [fasp_smoother_dcsr_sor](#) ([dvector](#) *u, const [INT](#) i_1, const [INT](#) i_n, const [INT](#) s, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L, const [REAL](#) w)

SOR method as a smoother.

- void [fasp_smoother_dcsr_sor_cf](#) ([dvector](#) *u, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L, const [REAL](#) w, [INT](#) *mark, const [INT](#) order)

SOR smoother with C/F ordering for Au=b.

- void [fasp_smoother_dcsr_ilu](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, void *data)

ILU method as a smoother.

- void [fasp_smoother_dcsr_kaczmarz](#) ([dvector](#) *u, const [INT](#) i_1, const [INT](#) i_n, const [INT](#) s, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L, const [REAL](#) w)

Kaczmarz method as a smoother.

- void [fasp_smoother_dcsr_L1diag](#) ([dvector](#) *u, const [INT](#) i_1, const [INT](#) i_n, const [INT](#) s, [dCSRmat](#) *A, [dvector](#) *b, [INT](#) L)

Diagonal scaling (using L1 norm) as a smoother.

9.101.1 Detailed Description

Smoothers for [dCSRmat](#) matrices.

Note

This file contains Level-2 (ltr) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), [BlaArray.c](#), and [BlaSpmvCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [ltrSmootherCSR.c](#).

9.101.2 Function Documentation

9.101.2.1 [fasp_smoother_dcsr_gs\(\)](#)

```
void fasp_smoother_dcsr_gs (
    dvector * u,
    const INT i_1,
    const INT i_n,
    const INT s,
    dCSRmat * A,
    dvector * b,
    INT L )
```

Gauss-Seidel method as a smoother.

Parameters

<i>u</i>	Pointer to dvector : the unknowns (IN: initial, OUT: approximation)
----------	---

Parameters

i_{\leftarrow} $_ \leftarrow$ 1	Starting index
i_{\leftarrow} $_ \leftarrow$ n	Ending index
s	Increasing step
A	Pointer to dBSRmat : the coefficient matrix
b	Pointer to dvector: the right hand side
L	Number of iterations

Author

Xuehai Huang, Chensong Zhang

Date

09/26/2009

Modified by Chunsheng Feng, Zheng Li on 09/01/2012

Definition at line [190](#) of file [ltrSmootherCSR.c](#).

9.101.2.2 fasp_smoother_dcsr_gs_cf()

```
void fasp_smoother_dcsr_gs_cf (
    dvector * u,
    dCSRmat * A,
    dvector * b,
    INT L,
    INT * mark,
    const INT order )
```

Gauss-Seidel smoother with C/F ordering for $Au=b$.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
A	Pointer to dBSRmat : the coefficient matrix
b	Pointer to dvector: the right hand side
L	Number of iterations
$mark$	C/F marker array
$order$	C/F ordering: -1: F-first; 1: C-first

Author

Zhiyang Zhou

Date

11/12/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/24/2012

Definition at line 363 of file [ltrSmootherCSR.c](#).**9.101.2.3 fasp_smoother_dcsr_ilu()**

```
void fasp_smoother_dcsr_ilu (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    void * data )
```

ILU method as a smoother.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>x</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>data</i>	Pointer to user defined data

Author

Shiquan Zhang, Xiaozhe Hu

Date

2010/11/12

form residual $zr = b - A x$ Definition at line 1065 of file [ltrSmootherCSR.c](#).**9.101.2.4 fasp_smoother_dcsr_jacobi()**

```
void fasp_smoother_dcsr_jacobi (
    dvector * u,
    const INT i_l,
    const INT i_n,
    const INT s,
    dCSRmat * A,
    dvector * b,
    INT L,
    const REAL w )
```

Weighted Jacobi method as a smoother.

Parameters

<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
$i \leftarrow$ \leftarrow 1	Starting index

Parameters

i_{\leftarrow} $_{\leftarrow}$ n	Ending index
s	Increasing step
A	Pointer to dBSRmat : the coefficient matrix
b	Pointer to dvector: the right hand side
L	Number of iterations
w	Over-relaxation weight

Author

Xuehai Huang, Chensong Zhang

Date

09/26/2009

Modified by Chunsheng Feng, Zheng Li on 08/29/2012 Modified by Chensong Zhang on 08/24/2017: Pass weight w as a parameter

Definition at line 50 of file [ltrSmootherCSR.c](#).

9.101.2.5 fasp_smoother_dcsr_kaczmarz()

```
void fasp_smoother_dcsr_kaczmarz (
    dvector * u,
    const INT i_1,
    const INT i_n,
    const INT s,
    dCSRmat * A,
    dvector * b,
    INT L,
    const REAL w )
```

Kaczmarz method as a smoother.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i_{\leftarrow} $_{\leftarrow}$ 1	Starting index
i_{\leftarrow} $_{\leftarrow}$ n	Ending index
s	Increasing step
A	Pointer to dBSRmat : the coefficient matrix
b	Pointer to dvector: the right hand side
L	Number of iterations
w	Over-relaxation weight

Author

Xiaozhe Hu

Date

2010/11/12

Modified by Chunsheng Feng, Zheng Li on 2012/09/01
 Definition at line 1144 of file [ltrSmootherCSR.c](#).

9.101.2.6 fasp_smoother_dcsr_L1diag()

```
void fasp_smoother_dcsr_L1diag (
    dvector * u,
    const INT i_l,
    const INT i_n,
    const INT s,
    dCSRmat * A,
    dvector * b,
    INT L )
```

Diagonal scaling (using L1 norm) as a smoother.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i_{\leftarrow} $_{\leftarrow}$ 1	Starting index
i_{\leftarrow} $_{\leftarrow}$ n	Ending index
s	Increasing step
A	Pointer to dBSRmat : the coefficient matrix
b	Pointer to dvector: the right hand side
L	Number of iterations

Author

Xiaozhe Hu, James Brannick

Date

01/26/2011

Modified by Chunsheng Feng, Zheng Li on 09/01/2012
 Definition at line 1284 of file [ltrSmootherCSR.c](#).

9.101.2.7 fasp_smoother_dcsr_sgs()

```
void fasp_smoother_dcsr_sgs (
    dvector * u,
    dCSRmat * A,
```



```

    dvector * b,
    INT L )

```

Symmetric Gauss-Seidel method as a smoother.

Parameters

<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>L</i>	Number of iterations

Author

Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng, Zheng Li on 09/01/2012

Definition at line [628](#) of file [ltrSmootherCSR.c](#).

9.101.2.8 fasp_smoother_dcsr_sor()

```

void fasp_smoother_dcsr_sor (
    dvector * u,
    const INT i_l,
    const INT i_n,
    const INT s,
    dCSRmat * A,
    dvector * b,
    INT L,
    const REAL w )

```

SOR method as a smoother.

Parameters

<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i_{\leftarrow} \leftarrow <i>1</i>	Starting index
i_{\leftarrow} \leftarrow <i>n</i>	Ending index
<i>s</i>	Increasing step
<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>L</i>	Number of iterations
<i>w</i>	Over-relaxation weight

Author

Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng, Zheng Li on 09/01/2012
 Definition at line 744 of file [ltrSmootherCSR.c](#).

9.101.2.9 fasp_smoother_dcsr_sor_cf()

```
void fasp_smoother_dcsr_sor_cf (
    dvector * u,
    dCSRmat * A,
    dvector * b,
    INT L,
    const REAL w,
    INT * mark,
    const INT order )
```

SOR smoother with C/F ordering for $Au=b$.

Parameters

<i>u</i>	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>L</i>	Number of iterations
<i>w</i>	Over-relaxation weight
<i>mark</i>	C/F marker array
<i>order</i>	C/F ordering: -1: F-first; 1: C-first

Author

Zhiyang Zhou

Date

2010/11/12

Modified by Chunsheng Feng, Zheng Li on 08/29/2012
 Definition at line 871 of file [ltrSmootherCSR.c](#).

9.102 ltrSmootherCSR.c

[Go to the documentation of this file.](#)

```
00001
00015 #include <math.h>
00016
00017 #ifdef _OPENMP
00018 #include <omp.h>
00019 #endif
00020
00021 #include "fasp.h"
00022 #include "fasp_funcs.h"
```

```

00023
00024 /*-----*/
00025 /*--      Public Functions      --*/
00026 /*-----*/
00027
00050 void fasp_smoother_dcsr_jacobi (dvector      *u,
00051                               const INT     i_1,
00052                               const INT     i_n,
00053                               const INT     s,
00054                               dCSRmat      *A,
00055                               dvector      *b,
00056                               INT          L,
00057                               const REAL    w)
00058 {
00059     const INT     N = ABS(i_n - i_1) + 1;
00060     const INT     ia = A->IA, ja = A->JA;
00061     const REAL    aval = A->val, bval = b->val;
00062     REAL          uval = u->val;
00063
00064     // local variables
00065     INT i, j, k, begin_row, end_row;
00066
00067     // OpenMP variables
00068 #ifdef _OPENMP
00069     INT myid, mybegin, myend;
00070     INT nthreads = fasp_get_num_threads();
00071 #endif
00072
00073     REAL *t = (REAL *)fasp_mem_calloc(N, sizeof(REAL));
00074     REAL *d = (REAL *)fasp_mem_calloc(N, sizeof(REAL));
00075
00076     while (L--) {
00077
00078         if (s>0) {
00079             #ifdef _OPENMP
00080                 if (N > OPENMP_HOLDS) {
00081                     #pragma omp parallel for private(myid, mybegin, myend, begin_row, end_row, i, k, j)
00082                     for (myid=0; myid<nthreads; ++myid) {
00083                         fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00084                         mybegin += i_1; myend += i_1;
00085                         for (i=mybegin; i<myend; i+=s) {
00086                             t[i]=bval[i];
00087                             begin_row=ia[i], end_row=ia[i+1];
00088                             for (k=begin_row; k<end_row; ++k) {
00089                                 j=ja[k];
00090                                 if (i!=j) t[i]-=aval[k]*uval[j];
00091                                 else d[i]=aval[k];
00092                             }
00093                         }
00094                     }
00095                 }
00096             else {
00097                 #endif
00098                 for (i=i_1; i<=i_n; i+=s) {
00099                     t[i]=bval[i];
00100                     begin_row=ia[i]; end_row=ia[i+1];
00101                     for (k=begin_row; k<end_row; ++k) {
00102                         j=ja[k];
00103                         if (i!=j) t[i]-=aval[k]*uval[j];
00104                         else d[i]=aval[k];
00105                     }
00106                 }
00107             #ifdef _OPENMP
00108             }
00109             #endif
00110
00111             #ifdef _OPENMP
00112             #pragma omp parallel for private (i)
00113             #endif
00114             for (i=i_1; i<=i_n; i+=s) {
00115                 if (ABS(d[i])>SMALLREAL) uval[i]=(1-w)*uval[i] + w*t[i]/d[i];
00116             }
00117
00118         }
00119
00120         else {
00121
00122             #ifdef _OPENMP
00123             if (N > OPENMP_HOLDS) {
00124                 #pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, k, j)
00125                 for (myid=0; myid<nthreads; myid++) {

```

```

00126         fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00127         mybegin = i_1-mybegin; myend = i_1-myend;
00128         for (i=mybegin; i>myend; i+=s) {
00129             t[i]=bval[i];
00130             begin_row=ia[i],end_row=ia[i+1];
00131             for (k=begin_row; k<end_row; ++k) {
00132                 j=ja[k];
00133                 if (i!=j) t[i]-=aval[k]*uval[j];
00134                 else d[i]=aval[k];
00135             }
00136         }
00137     }
00138 }
00139 else {
00140 #endif
00141     for (i=i_1;i>=i_n;i+=s) {
00142         t[i]=bval[i];
00143         begin_row=ia[i]; end_row=ia[i+1];
00144         for (k=begin_row;k<end_row;++k) {
00145             j=ja[k];
00146             if (i!=j) t[i]-=aval[k]*uval[j];
00147             else d[i]=aval[k];
00148         }
00149     }
00150 #ifdef _OPENMP
00151 }
00152 #endif
00153 #ifdef _OPENMP
00154 #pragma omp parallel for private(i)
00155 #endif
00156     for (i=i_1;i>=i_n;i+=s) {
00157         if (ABS(d[i])>SMALLREAL) uval[i]=(1-w)*uval[i]+ w*t[i]/d[i];
00158     }
00159 }
00160 }
00161 }
00162 } // end while
00163
00164 fasp_mem_free(t); t = NULL;
00165 fasp_mem_free(d); d = NULL;
00166
00167 return;
00168 }
00169 }
00170
00190 void fasp_smoother_dcsr_gs (dvector *u,
00191                             const INT i_1,
00192                             const INT i_n,
00193                             const INT s,
00194                             dCSRmat *A,
00195                             dvector *b,
00196                             INT L)
00197 {
00198     const INT *ia = A->IA, *ja = A->JA;
00199     const REAL *aval = A->val, *bval = b->val;
00200     REAL *uval = u->val;
00201
00202     // local variables
00203     INT i,j,k,begin_row,end_row;
00204     REAL t,d=0.0;
00205
00206 #ifdef _OPENMP
00207     const INT N = ABS(i_n - i_1)+1;
00208     INT myid, mybegin, myend;
00209     INT nthreads = fasp_get_num_threads();
00210 #endif
00211
00212     if (s > 0) {
00213         while (L--) {
00214             #ifdef _OPENMP
00215                 if (N > OPENMP_HOLDS) {
00216                     #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, d, k, j)
00217                     for (myid=0; myid<nthreads; myid++) {
00218                         fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00219                         mybegin += i_1, myend += i_1;
00220                         for (i=mybegin; i<myend; i+=s) {
00221                             t = bval[i];
00222                             begin_row=ia[i],end_row=ia[i+1];
00223                         }
00224                     #if DIAGONAL_PREF // diagonal first
00225                         d=aval[begin_row];

```

```

00226             if (ABS(d)>SMALLREAL) {
00227                 for (k=begin_row+1;k<end_row;++k) {
00228                     j=ja[k];
00229                     t-=aval[k]*uval[j];
00230                 }
00231                 uval[i]=t/d;
00232             }
00233 #else // general order
00234             for (k=begin_row;k<end_row;++k) {
00235                 j=ja[k];
00236                 if (i!=j)
00237                     t-=aval[k]*uval[j];
00238                 else if (ABS(aval[k])>SMALLREAL) d=1.e+0/aval[k];
00239             }
00240             uval[i]=t*d;
00241 #endif // end DIAGONAL_PREF
00242         } // end for i
00243     }
00244 }
00245 }
00246 }
00247 else {
00248 #endif
00249     for (i=i_1;i<=i_n;i+=s) {
00250         t = bval[i];
00251         begin_row=ia[i]; end_row=ia[i+1];
00252     }
00253 #if DIAGONAL_PREF // diagonal first
00254     d=aval[begin_row];
00255     if (ABS(d)>SMALLREAL) {
00256         for (k=begin_row+1;k<end_row;++k) {
00257             j=ja[k];
00258             t-=aval[k]*uval[j];
00259         }
00260         uval[i]=t/d;
00261     }
00262 #else // general order
00263     for (k=begin_row;k<end_row;++k) {
00264         j=ja[k];
00265         if (i!=j)
00266             t-=aval[k]*uval[j];
00267         else if (ABS(aval[k])>SMALLREAL) d=1.e+0/aval[k];
00268     }
00269     uval[i]=t*d;
00270 #endif
00271     } // end for i
00272 #ifdef _OPENMP
00273 }
00274 #endif
00275 } // end while
00276 }
00277 } // if s
00278 else {
00279     while (L--) {
00280 #ifdef _OPENMP
00281         if (N > OPENMP_HOLDS) {
00282             #pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, d, k, j, t)
00283             for (myid=0; myid<nthreads; myid++) {
00284                 fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00285                 mybegin = i_1 - mybegin; myend = i_1 - myend;
00286                 for (i=mybegin; i>myend; i+=s) {
00287                     t=bval[i];
00288                     begin_row=ia[i],end_row=ia[i+1];
00289 #if DIAGONAL_PREF // diagonal first
00290                     d=aval[begin_row];
00291                     if (ABS(d)>SMALLREAL) {
00292                         for (k=begin_row+1;k<end_row;++k) {
00293                             j=ja[k];
00294                             t-=aval[k]*uval[j];
00295                         }
00296                         uval[i]=t/d;
00297                     }
00298                 }
00299 #else // general order
00300                 for (k=begin_row;k<end_row;++k) {
00301                     j=ja[k];
00302                     if (i!=j)
00303                         t-=aval[k]*uval[j];
00304                     else if (ABS(aval[k])>SMALLREAL) d=1.0/aval[k];
00305                 }
00306                 uval[i]=t*d;

```

```

00307 #endif
00308             } // end for i
00309         }
00310     }
00311     else {
00312 #endif
00313         for (i=i_l;i>=i_n;i+=s) {
00314             t=bval[i];
00315             begin_row=ia[i]; end_row=ia[i+1];
00316 #if DIAGONAL_PREF // diagonal first
00317             d=aval[begin_row];
00318             if (ABS(d)>SMALLREAL) {
00319                 for (k=begin_row+1;k<end_row;++k) {
00320                     j=ja[k];
00321                     t-=aval[k]*uval[j];
00322                 }
00323                 uval[i]=t/d;
00324             }
00325 #else // general order
00326             for (k=begin_row;k<end_row;++k) {
00327                 j=ja[k];
00328                 if (i!=j)
00329                     t-=aval[k]*uval[j];
00330                 else if (ABS(aval[k])>SMALLREAL) d=1.0/aval[k];
00331             }
00332             uval[i]=t*d;
00333 #endif
00334         } // end for i
00335 #ifdef _OPENMP
00336     }
00337 #endif
00338 } // end while
00339
00340 } // end if
00341
00342 return;
00343 }
00344
00363 void fasp_smoother_dcsr_gs_cf (dvector *u,
00364                               dCSRmat *A,
00365                               dvector *b,
00366                               INT L,
00367                               INT *mark,
00368                               const INT order)
00369 {
00370     const INT nrow = b->nrow; // number of rows
00371     const INT *ia = A->IA, *ja = A->JA;
00372     const REAL *aval = A->val, *bval = b->val;
00373     REAL *uval = u->val;
00374
00375     INT i,j,k,begin_row,end_row;
00376     REAL t,d=0.0;
00377
00378 #ifdef _OPENMP
00379     INT myid,mybegin,myend;
00380     INT nthreads = fasp_get_num_threads();
00381 #endif
00382
00383     // F-point first, C-point second
00384     if (order == FFFIRST) {
00385
00386         while (L--) {
00387
00388 #ifdef _OPENMP
00389             if (nrow > OPENMP_HOLDS) {
00390 #pragma omp parallel for private(myid, mybegin, myend, i,t,begin_row,end_row,k,j,d)
00391                 for (myid = 0; myid < nthreads; myid++) {
00392                     fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00393                     for (i=mybegin; i<myend; i++) {
00394                         if (mark[i] != 1) {
00395                             t = bval[i];
00396                             begin_row = ia[i], end_row = ia[i+1];
00397 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00398                             d = aval[begin_row];
00399                             for (k = begin_row+1; k < end_row; k++) {
00400                                 j = ja[k];
00401                                 t -= aval[k]*uval[j];
00402                             } // end for k
00403 #else
00404                             for (k = begin_row; k < end_row; k++) {
00405                                 j = ja[k];

```

```

00406             if (i!=j) t -= aval[k]*uval[j];
00407             else d = aval[k];
00408         } // end for k
00409 #endif // end if DIAG_PREF
00410             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00411         }
00412     } // end for i
00413 }
00414 }
00415 else {
00416 #endif
00417     for (i = 0; i < nrow; i++) {
00418         if (mark[i] != 1) {
00419             t = bval[i];
00420             begin_row = ia[i]; end_row = ia[i+1];
00421 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00422             d = aval[begin_row];
00423             for (k = begin_row+1; k < end_row; k++) {
00424                 j = ja[k];
00425                 t -= aval[k]*uval[j];
00426             } // end for k
00427 #else
00428             for (k = begin_row; k < end_row; k++) {
00429                 j = ja[k];
00430                 if (i!=j) t -= aval[k]*uval[j];
00431                 else d = aval[k];
00432             } // end for k
00433 #endif // end if DIAG_PREF
00434             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00435         }
00436     } // end for i
00437 #ifdef _OPENMP
00438 }
00439 #endif
00440
00441 #ifdef _OPENMP
00442     if (nrow > OPENMP_HOLDS) {
00443 #pragma omp parallel for private(myid,mybegin,myend,i,t,begin_row,end_row,k,j,d)
00444         for (myid = 0; myid < nthreads; myid++) {
00445             fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00446             for (i=mybegin; i<myend; i++) {
00447                 if (mark[i] == 1) {
00448                     t = bval[i];
00449                     begin_row = ia[i]; end_row = ia[i+1];
00450 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00451                     d = aval[begin_row];
00452                     for (k = begin_row+1; k < end_row; k++) {
00453                         j = ja[k];
00454                         t -= aval[k]*uval[j];
00455                     } // end for k
00456 #else
00457                     for (k = begin_row; k < end_row; k++) {
00458                         j = ja[k];
00459                         if (i!=j) t -= aval[k]*uval[j];
00460                         else d = aval[k];
00461                     } // end for k
00462 #endif // end if DIAG_PREF
00463                     if (ABS(d) > SMALLREAL) uval[i] = t/d;
00464                 }
00465             } // end for i
00466         }
00467     }
00468     else {
00469 #endif
00470         for (i = 0; i < nrow; i++) {
00471             if (mark[i] == 1) {
00472                 t = bval[i];
00473                 begin_row = ia[i]; end_row = ia[i+1];
00474 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00475                 d = aval[begin_row];
00476                 for (k = begin_row+1; k < end_row; k++) {
00477                     j = ja[k];
00478                     t -= aval[k]*uval[j];
00479                 } // end for k
00480 #else
00481                 for (k = begin_row; k < end_row; k++) {
00482                     j = ja[k];
00483                     if (i!=j) t -= aval[k]*uval[j];
00484                     else d = aval[k];
00485                 } // end for k
00486 #endif // end if DIAG_PREF

```

```

00487             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00488         }
00489     } // end for i
00490 #ifdef _OPENMP
00491     }
00492 #endif
00493 } // end while
00494
00495 }
00496 // C-point first, F-point second
00497 else {
00498     while (L--) {
00499 #ifdef _OPENMP
00500         if (nrow > OPENMP_HOLDS) {
00501 #pragma omp parallel for private(myid,mybegin,myend,t,i,begin_row,end_row,k,j,d)
00502             for (myid = 0; myid < nthreads; myid++) {
00503                 fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00504                 for (i=mybegin; i<myend; i++) {
00505                     if (mark[i] == 1) {
00506                         t = bval[i];
00507                         begin_row = ia[i],end_row = ia[i+1];
00508 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00509                         d = aval[begin_row];
00510                         for (k = begin_row+1; k < end_row; k++) {
00511                             j = ja[k];
00512                             t -= aval[k]*uval[j];
00513                         } // end for k
00514 #else
00515                         for (k = begin_row; k < end_row; k++) {
00516                             j = ja[k];
00517                             if (i!=j) t -= aval[k]*uval[j];
00518                             else d = aval[k];
00519                         } // end for k
00520 #endif // end if DIAG_PREF
00521                     if (ABS(d) > SMALLREAL) uval[i] = t/d;
00522                 }
00523             } // end for i
00524         }
00525     } // end while
00526 }
00527 }
00528 else {
00529 #endif
00530     for (i = 0; i < nrow; i++) {
00531         if (mark[i] == 1) {
00532             t = bval[i];
00533             begin_row = ia[i], end_row = ia[i+1];
00534 #if DIAGONAL_PREF // Added by Chensong on 09/22/2012
00535             d = aval[begin_row];
00536             for (k = begin_row+1; k < end_row; k++) {
00537                 j = ja[k];
00538                 t -= aval[k]*uval[j];
00539             } // end for k
00540 #else
00541             for (k = begin_row; k < end_row; k++) {
00542                 j = ja[k];
00543                 if (i!=j) t -= aval[k]*uval[j];
00544                 else d = aval[k];
00545             } // end for k
00546 #endif // end if DIAG_PREF
00547             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00548         }
00549     } // end for i
00550 #ifdef _OPENMP
00551     }
00552 #endif
00553
00554 #ifdef _OPENMP
00555     if (nrow > OPENMP_HOLDS) {
00556 #pragma omp parallel for private(myid, mybegin, myend, i,t,begin_row,end_row,k,j,d)
00557         for (myid = 0; myid < nthreads; myid++) {
00558             fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00559             for (i=mybegin; i<myend; i++) {
00560                 if (mark[i] != 1) {
00561                     t = bval[i];
00562                     begin_row = ia[i],end_row = ia[i+1];
00563 #if DIAGONAL_PREF // Added by Chensong on 01/17/2013
00564                     d = aval[begin_row];
00565                     for (k = begin_row+1; k < end_row; k++) {
00566                         j = ja[k];
00567                         t -= aval[k]*uval[j];

```



```

00568             } // end for k
00569 #else
00570             for (k = begin_row; k < end_row; k++) {
00571                 j = ja[k];
00572                 if (i!=j) t -= aval[k]*uval[j];
00573                 else d = aval[k];
00574             } // end for k
00575 #endif // end if DIAG_PREF
00576             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00577         }
00578     } // end for i
00579 }
00580 }
00581 else {
00582 #endif
00583     for (i = 0; i < nrow; i++) {
00584         if (mark[i] != 1) {
00585             t = bval[i];
00586             begin_row = ia[i]; end_row = ia[i+1];
00587 #if DIAGONAL_PREF // Added by Chensong on 09/22/2012
00588             d = aval[begin_row];
00589             for (k = begin_row+1; k < end_row; k++) {
00590                 j = ja[k];
00591                 t -= aval[k]*uval[j];
00592             } // end for k
00593 #else
00594             for (k = begin_row; k < end_row; k++) {
00595                 j = ja[k];
00596                 if (i!=j) t -= aval[k]*uval[j];
00597                 else d = aval[k];
00598             } // end for k
00599 #endif // end if DIAG_PREF
00600             if (ABS(d) > SMALLREAL) uval[i] = t/d;
00601         }
00602     } // end for i
00603 #ifdef _OPENMP
00604 }
00605 #endif
00606 } // end while
00607 } // end if order
00608 }
00609 return;
00611 }
00612
00628 void fasp_smoother_dcsr_sgs (dvector *u,
00629                             dCSRmat *A,
00630                             dvector *b,
00631                             INT L)
00632 {
00633     const INT nml=b->row-1;
00634     const INT *ia=A->IA,*ja=A->JA;
00635     const REAL *aval=A->val,*bval=b->val;
00636     REAL uval=u->val;
00637
00638     // local variables
00639     INT i,j,k,begin_row,end_row;
00640     REAL t,d=0;
00641
00642 #ifdef _OPENMP
00643     INT myid, mybegin, myend, up;
00644     INT nthreads = fasp_get_num_threads();
00645 #endif
00646
00647     while (L--) {
00648         // forward sweep
00649 #ifdef _OPENMP
00650         up = nml + 1;
00651         if (up > OPENMP_HOLDS) {
00652 #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, j, k, d)
00653             for (myid=0; myid<nthreads; myid++) {
00654                 fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00655                 for (i=mybegin; i<myend; i++) {
00656                     t=bval[i];
00657                     begin_row=ia[i], end_row=ia[i+1];
00658                     for (k=begin_row;k<end_row;++k) {
00659                         j=ja[k];
00660                         if (i!=j) t-=aval[k]*uval[j];
00661                         else d=aval[k];
00662                     } // end for k
00663                     if (ABS(d)>SMALLREAL) uval[i]=t/d;

```

```

00664         } // end for i
00665     }
00666 }
00667 else {
00668 #endif
00669     for (i=0;i<=nml;++i) {
00670         t=bval[i];
00671         begin_row=ia[i]; end_row=ia[i+1];
00672         for (k=begin_row;k<end_row;++k) {
00673             j=ja[k];
00674             if (i!=j) t-=aval[k]*uval[j];
00675             else d=aval[k];
00676         } // end for k
00677         if (ABS(d)>SMALLREAL) uval[i]=t/d;
00678     } // end for i
00679 #ifdef _OPENMP
00680     }
00681 #endif
00682 // backward sweep
00683 #ifdef _OPENMP
00684     up = nml;
00685     if (up > OPENMP_HOLDS) {
00686 #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
00687         for (myid=0; myid<nthreads; myid++) {
00688             fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
00689             mybegin = nml-1-mybegin; myend = nml-1-myend;
00690             for (i=mybegin; i>myend; i--) {
00691                 t=bval[i];
00692                 begin_row=ia[i], end_row=ia[i+1];
00693                 for (k=begin_row; k<end_row; k++) {
00694                     j=ja[k];
00695                     if (i!=j) t-=aval[k]*uval[j];
00696                     else d=aval[k];
00697                 } // end for k
00698                 if (ABS(d)>SMALLREAL) uval[i]=t/d;
00699             } // end for i
00700         }
00701     }
00702 }
00703 else {
00704 #endif
00705     for (i=nml-1;i>=0;--i) {
00706         t=bval[i];
00707         begin_row=ia[i]; end_row=ia[i+1];
00708         for (k=begin_row;k<end_row;++k) {
00709             j=ja[k];
00710             if (i!=j) t-=aval[k]*uval[j];
00711             else d=aval[k];
00712         } // end for k
00713         if (ABS(d)>SMALLREAL) uval[i]=t/d;
00714     } // end for i
00715 #ifdef _OPENMP
00716     }
00717 #endif
00718 } // end while
00719 return;
00720 }
00721 }
00722
00744 void fasp_smoother_dcsr_sor (dvector *u,
00745                             const INT i_1,
00746                             const INT i_n,
00747                             const INT s,
00748                             dCSRmat *A,
00749                             dvector *b,
00750                             INT L,
00751                             const REAL w)
00752 {
00753     const INT *ia=A->IA, *ja=A->JA;
00754     const REAL *aval=A->val, *bval=b->val;
00755     REAL *uval=u->val;
00756 // local variables
00757     INT i, j, k, begin_row, end_row;
00758     REAL t, d=0;
00759
00760 #ifdef _OPENMP
00761     const INT N = ABS(i_n - i_1)+1;
00762     INT myid, mybegin, myend;
00763     INT nthreads = fasp_get_num_threads();
00764 #endif

```

```

00766
00767     while (L--) {
00768         if (s>0) {
00769 #ifdef _OPENMP
00770             if (N > OPENMP_HOLDS) {
00771 #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
00772                 for (myid=0; myid<nthreads; myid++) {
00773                     fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00774                     mybegin += i_1, myend += i_1;
00775                     for (i=mybegin; i<myend; i+=s) {
00776                         t=bval[i];
00777                         begin_row=ia[i], end_row=ia[i+1];
00778                         for (k=begin_row; k<end_row; k++) {
00779                             j=ja[k];
00780                             if (i!=j)
00781                                 t-=aval[k]*uval[j];
00782                             else
00783                                 d=aval[k];
00784                         }
00785                         if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00786                     }
00787                 }
00788             }
00789         }
00790     else {
00791 #endif
00792         for (i=i_1; i<=i_n; i+=s) {
00793             t=bval[i];
00794             begin_row=ia[i]; end_row=ia[i+1];
00795             for (k=begin_row; k<end_row; ++k) {
00796                 j=ja[k];
00797                 if (i!=j)
00798                     t-=aval[k]*uval[j];
00799                 else
00800                     d=aval[k];
00801             }
00802             if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00803         }
00804 #ifdef _OPENMP
00805     }
00806 #endif
00807     }
00808     else {
00809 #ifdef _OPENMP
00810         if (N > OPENMP_HOLDS) {
00811 #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
00812             for (myid=0; myid<nthreads; myid++) {
00813                 fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
00814                 mybegin = i_1 - mybegin, myend = i_1 - myend;
00815                 for (i=mybegin; i>myend; i+=s) {
00816                     t=bval[i];
00817                     begin_row=ia[i], end_row=ia[i+1];
00818                     for (k=begin_row; k<end_row; ++k) {
00819                         j=ja[k];
00820                         if (i!=j)
00821                             t-=aval[k]*uval[j];
00822                         else
00823                             d=aval[k];
00824                     }
00825                     if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00826                 }
00827             }
00828         }
00829     else {
00830 #endif
00831         for (i=i_1; i>=i_n; i+=s) {
00832             t=bval[i];
00833             begin_row=ia[i]; end_row=ia[i+1];
00834             for (k=begin_row; k<end_row; ++k) {
00835                 j=ja[k];
00836                 if (i!=j)
00837                     t-=aval[k]*uval[j];
00838                 else
00839                     d=aval[k];
00840             }
00841             if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00842         }
00843 #ifdef _OPENMP
00844     }
00845 #endif
00846     }

```

```

00847     } // end while
00848
00849     return;
00850 }
00851
00871 void fasp_smoother_dcsr_sor_cf (dvector      *u,
00872                                dCSRmat      *A,
00873                                dvector      *b,
00874                                INT          L,
00875                                const REAL   w,
00876                                INT          *mark,
00877                                const INT    order )
00878 {
00879     const INT    nrow = b->row; // number of rows
00880     const INT    *ia = A->IA, *ja=A->JA;
00881     const REAL   *aval = A->val,*bval=b->val;
00882     REAL         *uval = u->val;
00883
00884     // local variables
00885     INT    i,j,k,begin_row,end_row;
00886     REAL   t,d=0.0;
00887
00888 #ifdef _OPENMP
00889     INT    myid, mybegin, myend;
00890     INT    nthreads = fasp_get_num_threads();
00891 #endif
00892
00893     // F-point first
00894     if (order == -1) {
00895         while (L--) {
00896 #ifdef _OPENMP
00897             if (nrow > OPENMP_HOLDS) {
00898 #pragma omp parallel for private (myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
00899                 for (myid = 0; myid < nthreads; myid++) {
00900                     fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00901                     for (i = mybegin; i < myend; i++) {
00902                         if (mark[i] == 0 || mark[i] == 2) {
00903                             t = bval[i];
00904                             begin_row = ia[i], end_row = ia[i+1];
00905                             for (k = begin_row; k < end_row; k++) {
00906                                 j = ja[k];
00907                                 if (i!=j) t -= aval[k]*uval[j];
00908                                 else d = aval[k];
00909                             } // end for k
00910                             if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00911                         }
00912                     }
00913                 }
00914             } // end for i
00915         } else {
00916 #endif
00917             for (i = 0; i < nrow; i++) {
00918                 if (mark[i] == 0 || mark[i] == 2) {
00919                     t = bval[i];
00920                     begin_row = ia[i]; end_row = ia[i+1];
00921                     for (k = begin_row; k < end_row; k++) {
00922                         j = ja[k];
00923                         if (i!=j) t -= aval[k]*uval[j];
00924                         else d = aval[k];
00925                     } // end for k
00926                     if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00927                 }
00928             } // end for i
00929 #ifdef _OPENMP
00930         }
00931 #endif
00932
00933 #ifdef _OPENMP
00934         if (nrow > OPENMP_HOLDS) {
00935 #pragma omp parallel for private(myid, i, mybegin, myend, t, begin_row, end_row, k, j, d)
00936             for (myid = 0; myid < nthreads; myid++) {
00937                 fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00938                 for (i = mybegin; i < myend; i++) {
00939                     if (mark[i] == 1) {
00940                         t = bval[i];
00941                         begin_row = ia[i], end_row = ia[i+1];
00942                         for (k = begin_row; k < end_row; k++) {
00943                             j = ja[k];
00944                             if (i!=j) t -= aval[k]*uval[j];
00945                             else d = aval[k];
00946                         } // end for k

```

```

00947             if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00948         }
00949     } // end for i
00950 }
00951 }
00952 else {
00953 #endif
00954     for (i = 0; i < nrow; i++) {
00955         if (mark[i] == 1) {
00956             t = bval[i];
00957             begin_row = ia[i]; end_row = ia[i+1];
00958             for (k = begin_row; k < end_row; k++) {
00959                 j = ja[k];
00960                 if (i!=j) t -= avals[k]*uval[j];
00961                 else d = avals[k];
00962             } // end for k
00963             if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00964         }
00965     } // end for i
00966 #ifdef _OPENMP
00967 }
00968 #endif
00969 } // end while
00970 }
00971 else {
00972     while (L--) {
00973 #ifdef _OPENMP
00974         if (nrow > OPENMP_HOLDS) {
00975             #pragma omp parallel for private(myid, mybegin, myend, i, t, k, j, d, begin_row, end_row)
00976             for (myid = 0; myid < nthreads; myid++) {
00977                 fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
00978                 for (i = mybegin; i < myend; i++) {
00979                     if (mark[i] == 1) {
00980                         t = bval[i];
00981                         begin_row = ia[i]; end_row = ia[i+1];
00982                         for (k = begin_row; k < end_row; k++) {
00983                             j = ja[k];
00984                             if (i!=j) t -= avals[k]*uval[j];
00985                             else d = avals[k];
00986                         } // end for k
00987                         if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
00988                     }
00989                 } // end for i
00990             }
00991         }
00992     } else {
00993 #endif
00994         for (i = 0; i < nrow; i++) {
00995             if (mark[i] == 1) {
00996                 t = bval[i];
00997                 begin_row = ia[i]; end_row = ia[i+1];
00998                 for (k = begin_row; k < end_row; k++) {
00999                     j = ja[k];
01000                     if (i!=j) t -= avals[k]*uval[j];
01001                     else d = avals[k];
01002                 } // end for k
01003                 if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
01004             }
01005         } // end for i
01006 #ifdef _OPENMP
01007 }
01008 #endif
01009 }
01010 #ifdef _OPENMP
01011     if (nrow > OPENMP_HOLDS) {
01012         #pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
01013         for (myid = 0; myid < nthreads; myid++) {
01014             fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
01015             for (i = mybegin; i < myend; i++) {
01016                 if (mark[i] != 1) {
01017                     t = bval[i];
01018                     begin_row = ia[i]; end_row = ia[i+1];
01019                     for (k = begin_row; k < end_row; k++) {
01020                         j = ja[k];
01021                         if (i!=j) t -= avals[k]*uval[j];
01022                         else d = avals[k];
01023                     } // end for k
01024                     if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
01025                 }
01026             }
01027         }

```

```

01028         } // end for i
01029     else {
01030 #endif
01031         for (i = 0; i < nrow; i++) {
01032             if (mark[i] != 1) {
01033                 t = bval[i];
01034                 begin_row = ia[i]; end_row = ia[i+1];
01035                 for (k = begin_row; k < end_row; k++) {
01036                     j = ja[k];
01037                     if (i!=j) t -= aval[k]*uval[j];
01038                     else d = aval[k];
01039                 } // end for k
01040                 if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
01041             }
01042         } // end for i
01043 #ifdef _OPENMP
01044     }
01045 #endif
01046     } // end while
01047 }
01048
01049 return;
01050 }
01051
01065 void fasp_smoother_dcsr_ilu (dCSRmat *A,
01066                             dvector *b,
01067                             dvector *x,
01068                             void *data)
01069 {
01070     const INT m=A->row, m2=2*m, memneed=3*m;
01071     const ILU_data *iludata=(ILU_data *)data;
01072
01073     REAL *zz = iludata->work;
01074     REAL *zr = iludata->work+m;
01075     REAL *z = iludata->work+m2;
01076
01077     if (iludata->nwork<memneed) goto MEMERR;
01078
01079     {
01080         INT i, j, jj, begin_row, end_row;
01081         REAL *lu = iludata->luval;
01082         INT *ijlu = iludata->ijlu;
01083         REAL *xval = x->val, *bval = b->val;
01084
01085         fasp_darray_cp(m,bval,zr); fasp_blas_dcsr_aAxy(-1.0,A,xval,zr);
01086
01087         // forward sweep: solve unit lower matrix equation L*zz=zr
01088         zz[0]=zr[0];
01089         for (i=1;i<m;++i) {
01090             begin_row=ijlu[i]; end_row=ijlu[i+1];
01091             for (j=begin_row;j<end_row;++j) {
01092                 jj=ijlu[j];
01093                 if (jj<i) zr[i]-=lu[j]*zz[jj];
01094                 else break;
01095             }
01096             zz[i]=zr[i];
01097         }
01098
01099         // backward sweep: solve upper matrix equation U*z=zz
01100         z[m-1]=zz[m-1]*lu[m-1];
01101         for (i=m-2;i>=0;--i) {
01102             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
01103             for (j=end_row;j>=begin_row;--j) {
01104                 jj=ijlu[j];
01105                 if (jj>i) zz[i]-=lu[j]*z[jj];
01106                 else break;
01107             }
01108             z[i]=zz[i]*lu[i];
01109         }
01110
01111         fasp_blas_darray_axpy(m,1,z,xval);
01112     }
01113
01114     return;
01115 }
01116
01117 MEMERR:
01118     printf("### ERROR: ILU needs %d memory, only %d available! [%s:%d]\n",
01119           memneed, iludata->nwork, __FILE__, __LINE__);
01120     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01121 }
01122

```

```

01144 void fasp_smoother_dcsr_kaczmarz (dvector      *u,
01145                                     const INT    i_1,
01146                                     const INT    i_n,
01147                                     const INT    s,
01148                                     dCSRmat      *A,
01149                                     dvector      *b,
01150                                     INT          L,
01151                                     const REAL    w)
01152 {
01153     const INT    *ia=A->IA,*ja=A->JA;
01154     const REAL    *aval=A->val,*bval=b->val;
01155     REAL          uval=u->val;
01156
01157     // local variables
01158     INT    i,j,k,begin_row,end_row;
01159     REAL    temp1,temp2,alpha;
01160
01161 #ifdef _OPENMP
01162     const INT    N = ABS(i_n - i_1)+1;
01163     INT    myid, mybegin, myend;
01164     INT    nthreads = fasp_get_num_threads();
01165 #endif
01166     if (s > 0) {
01167         while (L--) {
01168             #ifdef _OPENMP
01169                 if (N > OPENMP_HOLDS) {
01170                     #pragma omp parallel for private(myid, mybegin, myend, i, temp1, temp2, begin_row, end_row, k, alpha, j)
01171                     for (myid=0; myid<nthreads; myid++) {
01172                         fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
01173                         mybegin += i_1, myend += i_1;
01174                         for (i=mybegin; i<myend; i+=s) {
01175                             temp1 = 0; temp2 = 0;
01176                             begin_row=ia[i], end_row=ia[i+1];
01177                             for (k=begin_row; k<end_row; k++) {
01178                                 j=ja[k];
01179                                 temp1 += aval[k]*aval[k];
01180                                 temp2 += aval[k]*uval[j];
01181                             } // end for k
01182                             alpha = (bval[i] - temp2)/temp1;
01183                             for (k=begin_row; k<end_row; ++k){
01184                                 j = ja[k];
01185                                 uval[j] += w*alpha*aval[k];
01186                             } // end for k
01187                         } // end for i
01188                     }
01189                 } else {
01190                     for (i=i_1;i<=i_n;i+=s) {
01191                         temp1 = 0; temp2 = 0;
01192                         begin_row=ia[i]; end_row=ia[i+1];
01193                         for (k=begin_row;k<end_row;++k) {
01194                             j=ja[k];
01195                             temp1 += aval[k]*aval[k];
01196                             temp2 += aval[k]*uval[j];
01197                         } // end for k
01198                         alpha = (bval[i] - temp2)/temp1;
01199                         for (k=begin_row;k<end_row;++k){
01200                             j = ja[k];
01201                             uval[j] += w*alpha*aval[k];
01202                         } // end for k
01203                     } // end for i
01204                 }
01205             } // end while
01206         } // if s
01207     } else {
01208         while (L--) {
01209             #ifdef _OPENMP
01210                 if (N > OPENMP_HOLDS) {
01211                     #pragma omp parallel for private(myid, mybegin, myend, i, temp1, temp2, begin_row, end_row, k, alpha, j)
01212                     for (myid=0; myid<nthreads; myid++) {
01213                         fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
01214                         mybegin = i_1 - mybegin, myend = i_1 - myend;
01215                         for (i=mybegin; i>myend; i+=s) {
01216                             temp1 = 0; temp2 = 0;

```

```

01225         begin_row=ia[i], end_row=ia[i+1];
01226         for (k=begin_row;k<end_row;++k) {
01227             j=ja[k];
01228             temp1 += aval[k]*aval[k];
01229             temp2 += aval[k]*uval[j];
01230         } // end for k
01231         alpha = (bval[i] - temp2)/temp1;
01232         for (k=begin_row;k<end_row;++k){
01233             j = ja[k];
01234             uval[j] += w*alpha*aval[k];
01235         }// end for k
01236     } // end for i
01237 }
01238 }
01239 else {
01240 #endif
01241     for (i=i_1;i>=i_n;i+=s) {
01242         temp1 = 0; temp2 = 0;
01243         begin_row=ia[i]; end_row=ia[i+1];
01244         for (k=begin_row;k<end_row;++k) {
01245             j=ja[k];
01246             temp1 += aval[k]*aval[k];
01247             temp2 += aval[k]*uval[j];
01248         } // end for k
01249         alpha = (bval[i] - temp2)/temp1;
01250         for (k=begin_row;k<end_row;++k){
01251             j = ja[k];
01252             uval[j] += w*alpha*aval[k];
01253         }// end for k
01254     } // end for i
01255 #ifdef _OPENMP
01256     }
01257 #endif
01258 } // end while
01259 } // end if
01260 } // end if
01261 return;
01262 }
01263 }
01264
01284 void fasp_smoother_dcsr_Lldiag (dvector      *u,
01285                                const INT      i_1,
01286                                const INT      i_n,
01287                                const INT      s,
01288                                dCSRmat        *A,
01289                                dvector        *b,
01290                                INT            L)
01291 {
01292     const INT    N = ABS(i_n - i_1)+1;
01293     const INT    *ia=A->IA, *ja=A->JA;
01294     const REAL    *aval=A->val, *bval=b->val;
01295     REAL          *uval=u->val;
01296
01297     // local variables
01298     INT    i, j, k, begin_row, end_row;
01299
01300 #ifdef _OPENMP
01301     INT    myid, mybegin, myend;
01302     INT    nthreads = fasp_get_num_threads();
01303 #endif
01304
01305     // Checks should be outside of for; t,d can be allocated before calling!!! --Chensong
01306     REAL *t = (REAL *)fasp_mem_calloc(N, sizeof(REAL));
01307     REAL *d = (REAL *)fasp_mem_calloc(N, sizeof(REAL));
01308
01309     while (L--) {
01310         if (s>0) {
01311 #ifdef _OPENMP
01312             if (N > OPENMP_HOLDS) {
01313 #pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, k, j)
01314                 for (myid=0; myid<nthreads; myid++) {
01315                     fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
01316                     mybegin += i_1, myend += i_1;
01317                     for (i=mybegin; i<myend; i+=s) {
01318                         t[i]=bval[i]; d[i]=0.0;
01319                         begin_row=ia[i], end_row=ia[i+1];
01320                         for (k=begin_row; k<end_row; k++) {
01321                             j=ja[k];
01322                             t[i]-=aval[k]*uval[j];
01323                             d[i]+=ABS(aval[k]);
01324                         }

```



```

01325     }
01326     }
01327 #pragma omp parallel for private(i)
01328     for (i=i_1;i<=i_n;i+=s) {
01329         if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
01330     }
01331     }
01332     else {
01333 #endif
01334         for (i=i_1;i<=i_n;i+=s) {
01335             t[i]=bval[i]; d[i]=0.0;
01336             begin_row=ia[i]; end_row=ia[i+1];
01337             for (k=begin_row;k<end_row;++k) {
01338                 j=ja[k];
01339                 t[i]-=aval[k]*uval[j];
01340                 d[i]+=ABS(aval[k]);
01341             }
01342         }
01343     }
01344     for (i=i_1;i<=i_n;i+=s) {
01345         if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
01346     }
01347 #ifdef _OPENMP
01348     }
01349 #endif
01350     }
01351     else {
01352 #ifdef _OPENMP
01353         if (N > OPENMP_HOLDS) {
01354 #pragma omp parallel for private(myid, mybegin, myend, i, k, j, begin_row, end_row)
01355             for (myid=0; myid<nthreads; myid++) {
01356                 fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
01357                 mybegin = i_1 - mybegin, myend = i_1 - myend;
01358                 for (i=mybegin; i>myend; i+=s) {
01359                     t[i]=bval[i]; d[i]=0.0;
01360                     begin_row=ia[i]; end_row=ia[i+1];
01361                     for (k=begin_row; k<end_row; k++) {
01362                         j=ja[k];
01363                         t[i]-=aval[k]*uval[j];
01364                         d[i]+=ABS(aval[k]);
01365                     }
01366                 }
01367             }
01368 #pragma omp parallel for private(i)
01369             for (i=i_1;i>=i_n;i+=s) {
01370                 if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
01371             }
01372         }
01373     else {
01374 #endif
01375         for (i=i_1;i>=i_n;i+=s) {
01376             t[i]=bval[i]; d[i]=0.0;
01377             begin_row=ia[i]; end_row=ia[i+1];
01378             for (k=begin_row;k<end_row;++k) {
01379                 j=ja[k];
01380                 t[i]-=aval[k]*uval[j];
01381                 d[i]+=ABS(aval[k]);
01382             }
01383         }
01384     }
01385     for (i=i_1;i>=i_n;i+=s) {
01386         if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
01387     }
01388 #ifdef _OPENMP
01389     }
01390 #endif
01391     }
01392 } // end while
01393
01394 fasp_mem_free(t); t = NULL;
01395 fasp_mem_free(d); d = NULL;
01396
01397 return;
01398 }
01399
01400
01401 #if 0
01422 static dCSRmat form_contractor (dCSRmat *A,
01423                                const INT smoother,
01424                                const INT steps,
01425                                const INT ndeg,

```

```

01426             const REAL relax,
01427             const REAL dtol)
01428 {
01429     const INT    n=A->row;
01430     INT          i;
01431
01432     REAL *work = (REAL *)fasp_mem_calloc(2*n,sizeof(REAL));
01433
01434     dvector b, x;
01435     b.row=x.row=n;
01436     b.val=work; x.val=work+n;
01437
01438     INT *index = (INT *)fasp_mem_calloc(n,sizeof(INT));
01439
01440     for (i=0; i<n; ++i) index[i]=i;
01441
01442     dCSRmat B = fasp_dcsr_create(n, n, n*n); // too much memory required, need to change!!
01443
01444     dCSRmat C, D;
01445
01446     for (i=0; i<n; ++i){
01447         // get i-th column
01448         fasp_dcsr_getcol(i, A, b.val);
01449
01450         // set x =0.0
01451         fasp_dvec_set(n, &x, 0.0);
01452
01453         // smooth
01454         switch (smoother) {
01455             case GS:
01456                 fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
01457                 break;
01458             case POLY:
01459                 fasp_smoother_dcsr_poly(A, &b, &x, n, ndeg, steps);
01460                 break;
01461             case JACOBI:
01462                 fasp_smoother_dcsr_jacobi(&x, 0, n-1, 1, A, &b, steps);
01463                 break;
01464             case SGS:
01465                 fasp_smoother_dcsr_sgs(&x, A, &b, steps);
01466                 break;
01467             case SOR:
01468                 fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
01469                 break;
01470             case SSOR:
01471                 fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
01472                 fasp_smoother_dcsr_sor(&x, n-1, 0,-1, A, &b, steps, relax);
01473                 break;
01474             case GSOR:
01475                 fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
01476                 fasp_smoother_dcsr_sor(&x, n-1, 0, -1, A, &b, steps, relax);
01477                 break;
01478             case SGSOR:
01479                 fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
01480                 fasp_smoother_dcsr_gs(&x, n-1, 0,-1, A, &b, steps);
01481                 fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
01482                 fasp_smoother_dcsr_sor(&x, n-1, 0,-1, A, &b, steps, relax);
01483                 break;
01484             default:
01485                 printf("### ERROR: Unknown smoother type! [%s:%d]\n",
01486                     __FILE__, __LINE__);
01487                 fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
01488         }
01489
01490         // store to B
01491         B.IA[i] = i*n;
01492         memcpy(&(B.JA[i*n]), index, n*sizeof(INT));
01493         memcpy(&(B.val[i*n]), x.val, x.row*sizeof(REAL));
01494     }
01495
01496     B.IA[n] = n*n;
01497
01498     // drop small entries
01499     compress_dCSRmat(&B, &D, dtol);
01500
01501     // get contractor
01502     fasp_dcsr_trans(&D, &C);
01503
01504     // clean up

```

```

01507     fasp_mem_free(work); work = NULL;
01508     fasp_dcsr_free(&B);
01509     fasp_dcsr_free(&D);
01510
01511     return C;
01512 }
01513 #endif
01514
01515 /*-----*/
01516 /*--      End of File      --*/
01517 /*-----*/

```

9.103 ltrSmootherCSRcr.c File Reference

Smoothers for [dCSRmat](#) matrices using compatible relaxation.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_smoother_dcsr_gscr](#) (INT pt, INT n, REAL *u, INT *ia, INT *ja, REAL *a, REAL *b, INT L, INT *CF)
Gauss Seidel method restriced to a block.

9.103.1 Detailed Description

Smoothers for [dCSRmat](#) matrices using compatible relaxation.

Note

Restricted smoothers for compatible relaxation, C/F smoothing, etc.

This file contains Level-2 (ltr) functions. It requires: [AuxMessage.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

// TODO: Need to optimize routines here! –Chensong

Definition in file [ltrSmootherCSRcr.c](#).

9.103.2 Function Documentation

9.103.2.1 fasp_smoother_dcsr_gscr()

```

void fasp_smoother_dcsr_gscr (
    INT pt,
    INT n,
    REAL * u,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    INT L,
    INT * CF )

```

Gauss Seidel method restriced to a block.

Parameters

<i>pt</i>	Relax type, e.g., cpt, fpt, etc..
<i>n</i>	Number of variables
<i>u</i>	Iterated solution
<i>ia</i>	Row pointer
<i>ja</i>	Column index
<i>a</i>	Pointers to sparse matrix values in CSR format
<i>b</i>	Pointer to right hand side
<i>L</i>	Number of iterations
<i>CF</i>	Marker for C, F points

Author

James Brannick

Date

09/07/2010

Note

Gauss Seidel CR smoother (Smoother_Type = 99)

Definition at line 48 of file [ltrSmootherCSRcr.c](#).

9.104 ltrSmootherCSRcr.c

[Go to the documentation of this file.](#)

```

00001
00018 #include <math.h>
00019
00020 #include "fasp.h"
00021 #include "fasp_functs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
00048 void fasp_smoother_dcsr_gscr (INT  pt,
00049                               INT  n,
00050                               REAL  *u,
00051                               INT  *ia,
00052                               INT  *ja,
00053                               REAL  *a,
00054                               REAL  *b,
00055                               INT  L,
00056                               INT  *CF)
00057 {
00058     INT i,j,k,l;
00059     REAL t, d=0;
00060
00061     for (l=0;l<L;++l) {
00062         for (i=0;i<n;++i) {
00063             if (CF[i] == pt) {
00064                 t=b[i];
00065                 for (k=ia[i];k<ia[i+1];++k) {
00066                     j=ja[k];
00067                     if (CF[j] == pt) {
00068                         if (i!=j) {
00069                             t-=a[k]*u[j];
00070                         }
00071                     }
00072                     else {
00073                         d=a[k];
00074                     }
00075                 }
00076             }
00077         }
00078     }
00079     u[i]=t/d;
00080 }

```

```

00074             if (ABS(d)>SMALLREAL) {
00075                 u[i]=t/d;
00076             }
00077             else {
00078                 printf("### ERROR: Diagonal entry_%d (%e) close to 0!\n",
00079                     i, d);
00080                 fasp_chkerr(ERROR_MISC, __FUNCTION__);
00081             }
00082         }
00083     }
00084 }
00085     else {
00086         u[i]=0.e0;
00087     }
00088 }
00089 }
00090 }
00091
00092 /*-----*/
00093 /*--      End of File      --*/
00094 /*-----*/

```

9.105 ltrSmootherCSRpoly.c File Reference

Smoothers for [dCSRmat](#) matrices using poly. approx. to A^{-1} .

```

#include <math.h>
#include <time.h>
#include <float.h>
#include <limits.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_smoother_dcsr_poly](#) ([dCSRmat](#) *Amat, [dvector](#) *brhs, [dvector](#) *usol, [INT](#) n, [INT](#) ndeg, [INT](#) L)
poly approx to A^{-1} as MG smoother
- void [fasp_smoother_dcsr_poly_old](#) ([dCSRmat](#) *Amat, [dvector](#) *brhs, [dvector](#) *usol, [INT](#) n, [INT](#) ndeg, [INT](#) L)
poly approx to A^{-1} as MG smoother: JK<Z2010

9.105.1 Detailed Description

Smoothers for [dCSRmat](#) matrices using poly. approx. to A^{-1} .

Note

This file contains Level-2 (ltr) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [BlaArray.c](#), and [BlaSpmvCSR.c](#)

Reference: Johannes K. Kraus, Panayot S. Vassilevski, Ludmil T. Zikatanov Polynomial of best uniform approximation to x^{-1} and smoothing in two-level methods, 2013.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Warning

Do NOT use auto-indentation in this file!

// TODO: Need to optimize routines here! –Chensong

Definition in file [ltrSmootherCSRpoly.c](#).

9.105.2 Function Documentation

9.105.2.1 fasp_smoother_dcsr_poly()

```
void fasp_smoother_dcsr_poly (
    dCSRmat * Amat,
    dvector * brhs,
    dvector * usol,
    INT n,
    INT ndeg,
    INT L )
```

poly approx to A^{-1} as MG smoother

Parameters

<i>Amat</i>	Pointer to stiffness matrix, consider square matrix.
<i>brhs</i>	Pointer to right hand side
<i>usol</i>	Pointer to solution
<i>n</i>	Problem size
<i>ndeg</i>	Degree of poly
<i>L</i>	Number of iterations

Author

Fei Cao, Xiaozhe Hu

Date

05/24/2012

Definition at line 67 of file [ltrSmootherCSRpoly.c](#).

9.105.2.2 fasp_smoother_dcsr_poly_old()

```
void fasp_smoother_dcsr_poly_old (
    dCSRmat * Amat,
    dvector * brhs,
    dvector * usol,
    INT n,
    INT ndeg,
    INT L )
```

poly approx to A^{-1} as MG smoother: JK<Z2010

Parameters

<i>Amat</i>	Pointer to stiffness matrix
<i>brhs</i>	Pointer to right hand side
<i>usol</i>	Pointer to solution
<i>n</i>	Problem size
<i>ndeg</i>	Degree of poly
<i>L</i>	Number of iterations

Author

James Brannick and Ludmil T Zikatanov

Date

06/28/2010

Modified by Chunsheng Feng, Zheng Li on 10/18/2012

Definition at line 165 of file ltrSmootherCSRpoly.c.

9.106 ltrSmootherCSRpoly.c

[Go to the documentation of this file.](#)

```

00001
00023 #include <math.h>
00024 #include <time.h>
00025 #include <float.h>
00026 #include <limits.h>
00027
00028 #ifdef _OPENMP
00029 #include <omp.h>
00030 #endif
00031
00032 #include "fasp.h"
00033 #include "fasp_functs.h"
00034
00035 /*-----*/
00036 /*--  Declare Private Functions  --*/
00037 /*-----*/
00038
00039 static void bminax (REAL *,INT *,INT *, REAL *, REAL *,INT *, REAL *);
00040 static void Diagin (dCSRmat *, REAL *);
00041 static REAL DinvaNorminf (dCSRmat *, REAL *);
00042 static void Diagx (REAL *, INT, REAL *, REAL *);
00043 static void Rr (dCSRmat *, REAL *, REAL *, REAL *, REAL *, REAL *, REAL *, REAL *, INT);
00044 static void fasp_aux_uuplv0_ (REAL *, REAL *, INT *);
00045 static void fasp_aux_norml_ (INT *, INT *, REAL *, INT *, REAL *);
00046
00047 /*-----*/
00048 /*--      Public Function      --*/
00049 /*-----*/
00050
00067 void fasp_smoother_dcsr_poly (dCSRmat *Amat,
00068                             dvector *brhs,
00069                             dvector *usol,
00070                             INT      n,
00071                             INT      ndeg,
00072                             INT      L)
00073 {
00074     // local variables
00075     INT i;
00076     REAL *b = brhs->val, *u = usol->val;
00077     REAL *Dinv = NULL, *r = NULL, *rbar = NULL, *v0 = NULL, *v1 = NULL;
00078     REAL *error = NULL, *k = NULL;
00079     REAL mu0, mul, smu0, smul;
00080
00081     /* allocate memory */
00082     Dinv = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00083     r = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00084     rbar = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00085     v0 = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00086     v1 = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00087     error = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00088     k = (REAL *) fasp_mem_malloc(6,sizeof(REAL)); // coefficients for calculation
00089
00090     // get the inverse of the diagonal of A
00091     Diagin(Amat, Dinv);
00092
00093     // set up parameter
00094     mu0 = DinvaNorminf(Amat, Dinv); // get the inf norm of Dinv*A;
00095
00096     mu0 = 1.0/mu0; mul = 4.0*mu0; // default set 8;
00097     smu0 = sqrt(mu0); smul = sqrt(mul);
00098

```

```

00099     k[1] = (mu0+mu1)/2.0;
00100     k[2] = (smu0 + smu1)*(smu0 + smu1)/2.0;
00101     k[3] = mu0 * mu1;
00102
00103     // 4.0*mu0*mu1/(sqrt(mu0)+sqrt(mu1))/(sqrt(mu0)+sqrt(mu1));
00104     k[4] = 2.0*k[3]/k[2];
00105
00106     // square of (sqrt(kappa)-1)/(sqrt(kappa)+1);
00107     k[5] = (mu1-2.0*smu0*smu1+mu0)/(mu1+2.0*smu0*smu1+mu0);
00108
00109 #if DEBUG_MODE > 0
00110     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00111 #endif
00112
00113     // Update
00114     for ( i=0; i<L; i++ ) {
00115         // get residual
00116         fasp_blas_dcsr_mxv(Amat, u, r); // r= Amat*u;
00117         fasp_blas_darray_axpyz(n, -1, r, b, r); // r= -r+b;
00118
00119         // Get correction error = R*r
00120         Rr(Amat, Dinv, r, rbar, v0, v1, error, k, ndeg);
00121
00122         // update solution
00123         fasp_blas_darray_axpy(n, 1, error, u);
00124     }
00125
00126
00127 #if DEBUG_MODE > 1
00128     printf("### DEBUG: Degree of polysmoothing is: %d\n",ndeg);
00129 #endif
00130
00131     // free memory
00132     fasp_mem_free(Dinv); Dinv = NULL;
00133     fasp_mem_free(r); r = NULL;
00134     fasp_mem_free(rbar); rbar = NULL;
00135     fasp_mem_free(v0); v0 = NULL;
00136     fasp_mem_free(v1); v1 = NULL;
00137     fasp_mem_free(error); error = NULL;
00138     fasp_mem_free(k); k = NULL;
00139
00140 #if DEBUG_MODE > 0
00141     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00142 #endif
00143
00144     return;
00145 }
00146
00165 void fasp_smoother_dcsr_poly_old (dCSRmat *Amat,
00166                                   dvector *brhs,
00167                                   dvector *usol,
00168                                   INT n,
00169                                   INT ndeg,
00170                                   INT L)
00171 {
00172     INT *ia=Amat->IA,*ja=Amat->JA;
00173     INT i,j,k,it,jk,iaa,iab,ndeg0; // id and ij for scaling of A
00174
00175     REAL *a=Amat->val, *b=brhs->val, *u=usol->val;
00176     REAL *v,*v0,*r,*vsave; // one can get away without r as well;
00177     REAL smaxa,smina,delinv,s,smu0,smu1,skappa,th,thl,sq;
00178     REAL ri,ari,vj,ravj,snj,sm,sm01,smsqrt,delta,delta2,chi;
00179
00180 #ifdef _OPENMP
00181     // variables for OpenMP
00182     INT myid, mybegin, myend;
00183     INT nthreads = fasp_get_num_threads();
00184 #endif
00185
00186 #if DEBUG_MODE > 0
00187     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00188 #endif
00189
00190     /* WORKING MEM */
00191     v = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00192     v0 = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00193     vsave = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00194     r = (REAL *) fasp_mem_malloc(n,sizeof(REAL));
00195
00196     /* COMPUTE PARAMS*/
00197     // min INT for approx -- could be done upfront

```



```

00198 // i.e., only once per level... only norml ...
00199 fasp_aux_norml_(ia, ja, a, &n, &smaxa);
00200 smina=smaxa/8;
00201 delinv=(smaxa+smina)/(smaxa-smina);
00202 th=delinv+sqrt(delinv*delinv-1e+00);
00203 th1=1e+00/th;
00204 sq=(th-th1)*(th-th1);
00205 //
00206 ndeg0=(int)floor(log(2*(2e0+th+th1)/sq)/log(th)+1e0);
00207 if (ndeg0 < ndeg) ndeg0=ndeg;
00208 //
00209 smu0=1e+00/smaxa;
00210 smul=1e+00/smina;
00211 skappa=sqrt(smaxa/smina);
00212 delta=(skappa-1e+00)/(skappa+1);
00213 delta2=delta*delta;
00214 s=sqrt(smu0)+sqrt(smul);
00215 s=s*s;
00216 smsqrt=0.5e+00*s;
00217 chi=4e+00*smu0*smul/s;
00218 sm=0.5e+00*(smu0+smul);
00219 sm01=smu0*smul;
00220
00221 #if DEBUG_MODE > 1
00222 printf("### DEBUG: Degree of polysmoothing is: %d\n", ndeg);
00223 #endif
00224
00225 /* BEGIN POLY ITS */
00226
00227 /* auv_(ia, ja, a, u, u, &n, &err0); NA: u = 0 */
00228 //bminax(b, ia, ja, a, u, &n, r);
00229 //for (i=0; i < n; ++i) {res0 += r[i]*r[i];}
00230 //res0=sqrt(res0);
00231
00232 for (it = 0 ; it < L; it++) {
00233     bminax(b, ia, ja, a, u, &n, r);
00234 #ifdef _OPENMP
00235 #pragma omp parallel for private(myid, mybegin, myend, i, iaa, iab, ari, jk, j, ri) if (n>OPENMP_HOLDS)
00236     for (myid=0; myid<nthreads; ++myid) {
00237         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00238         for (i=mybegin; i<myend; ++i) {
00239 #else
00240         for (i=0; i < n ; ++i) {
00241 #endif
00242             iaa = ia[i];
00243             iab = ia[i+1];
00244             ari=0e+00; /* ari is (A*r)[i] */
00245             if (iab > iaa) {
00246                 for (jk = iaa; jk < iab; jk++) {
00247                     j=ja[jk];
00248                     ari += a[jk] * r[j];
00249                 }
00250             }
00251             ri=r[i];
00252             v0[i]=sm*ri;
00253             v[i]=smsqrt*ri-sm01*ari;
00254         }
00255 #ifdef _OPENMP
00256     }
00257 #endif
00258     for (i=1; i < ndeg0; ++i) {
00259         //for (j=0; j < n ; ++j) vsave[j]=v[j];
00260         fasp_darray_cp(n, v, vsave);
00261     }
00262 #ifdef _OPENMP
00263 #pragma omp parallel for private(myid, mybegin, myend, j, ravj, iaa, iab, jk, k, vj, snj) if (n>OPENMP_HOLDS)
00264     for (myid=0; myid<nthreads; ++myid) {
00265         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00266         for (j=mybegin; j<myend; ++j) {
00267 #else
00268         for (j=0; j < n ; ++j) {
00269 #endif
00270             /* ravj = (r- A*v)[j] */
00271             ravj= r[j];
00272             iaa = ia[j];
00273             iab = ia[j+1];
00274             if (iab > iaa) {
00275                 for (jk = iaa; jk < iab; jk++) {
00276                     k=ja[jk];
00277                     ravj -= a[jk] * vsave[k];
00278                 }

```

```

00279         }
00280         vj=v[j];
00281         snj = chi*ravj+delta2*(vj-v0[j]);
00282         v0[j]=vj;
00283         v[j]=vj+snj;
00284     }
00285 }
00286 #ifdef _OPENMP
00287 }
00288 #endif
00289 fasp_aux_uuplv0_(u,v,&n);
00290 //bminax(b,ia,ja,a,u,&n,r);
00291 //for (i=0; i < n ; ++i)
00292 //resk += r[i]*r[i];
00293 //resk=sqrt(resk);
00294 //fprintf("\nres0=%12.5g\n",res0);
00295 //fprintf("\nresk=%12.5g\n",resk);
00296 //res0=resk;
00297 //resk=0.0e0;
00298 }
00299
00300 fasp_mem_free(v);      v      = NULL;
00301 fasp_mem_free(v0);     v0     = NULL;
00302 fasp_mem_free(r);      r      = NULL;
00303 fasp_mem_free(vsav);   vsave  = NULL;
00304
00305 #if DEBUG_MODE > 0
00306 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00307 #endif
00308
00309 return;
00310 }
00311
00312 /*-----*/
00313 /*--      Private Functions      --*/
00314 /*-----*/
00315
00316 static void bminax (REAL *b,
00317                   INT  *ia,
00318                   INT  *ja,
00319                   REAL *a,
00320                   REAL *x,
00321                   INT  *nn,
00322                   REAL *res)
00323 {
00324     /* Computes b-A*x */
00325
00326     INT i,j,jk,iaa,iab;
00327     INT n;
00328     REAL u;
00329     n=nn;
00330
00331 #ifdef _OPENMP
00332     // variables for OpenMP
00333     INT myid, mybegin, myend;
00334     INT nthreads = fasp_get_num_threads();
00335 #endif
00336
00337 #ifdef _OPENMP
00338 #pragma omp parallel for private(myid,mybegin,myend,i,iaa,iab,u,jk,j) if(n>OPENMP_HOLDS)
00339     for (myid=0; myid<nthreads; ++myid) {
00340         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00341         for (i=mybegin; i<myend; ++i) {
00342             #else
00343             for (i=0; i < n ; ++i) {
00344             #endif
00345                 iaa = ia[i];
00346                 iab = ia[i+1];
00347                 u = b[i];
00348                 if(iab > iaa)
00349                     for (jk = iaa; jk < iab; jk++) {
00350                         j=ja[jk];
00351                         u -= a[jk] * x[j];
00352                     }
00353                 res[i] = u;
00354             }
00355 #endif
00356 #ifdef _OPENMP
00357 }
00358 #endif
00359 return;
00360 }

```

```

00378
00392 static void Diaginv (dCSRmat *Amat,
00393                     REAL *Dinv)
00394 {
00395     const INT n = Amat->row;
00396     const INT *ia = Amat->IA, *ja = Amat->JA;
00397     const REAL *a = Amat->val;
00398     INT i, j;
00399
00400 #ifdef _OPENMP
00401 #pragma omp parallel for private(j) if(n>OPENMP_HOLDS)
00402 #endif
00403     for (i=0; i<n; i++) {
00404         for(j=ia[i]; j<ia[i+1]; j++) {
00405             if(i==ja[j]) // find the diagonal
00406                 break;
00407         }
00408         Dinv[i] = 1.0/a[j];
00409     }
00410     return;
00411 }
00412
00428 static REAL DinvAnorminf (dCSRmat *Amat,
00429                          REAL *Dinv)
00430 {
00431     //local variable
00432     const INT n = Amat->row;
00433     const INT *ia = Amat->IA;
00434     const REAL *a = Amat->val;
00435
00436     INT i, j;
00437     REAL norm, temp;
00438
00439 #ifdef _OPENMP
00440     // variables for OpenMP
00441     INT myid, mybegin, myend;
00442     REAL sub_norm = 0.0;
00443     INT nthreads = fasp_get_num_threads();
00444 #endif
00445     norm = 0.0;
00446
00447     // get the infinity norm of Dinv*A
00448 #ifdef _OPENMP
00449 #pragma omp parallel for private(myid,mybegin,myend,i,temp,sub_norm) if(n>OPENMP_HOLDS)
00450     for (myid=0; myid<nthreads; ++myid) {
00451         fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00452         sub_norm = 0.0;
00453         for (i=mybegin; i<myend; ++i) {
00454             for (j=ia[i]; j<ia[i+1]; j++) {
00455                 temp += ABS(a[j]);
00456             }
00457             temp *= Dinv[i]; // temp is the L1 norm of the ith row of Dinv*A;
00458             sub_norm = MAX(sub_norm, temp);
00459         }
00460     }
00461 #endif
00462     norm = MAX(norm, sub_norm);
00463 #ifdef _OPENMP
00464 #pragma omp critical(norm)
00465     norm = MAX(norm, sub_norm);
00466 #endif
00467     return norm;
00468 }
00469
00470 static void Diagx (REAL *Dinv,
00471                  INT n,
00472                  REAL *x,
00473                  REAL *b)
00474 {
00475     INT i;
00476
00477     // Variables for OpenMP
00478     SHORT nthreads = 1, use_omp = FALSE;

```

```

00502     INT myid, mybegin, myend;
00503
00504 #ifdef _OPENMP
00505     if (n > OPENMP_HOLDS) {
00506         use_openmp = TRUE;
00507         nthreads = fasp_get_num_threads();
00508     }
00509 #endif
00510
00511     if (use_openmp) {
00512 #ifdef _OPENMP
00513 #pragma omp parallel for private(myid, mybegin, myend, i)
00514 #endif
00515         for (myid = 0; myid < nthreads; myid++) {
00516             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00517             for (i = mybegin; i < myend; i++) {
00518                 b[i] = Dinv[i] * x[i];
00519             }
00520         }
00521     }
00522     else {
00523         for (i=0; i<n; i++) {
00524             b[i] = Dinv[i] * x[i];
00525         }
00526     }
00527     return;
00528 }
00529
00551 static void Rr (dCSRmat *Amat,
00552                REAL      *Dinv,
00553                REAL      *r,
00554                REAL      *rbar,
00555                REAL      *v0,
00556                REAL      *v1,
00557                REAL      *vnew,
00558                REAL      *k,
00559                INT       m)
00560 {
00561     // local variables
00562     const INT    n = Amat->nrow;
00563     INT i, j;
00564
00565 #ifdef _OPENMP
00566     // variables for OpenMP
00567     INT myid, mybegin, myend;
00568     INT nthreads = fasp_get_num_threads();
00569 #endif
00570
00571     //1 set up rbar
00572     Diagx(Dinv, n, r, rbar); // rbar = Dinv * r;
00573
00574     //2 set up v0, v1;
00575     fasp_blas_dcsr_mxv(Amat, rbar, v1); //v1= A*rbar;
00576     Diagx(Dinv, n, v1, v1); // v1=Dinv *v1;
00577
00578 #ifdef _OPENMP
00579 #pragma omp parallel for if(n>OPENMP_HOLDS)
00580 #endif
00581     for(i=0;i<n;i++) {
00582         v0[i] = k[1] * rbar[i];
00583         v1[i] = k[2] * rbar[i] - k[3] * v1[i];
00584     }
00585
00586     //3 iterate to get v_(j+1)
00587
00588     for (j=1;j<m;j++) {
00589         fasp_blas_dcsr_mxv(Amat, v1, rbar); //rbar= A*v_(j);
00590
00591 #ifdef _OPENMP
00592 #pragma omp parallel for private(myid,mybegin,myend,i) if(n>OPENMP_HOLDS)
00593 #endif
00594         for (myid=0; myid<nthreads; ++myid) {
00595             fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
00596             for (i=mybegin; i<myend; ++i) {
00597 #else
00598                 for(i=0;i<n;i++) {
00599 #endif
00599                 rbar[i] = (r[i] - rbar[i])*Dinv[i]; // indeed rbar=Dinv*(r-A*v_(j));
00600                 vnew[i] = v1[i] + k[5] *(v1[i] - v0[i]) + k[4] * rbar[i]; // compute v_(j+1)
00601                 // prepare for next cycle
00602                 v0[i]=v1[i];
00603                 v1[i]=vnew[i];

```

```

00604     }
00605 #ifdef _OPENMP
00606     }
00607 #endif
00608 }
00609 }
00610
00611 static void fasp_aux_uuplv0_ (REAL *u,
00612                             REAL *v,
00613                             INT *n)
00614 {
00615     /*
00616     This computes y = y + x.
00617     */
00618     INT i;
00619     for ( i=0; i < *n; i++ ) u[i] += v[i];
00620     return;
00621 }
00622
00623 static void fasp_aux_norm1_ (INT *ia,
00624                             INT *ja,
00625                             REAL *a,
00626                             INT *nn,
00627                             REAL *alnorm)
00628 {
00629     INT n,i,jk,iaa,iab;
00630     REAL sum,s;
00631     /* computes one norm of a matrix a and stores it in the variable
00632     pointed to by *alnorm*/
00633     n = *nn;
00634     s = 0.0;
00635     for ( i=0; i < n ; i++ ) {
00636         iaa = ia[i];
00637         iab = ia[i+1];
00638         sum = 0e+00;
00639         for ( jk = iaa; jk < iab; jk++ ) sum += fabs(a[jk]);
00640         if ( sum > s ) s = sum;
00641     }
00642     *alnorm=s;
00643 }
00644
00645 /*-----*/
00646 /*--      End of File      --*/
00647 /*-----*/

```

9.107 ItrSmootherSTR.c File Reference

Smoothers for [dSTRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void [fasp_smoother_dstr_jacobi](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u)
Jacobi method as the smoother.
- void [fasp_smoother_dstr_jacobi1](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)
Jacobi method as the smoother with diag_inv given.
- void [fasp_smoother_dstr_gs](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, const [INT](#) order, [INT](#) *mark)
Gauss-Seidel method as the smoother.
- void [fasp_smoother_dstr_gs1](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, const [INT](#) order, [INT](#) *mark, [REAL](#) *diaginv)
Gauss-Seidel method as the smoother with diag_inv given.
- void [fasp_smoother_dstr_gs_ascend](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)
Gauss-Seidel method as the smoother in the ascending manner.
- void [fasp_smoother_dstr_gs_descend](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv)

Gauss-Seidel method as the smoother in the descending manner.

- void [fasp_smoother_dstr_gs_order](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark)

Gauss method as the smoother in the user-defined order.

- void [fasp_smoother_dstr_gs_cf](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark, const [INT](#) order)

Gauss method as the smoother in the C-F manner.

- void [fasp_smoother_dstr_sor](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, const [INT](#) order, [INT](#) *mark, const [REAL](#) weight)

SOR method as the smoother.

- void [fasp_smoother_dstr_sor1](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, const [INT](#) order, [INT](#) *mark, [REAL](#) *diaginv, const [REAL](#) weight)

SOR method as the smoother.

- void [fasp_smoother_dstr_sor_ascend](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [REAL](#) weight)

SOR method as the smoother in the ascending manner.

- void [fasp_smoother_dstr_sor_descend](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [REAL](#) weight)

SOR method as the smoother in the descending manner.

- void [fasp_smoother_dstr_sor_order](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark, [REAL](#) weight)

SOR method as the smoother in the user-defined order.

- void [fasp_smoother_dstr_sor_cf](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [REAL](#) *diaginv, [INT](#) *mark, const [INT](#) order, const [REAL](#) weight)

SOR method as the smoother in the C-F manner.

- void [fasp_generate_diaginv_block](#) ([dSTRmat](#) *A, [ivector](#) *neigh, [dvector](#) *diaginv, [ivector](#) *pivot)

Generate inverse of diagonal block for block smoothers.

- void [fasp_smoother_dstr_swz](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [dvector](#) *diaginv, [ivector](#) *pivot, [ivector](#) *neigh, [ivector](#) *order)

9.107.1 Detailed Description

Smoothers for [dSTRmat](#) matrices.

Note

This file contains Level-2 (ltr) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaSmallMat.c](#), [BlaSmallMatInv.c](#), [BlaSmallMatLU.c](#), and [BlaSpmvSTR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [ltrSmootherSTR.c](#).

9.107.2 Function Documentation

9.107.2.1 fasp_generate_diaginv_block()

```
void fasp_generate_diaginv_block (
    dSTRmat * A,
    ivector * neigh,
    dvector * diaginv,
    ivector * pivot )
```

Generate inverse of diagonal block for block smoothers.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>neigh</i>	Pointer to ivector: neighborhoods
<i>diaginv</i>	Pointer to dvector: the inverse of the diagonals
<i>pivot</i>	Pointer to ivector: the pivot of diagonal blocks

Author

Xiaozhe Hu

Date

10/01/2011

Definition at line [1543](#) of file [ltrSmootherSTR.c](#).**9.107.2.2 fasp_smoother_dstr_gs()**

```
void fasp_smoother_dstr_gs (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    const INT order,
    INT * mark )
```

Gauss-Seidel method as the smoother.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending manner DESCEND 21: in descending manner If mark != NULL USERDEFINED 0 : in the user-defined manner CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points
<i>mark</i>	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [217](#) of file [ltrSmootherSTR.c](#).**9.107.2.3 fasp_smoother_dstr_gs1()**

```
void fasp_smoother_dstr_gs1 (
    dSTRmat * A,
    dvector * b,
```

```

dvector * u,
const INT order,
INT * mark,
REAL * diaginv )

```

Gauss-Seidel method as the smoother with `diag_inv` given.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending manner DESCEND 21: in descending manner If mark != NULL USERDEFINED 0 : in the user-defined manner CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points
<i>mark</i>	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [277](#) of file [ltrSmootherSTR.c](#).

9.107.2.4 fasp_smoother_dstr_gs_ascend()

```

void fasp_smoother_dstr_gs_ascend (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )

```

Gauss-Seidel method as the smoother in the ascending manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [322](#) of file [ltrSmootherSTR.c](#).

9.107.2.5 fasp_smoother_dstr_gs_cf()

```
void fasp_smoother_dstr_gs_cf (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    INT * mark,
    const INT order )
```

Gauss method as the smoother in the C-F manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
<i>mark</i>	Pointer to the user-defined order array
<i>order</i>	Flag to indicate the order for smoothing CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [680](#) of file [ltrSmootherSTR.c](#).

9.107.2.6 fasp_smoother_dstr_gs_descend()

```
void fasp_smoother_dstr_gs_descend (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )
```

Gauss-Seidel method as the smoother in the descending manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 438 of file [ltrSmootherSTR.c](#).**9.107.2.7 fasp_smoother_dstr_gs_order()**

```
void fasp_smoother_dstr_gs_order (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginvs,
    INT * mark )
```

Gauss method as the smoother in the user-defined order.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
<i>mark</i>	Pointer to the user-defined order array

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 556 of file [ltrSmootherSTR.c](#).**9.107.2.8 fasp_smoother_dstr_jacobi()**

```
void fasp_smoother_dstr_jacobi (
    dSTRmat * A,
    dvector * b,
    dvector * u )
```

Jacobi method as the smoother.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 43 of file [ltrSmootherSTR.c](#).**9.107.2.9 fasp_smoother_dstr_jacobi1()**

```
void fasp_smoother_dstr_jacobi1 (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv )
```

Jacobi method as the smoother with diag_inv given.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 92 of file [ltrSmootherSTR.c](#).**9.107.2.10 fasp_smoother_dstr_sor()**

```
void fasp_smoother_dstr_sor (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    const INT order,
    INT * mark,
    const REAL weight )
```

SOR method as the smoother.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending manner DESCEND 21: in descending manner If mark != NULL USERDEFINED 0 : in the user-defined manner CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points
<i>mark</i>	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 873 of file [ltrSmootherSTR.c](#).**9.107.2.11 fasp_smoother_dstr_sor1()**

```
void fasp_smoother_dstr_sor1 (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    const INT order,
    INT * mark,
    REAL * diaginv,
    const REAL weight )
```

SOR method as the smoother.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>order</i>	Flag to indicate the order for smoothing If mark = NULL ASCEND 12: in ascending manner DESCEND 21: in descending manner If mark != NULL USERDEFINED 0 : in the user-defined manner CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points
<i>mark</i>	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
<i>diaginv</i>	Inverse of the diagonal entries
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 935 of file [ltrSmootherSTR.c](#).**9.107.2.12 fasp_smoother_dstr_sor_ascend()**

```
void fasp_smoother_dstr_sor_ascend (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    REAL weight )
```

SOR method as the smoother in the ascending manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diagin</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [981](#) of file [ltrSmootherSTR.c](#).

9.107.2.13 fasp_smoother_dstr_sor_cf()

```
void fasp_smoother_dstr_sor_cf (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    INT * mark,
    const INT order,
    const REAL weight )
```

SOR method as the smoother in the C-F manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
<i>mark</i>	Pointer to the user-defined order array
<i>order</i>	Flag to indicate the order for smoothing CPFIRST 1 : C-points first and then F-points FPFIRST -1 : F-points first and then C-points
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [1355](#) of file [ltrSmootherSTR.c](#).

9.107.2.14 fasp_smoother_dstr_sor_descend()

```
void fasp_smoother_dstr_sor_descend (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    REAL weight )
```

SOR method as the smoother in the descending manner.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of <i>A</i> when $(A->nc)>1$, and NULL when $(A->nc)=1$
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line [1102](#) of file [ltrSmootherSTR.c](#).

9.107.2.15 fasp_smoother_dstr_sor_order()

```
void fasp_smoother_dstr_sor_order (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    REAL * diaginv,
    INT * mark,
    REAL weight )
```

SOR method as the smoother in the user-defined order.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>diaginv</i>	All the inverse matrices for all the diagonal block of <i>A</i> when $(A->nc)>1$, and NULL when $(A->nc)=1$
<i>mark</i>	Pointer to the user-defined order array
<i>weight</i>	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 1224 of file [ltrSmootherSTR.c](#).**9.107.2.16 fasp_smoother_dstr_swz()**

```
void fasp_smoother_dstr_swz (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    dvector * diaginv,
    ivector * pivot,
    ivector * neigh,
    ivector * order )
```

Definition at line 1665 of file [ltrSmootherSTR.c](#).**9.108 ltrSmootherSTR.c**[Go to the documentation of this file.](#)

```
00001
00015 #include <math.h>
00016
00017 #include "fasp.h"
00018 #include "fasp_functs.h"
00019
00020 /*-----*/
00021 /*--  Declare Private Functions  --*/
00022 /*-----*/
00023
00024 static void blkcontr2 (INT, INT, INT, INT, REAL *, REAL *, REAL *);
00025 static void aAxpby (REAL, REAL, INT, REAL *, REAL *, REAL *);
00026
00027 /*-----*/
00028 /*--      Public Functions      --*/
00029 /*-----*/
00030
00043 void fasp_smoother_dstr_jacobi (dSTRmat *A,
00044                                dvector *b,
00045                                dvector *u)
00046 {
00047     INT    nc    = A->nc; // size of each block (number of components)
00048     INT    ngrid = A->ngrid; // number of grids
00049     REAL  *diag  = A->diag; // Diagonal entries
00050     REAL  *diaginv = NULL; // Diagonal inverse, same size and storage scheme as A->diag
00051
00052     INT nc2    = nc*nc;
00053     INT size   = nc2*ngrid;
00054     INT block  = 0;
00055     INT start  = 0;
00056
00057     if (nc > 1) {
00058         // allocate memory
00059         diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00060
00061         // diaginv = diag;
00062         fasp_darray_cp(size, diag, diaginv);
00063
00064         // generate diaginv
00065         for (block = 0; block < ngrid; block++) {
00066             fasp_smat_inv(diaginv+start, nc);
00067             start += nc2;
00068         }
00069     }
```

```

00069     }
00070
00071     fasp_smoother_dstr_jacobil(A, b, u, diagininv);
00072
00073     fasp_mem_free(diagininv); diagininv = NULL;
00074 }
00075
00076
00092 void fasp_smoother_dstr_jacobil (dSTRmat *A,
00093                                dvector *b,
00094                                dvector *u,
00095                                REAL *diaginv)
00096 {
00097     // information of A
00098     INT ngrid = A->ngrid; // number of grids
00099     INT nc = A->nc; // size of each block (number of components)
00100     INT nband = A->nband; // number of off-diag band
00101     INT *offsets = A->offsets; // offsets of the off-diagonals
00102     REAL *diag = A->diag; // Diagonal entries
00103     REAL **offdiag = A->offdiag; // Off-diagonal entries
00104
00105     // values of dvector b and u
00106     REAL *b_val = b->val;
00107     REAL *u_val = u->val;
00108
00109     // local variables
00110     INT block = 0;
00111     INT point = 0;
00112     INT band = 0;
00113     INT width = 0;
00114     INT size = nc*ngrid;
00115     INT nc2 = nc*nc;
00116     INT start = 0;
00117     INT column = 0;
00118     INT start_data = 0;
00119     INT start_DATA = 0;
00120     INT start_vecb = 0;
00121     INT start_vecu = 0;
00122
00123     // auxiliary array
00124     REAL *b_tmp = NULL;
00125
00126     // this should be done once and for all!!
00127     b_tmp = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00128
00129     // b_tmp = b_val
00130     fasp_darray_cp(size, b_val, b_tmp);
00131
00132     // It's not necessary to assign the smoothing order since the results doesn't depend on it
00133     if (nc == 1) {
00134         for (point = 0; point < ngrid; point++) {
00135             for (band = 0; band < nband; band++) {
00136                 width = offsets[band];
00137                 column = point + width;
00138                 if (width < 0) {
00139                     if (column >= 0) {
00140                         b_tmp[point] -= offdiag[band][column]*u_val[column];
00141                     }
00142                 }
00143                 else { // width > 0
00144                     if (column < ngrid) {
00145                         b_tmp[point] -= offdiag[band][point]*u_val[column];
00146                     }
00147                 }
00148             } // end for band
00149         } // end for point
00150
00151         for (point = 0; point < ngrid; point++) {
00152             // zero-diagonal should be tested previously
00153             u_val[point] = b_tmp[point] / diag[point];
00154         }
00155     } // end if (nc == 1)
00156     else if (nc > 1) {
00157         for (block = 0; block < ngrid; block++) {
00158             start_DATA = nc2*block;
00159             start_vecb = nc*block;
00160             for (band = 0; band < nband; band++) {
00161                 width = offsets[band];
00162                 column = block + width;
00163                 if (width < 0) {
00164                     if (column >= 0) {

```



```

00165         start_data = nc2*column;
00166         start_vecu = nc*column;
00167         blkcontr2( start_data, start_vecu, start_vecb,
00168                 nc, offdiag[band], u_val, b_tmp );
00169     }
00170 }
00171     else { // width > 0
00172         if (column < ngrid) {
00173             start_vecu = nc*column;
00174             blkcontr2( start_DATA, start_vecu, start_vecb,
00175                     nc, offdiag[band], u_val, b_tmp );
00176         }
00177     }
00178 } // end for band
00179 } // end for block
00180
00181     for (block = 0; block < ngrid; block++) {
00182         start = nc*block;
00183         fasp_blas_smat_mxv(diaginv+nc2*block, b_tmp+start, u_val+start, nc);
00184     }
00185 } // end else if (nc > 1)
00186 else {
00187     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
00188     return;
00189 }
00190
00191 fasp_mem_free(b_tmp); b_tmp = NULL;
00192 }
00193
00217 void fasp_smoother_dstr_gs (dSTRmat      *A,
00218                             dvector      *b,
00219                             dvector      *u,
00220                             const INT    order,
00221                             INT          *mark)
00222 {
00223     INT    nc      = A->nc;    // size of each block (number of components)
00224     INT    ngrid   = A->ngrid;  // number of grids
00225     REAL *diag     = A->diag;   // Diagonal entries
00226     REAL *diaginv  = NULL;     // Diagonal inverse(when nc>1), same size and storage scheme as A->diag
00227
00228     INT nc2      = nc*nc;
00229     INT size     = nc2*ngrid;
00230     INT block    = 0;
00231     INT start    = 0;
00232
00233     if (nc > 1) {
00234         // allocate memory
00235         diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00236
00237         // diaginv = diag;
00238         fasp_darray_cp(size, diag, diaginv);
00239
00240         // generate diaginv
00241         for (block = 0; block < ngrid; block++) {
00242             fasp_smat_inv(diaginv+start, nc);
00243             start += nc2;
00244         }
00245     }
00246
00247     fasp_smoother_dstr_gs1(A, b, u, order, mark, diaginv);
00248
00249     fasp_mem_free(diaginv); diaginv = NULL;
00250 }
00251
00277 void fasp_smoother_dstr_gs1 (dSTRmat      *A,
00278                             dvector      *b,
00279                             dvector      *u,
00280                             const INT    order,
00281                             INT          *mark,
00282                             REAL        *diaginv)
00283 {
00284
00285     if (!mark) {
00286         if (order == ASCEND) // smooth ascendingly
00287         {
00288             fasp_smoother_dstr_gs_ascend(A, b, u, diaginv);
00289         }
00290         else if (order == DESCEND) // smooth descendingly
00291         {
00292             fasp_smoother_dstr_gs_descend(A, b, u, diaginv);
00293         }
00294     }

```

```

00294     }
00295     else {
00296         if (order == USERDEFINED) // smooth according to the order 'mark' defined by user
00297         {
00298             fasp_smoother_dstr_gs_order(A, b, u, diaginv, mark);
00299         }
00300         else // smooth according to 'mark', where 'mark' is a CF_marker array
00301         {
00302             fasp_smoother_dstr_gs_cf(A, b, u, diaginv, mark, order);
00303         }
00304     }
00305 }
00306
00322 void fasp_smoother_dstr_gs_ascend (dSTRmat *A,
00323                                   dvector *b,
00324                                   dvector *u,
00325                                   REAL *diaginv)
00326 {
00327     // information of A
00328     INT ngrid = A->ngrid; // number of grids
00329     INT nc = A->nc; // size of each block (number of components)
00330     INT nband = A->nband; // number of off-diag band
00331     INT *offsets = A->offsets; // offsets of the off-diagonals
00332     REAL *diag = A->diag; // Diagonal entries
00333     REAL **offdiag = A->offdiag; // Off-diagonal entries
00334
00335     // values of dvector b and u
00336     REAL *b_val = b->val;
00337     REAL *u_val = u->val;
00338
00339     // local variables
00340     INT block = 0;
00341     INT point = 0;
00342     INT band = 0;
00343     INT width = 0;
00344     INT nc2 = nc*nc;
00345     INT ncb = 0;
00346     INT column = 0;
00347     INT start_data = 0;
00348     INT start_DATA = 0;
00349     INT start_vecu = 0;
00350     REAL rhs = 0.0;
00351
00352     // auxiliary array(nc*1 vector)
00353     REAL *vec_tmp = NULL;
00354
00355     vec_tmp = (REAL *)fasp_mem_calloc(nc, sizeof(REAL));
00356
00357     if (nc == 1) {
00358         for (point = 0; point < ngrid; point++) {
00359             rhs = b_val[point];
00360             for (band = 0; band < nband; band++) {
00361                 width = offsets[band];
00362                 column = point + width;
00363                 if (width < 0) {
00364                     if (column >= 0) {
00365                         rhs -= offdiag[band][column]*u_val[column];
00366                     }
00367                 }
00368                 else { // width > 0
00369                     if (column < ngrid) {
00370                         rhs -= offdiag[band][point]*u_val[column];
00371                     }
00372                 }
00373             } // end for band
00374
00375             // zero-diagonal should be tested previously
00376             u_val[point] = rhs / diag[point];
00377         } // end for point
00378     } // end if (nc == 1)
00379
00380     // end if (nc == 1)
00381
00382     else if (nc > 1) {
00383         for (block = 0; block < ngrid; block++) {
00384             ncb = nc*block;
00385             for (point = 0; point < nc; point++) {
00386                 vec_tmp[point] = b_val[ncb+point];
00387             }
00388             start_DATA = nc2*block;
00389             for (band = 0; band < nband; band++) {

```

```

00390         width = offsets[band];
00391         column = block + width;
00392         if (width < 0) {
00393             if (column >= 0) {
00394                 start_data = nc2*column;
00395                 start_vecu = nc*column;
00396                 blkcontr2( start_data, start_vecu, 0, nc,
00397                     offdiag[band], u_val, vec_tmp );
00398             }
00399         }
00400         else { // width > 0
00401             if (column < ngrid) {
00402                 start_vecu = nc*column;
00403                 blkcontr2( start_DATA, start_vecu, 0, nc,
00404                     offdiag[band], u_val, vec_tmp );
00405             }
00406         }
00407     } // end for band
00408
00409     // subblock smoothing
00410     fasp_blas_smat_mnv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
00411 } // end for block
00412
00413 } // end else if (nc > 1)
00414 else {
00415     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
00416     return;
00417 }
00418
00419 fasp_mem_free(vec_tmp); vec_tmp = NULL;
00420 }
00421
00422 void fasp_smoother_dstr_gs_descend (dSTRmat *A,
00423     dvector *b,
00424     dvector *u,
00425     REAL *diaginv)
00426 {
00427     // information of A
00428     INT ngrid = A->ngrid; // number of grids
00429     INT nc = A->nc; // size of each block (number of components)
00430     INT nband = A->nband; // number of off-diag band
00431     INT *offsets = A->offsets; // offsets of the off-diagals
00432     REAL *diag = A->diag; // Diagonal entries
00433     REAL **offdiag = A->offdiag; // Off-diagonal entries
00434
00435     // values of dvector b and u
00436     REAL *b_val = b->val;
00437     REAL *u_val = u->val;
00438
00439     // local variables
00440     INT block = 0;
00441     INT point = 0;
00442     INT band = 0;
00443     INT width = 0;
00444     INT nc2 = nc*nc;
00445     INT ncb = 0;
00446     INT column = 0;
00447     INT start_data = 0;
00448     INT start_DATA = 0;
00449     INT start_vecu = 0;
00450     REAL rhs = 0.0;
00451
00452     // auxiliary array(nc*1 vector)
00453     REAL *vec_tmp = NULL;
00454
00455     vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));
00456
00457     if (nc == 1) {
00458         for (point = ngrid-1; point >= 0; point --) {
00459             rhs = b_val[point];
00460             for (band = 0; band < nband; band++) {
00461                 width = offsets[band];
00462                 column = point + width;
00463                 if (width < 0) {
00464                     if (column >= 0) {
00465                         rhs -= offdiag[band][column]*u_val[column];
00466                     }
00467                 }
00468                 else { // width > 0
00469                     if (column < ngrid) {

```

```

00486             rhs -= offdiag[band][point]*u_val[column];
00487         }
00488     }
00489 } // end for band
00490
00491 // zero-diagonal should be tested previously
00492 u_val[point] = rhs / diag[point];
00493
00494 } // end for point
00495
00496 } // end if (nc == 1)
00497
00498 else if (nc > 1) {
00499     for (block = ngrid-1; block >= 0; block --) {
00500         ncb = nc*block;
00501         for (point = 0; point < nc; point ++) {
00502             vec_tmp[point] = b_val[ncb+point];
00503         }
00504         start_DATA = nc2*block;
00505         for (band = 0; band < nband; band ++) {
00506             width = offsets[band];
00507             column = block + width;
00508             if (width < 0) {
00509                 if (column >= 0) {
00510                     start_data = nc2*column;
00511                     start_vecu = nc*column;
00512                     blkcontr2( start_data, start_vecu, 0, nc,
00513                             offdiag[band], u_val, vec_tmp );
00514                 }
00515             }
00516             else { // width > 0
00517                 if (column < ngrid) {
00518                     start_vecu = nc*column;
00519                     blkcontr2( start_DATA, start_vecu, 0, nc,
00520                             offdiag[band], u_val, vec_tmp );
00521                 }
00522             }
00523         } // end for band
00524
00525         // subblock smoothing
00526         fasp_blas_smat_mnv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
00527
00528     } // end for block
00529
00530 } // end else if (nc > 1)
00531
00532 else {
00533     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
00534     return;
00535 }
00536
00537 fasp_mem_free(vec_tmp); vec_tmp = NULL;
00538 }
00539
00556 void fasp_smoother_dstr_gs_order (dSTRmat *A,
00557                                   dvector *b,
00558                                   dvector *u,
00559                                   REAL *diaginv,
00560                                   INT *mark)
00561 {
00562     // information of A
00563     INT ngrid = A->ngrid; // number of grids
00564     INT nc = A->nc; // size of each block (number of components)
00565     INT nband = A->nband; // number of off-diag band
00566     INT *offsets = A->offsets; // offsets of the off-diagals
00567     REAL *diag = A->diag; // Diagonal entries
00568     REAL **offdiag = A->offdiag; // Off-diagonal entries
00569
00570     // values of dvector b and u
00571     REAL *b_val = b->val;
00572     REAL *u_val = u->val;
00573
00574     // local variables
00575     INT block = 0;
00576     INT point = 0;
00577     INT band = 0;
00578     INT width = 0;
00579     INT nc2 = nc*nc;
00580     INT ncb = 0;
00581     INT index = 0;
00582     INT column = 0;

```

```

00583     INT start_data = 0;
00584     INT start_DATA = 0;
00585     INT start_vecu = 0;
00586     REAL rhs = 0.0;
00587
00588     // auxiliary array(nc+1 vector)
00589     REAL *vec_tmp = NULL;
00590
00591     vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));
00592
00593     if (nc == 1) {
00594         for (index = 0; index < ngrid; index++) {
00595             point = mark[index];
00596             rhs = b_val[point];
00597             for (band = 0; band < nband; band++) {
00598                 width = offsets[band];
00599                 column = point + width;
00600                 if (width < 0) {
00601                     if (column >= 0) {
00602                         rhs -= offdiag[band][column]*u_val[column];
00603                     }
00604                 }
00605                 else { // width > 0
00606                     if (column < ngrid) {
00607                         rhs -= offdiag[band][point]*u_val[column];
00608                     }
00609                 }
00610             } // end for band
00611
00612             // zero-diagonal should be tested previously
00613             u_val[point] = rhs / diag[point];
00614
00615         } // end for index
00616     } // end if (nc == 1)
00617
00618     else if (nc > 1) {
00619         for (index = 0; index < ngrid; index++) {
00620             block = mark[index];
00621             ncb = nc*block;
00622             for (point = 0; point < ncb; point++) {
00623                 vec_tmp[point] = b_val[ncb+point];
00624             }
00625             start_DATA = nc2*block;
00626             for (band = 0; band < nband; band++) {
00627                 width = offsets[band];
00628                 column = block + width;
00629                 if (width < 0) {
00630                     if (column >= 0) {
00631                         start_data = nc2*column;
00632                         start_vecu = nc*column;
00633                         blkcontr2( start_data, start_vecu, 0, nc,
00634                             offdiag[band], u_val, vec_tmp );
00635                     }
00636                 }
00637                 else { // width > 0
00638                     if (column < ngrid) {
00639                         start_vecu = nc*column;
00640                         blkcontr2( start_DATA, start_vecu, 0, nc,
00641                             offdiag[band], u_val, vec_tmp );
00642                     }
00643                 }
00644             } // end for band
00645
00646             // subblock smoothing
00647             fasp_blas_smat_mnv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
00648
00649         } // end for index
00650     } // end else if (nc > 1)
00651
00652     else {
00653         printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
00654         return;
00655     }
00656
00657     fasp_mem_free(vec_tmp); vec_tmp = NULL;
00658 }
00659
00660 void fasp_smoother_dstr_gs_cf (dSTRmat *A,
00680     dvector *b,
00681     dvector *u,

```

```

00683                                     REAL      *diaginv,
00684                                     INT        *mark,
00685                                     const INT  order)
00686 {
00687     // information of A
00688     INT ngrid = A->ngrid; // number of grids
00689     INT nc = A->nc; // size of each block (number of components)
00690     INT nband = A->nband; // number of off-diag band
00691     INT *offsets = A->offsets; // offsets of the off-diagals
00692     REAL *diag = A->diag; // Diagonal entries
00693     REAL **offdiag = A->offdiag; // Off-diagonal entries
00694
00695     // values of dvector b and u
00696     REAL *b_val = b->val;
00697     REAL *u_val = u->val;
00698
00699     // local variables
00700     INT block = 0;
00701     INT point = 0;
00702     INT band = 0;
00703     INT width = 0;
00704     INT nc2 = nc*nc;
00705     INT ncb = 0;
00706     INT column = 0;
00707     INT start_data = 0;
00708     INT start_DATA = 0;
00709     INT start_vecu = 0;
00710     INT FIRST = order; // which kind of points to be smoothed firstly?
00711     INT SECOND = -order; // which kind of points to be smoothed secondly?
00712
00713     REAL rhs = 0.0;
00714
00715     // auxiliary array(nc*1 vector)
00716     REAL *vec_tmp = NULL;
00717
00718     vec_tmp = (REAL *)fasp_mem_calloc(nc, sizeof(REAL));
00719
00720     if (nc == 1) {
00721         // deal with the points marked FIRST
00722         for (point = 0; point < ngrid; point++) {
00723             if (mark[point] == FIRST) {
00724                 rhs = b_val[point];
00725                 for (band = 0; band < nband; band++) {
00726                     width = offsets[band];
00727                     column = point + width;
00728                     if (width < 0) {
00729                         if (column >= 0) {
00730                             rhs -= offdiag[band][column]*u_val[column];
00731                         }
00732                     }
00733                     else { // width > 0
00734                         if (column < ngrid) {
00735                             rhs -= offdiag[band][point]*u_val[column];
00736                         }
00737                     }
00738                 } // end for band
00739
00740                 // zero-diagonal should be tested previously
00741                 u_val[point] = rhs / diag[point];
00742             } // end if (mark[point] == FIRST)
00743         } // end for point
00744
00745         // deal with the points marked SECOND
00746         for (point = 0; point < ngrid; point++) {
00747             if (mark[point] == SECOND) {
00748                 rhs = b_val[point];
00749                 for (band = 0; band < nband; band++) {
00750                     width = offsets[band];
00751                     column = point + width;
00752                     if (width < 0) {
00753                         if (column >= 0) {
00754                             rhs -= offdiag[band][column]*u_val[column];
00755                         }
00756                     }
00757                     else { // width > 0
00758                         if (column < ngrid) {
00759                             rhs -= offdiag[band][point]*u_val[column];
00760                         }
00761                     }
00762                 } // end for band
00763             }

```

```

00764         // zero-diagonal should be tested previously
00765         u_val[point] = rhs / diag[point];
00766     } // end if (mark[point] == SECOND)
00767 } // end for point
00768
00769 } // end if (nc == 1)
00770
00771 else if (nc > 1) {
00772     // deal with the blocks marked FIRST
00773     for (block = 0; block < ngrid; block++) {
00774         if (mark[block] == FIRST) {
00775             ncb = nc*block;
00776             for (point = 0; point < nc; point++) {
00777                 vec_tmp[point] = b_val[ncb+point];
00778             }
00779             start_DATA = nc2*block;
00780             for (band = 0; band < nband; band++) {
00781                 width = offsets[band];
00782                 column = block + width;
00783                 if (width < 0) {
00784                     if (column >= 0) {
00785                         start_data = nc2*column;
00786                         start_vecu = nc*column;
00787                         blkcontr2( start_data, start_vecu, 0, nc,
00788                             offdiag[band], u_val, vec_tmp );
00789                     }
00790                 }
00791                 else { // width > 0
00792                     if (column < ngrid) {
00793                         start_vecu = nc*column;
00794                         blkcontr2( start_DATA, start_vecu, 0, nc,
00795                             offdiag[band], u_val, vec_tmp );
00796                     }
00797                 }
00798             } // end for band
00799
00800             // subblock smoothing
00801             fasp_blas_smat_mnv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
00802         } // end if (mark[block] == FIRST)
00803     } // end for block
00804
00805     // deal with the blocks marked SECOND
00806     for (block = 0; block < ngrid; block++) {
00807         if (mark[block] == SECOND) {
00808             ncb = nc*block;
00809             for (point = 0; point < nc; point++) {
00810                 vec_tmp[point] = b_val[ncb+point];
00811             }
00812             start_DATA = nc2*block;
00813             for (band = 0; band < nband; band++) {
00814                 width = offsets[band];
00815                 column = block + width;
00816                 if (width < 0) {
00817                     if (column >= 0) {
00818                         start_data = nc2*column;
00819                         start_vecu = nc*column;
00820                         blkcontr2( start_data, start_vecu, 0, nc,
00821                             offdiag[band], u_val, vec_tmp );
00822                     }
00823                 }
00824                 else { // width > 0
00825                     if (column < ngrid) {
00826                         start_vecu = nc*column;
00827                         blkcontr2( start_DATA, start_vecu, 0, nc,
00828                             offdiag[band], u_val, vec_tmp );
00829                     }
00830                 }
00831             } // end for band
00832
00833             // subblock smoothing
00834             fasp_blas_smat_mnv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
00835         } // end if (mark[block] == SECOND)
00836     } // end for block
00837
00838 } // end else if (nc > 1)
00839
00840 } // end else if (nc > 1)
00841 else {
00842     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
00843     return;
00844 }

```

```

00845
00846     fasp_mem_free(vec_tmp); vec_tmp = NULL;
00847 }
00848
00873 void fasp_smoother_dstr_sor (dSTRmat    *A,
00874                             dvector    *b,
00875                             dvector    *u,
00876                             const INT  order,
00877                             INT        *mark,
00878                             const REAL weight)
00879 {
00880     INT    nc    = A->nc;    // size of each block (number of components)
00881     INT    ngrid = A->ngrid; // number of grids
00882     REAL   *diag = A->diag;   // Diagonal entries
00883     REAL *diaginv = NULL;    // Diagonal inverse(when nc>1),same size and storage scheme as A->diag
00884
00885     INT nc2    = nc*nc;
00886     INT size   = nc2*ngrid;
00887     INT block  = 0;
00888     INT start  = 0;
00889
00890     if (nc > 1) {
00891         // allocate memory
00892         diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
00893
00894         // diaginv = diag;
00895         fasp_darray_cp(size, diag, diaginv);
00896
00897         // generate diaginv
00898         for (block = 0; block < ngrid; block++) {
00899             fasp_smat_inv(diaginv+start, nc);
00900             start += nc2;
00901         }
00902     }
00903
00904     fasp_smoother_dstr_sor1(A, b, u, order, mark, diaginv, weight);
00905
00906     fasp_mem_free(diaginv); diaginv = NULL;
00907 }
00908
00935 void fasp_smoother_dstr_sor1 (dSTRmat    *A,
00936                              dvector    *b,
00937                              dvector    *u,
00938                              const INT  order,
00939                              INT        *mark,
00940                              REAL       *diaginv,
00941                              const REAL weight)
00942 {
00943     if (!mark) {
00944         if (order == ASCEND) // smooth ascendingly
00945         {
00946             fasp_smoother_dstr_sor_ascend(A, b, u, diaginv, weight);
00947         }
00948         else if (order == DESCEND) // smooth descendingly
00949         {
00950             fasp_smoother_dstr_sor_descend(A, b, u, diaginv, weight);
00951         }
00952     }
00953     else {
00954         if (order == USERDEFINED) // smooth according to the order 'mark' defined by user
00955         {
00956             fasp_smoother_dstr_sor_order(A, b, u, diaginv, mark, weight);
00957         }
00958         else // smooth according to 'mark', where 'mark' is a CF_marker array
00959         {
00960             fasp_smoother_dstr_sor_cf(A, b, u, diaginv, mark, order, weight);
00961         }
00962     }
00963 }
00964
00981 void fasp_smoother_dstr_sor_ascend (dSTRmat *A,
00982                                    dvector *b,
00983                                    dvector *u,
00984                                    REAL     *diaginv,
00985                                    REAL     weight)
00986 {
00987     // information of A
00988     INT ngrid = A->ngrid; // number of grids
00989     INT nc    = A->nc;    // size of each block (number of components)
00990     INT nband = A->nband; // number of off-diag band
00991     INT *offsets = A->offsets; // offsets of the off-diagals

```



```

00992     REAL *diag = A->diag;          // Diagonal entries
00993     REAL **offdiag = A->offdiag;    // Off-diagonal entries
00994
00995     // values of dvector b and u
00996     REAL *b_val = b->val;
00997     REAL *u_val = u->val;
00998
00999     // local variables
01000     INT block = 0;
01001     INT point = 0;
01002     INT band = 0;
01003     INT width = 0;
01004     INT nc2 = nc*nc;
01005     INT ncb = 0;
01006     INT column = 0;
01007     INT start_data = 0;
01008     INT start_DATA = 0;
01009     INT start_vecu = 0;
01010     REAL rhs = 0.0;
01011     REAL one_minus_weight = 1.0 - weight;
01012
01013     // auxiliary array(nc*1 vector)
01014     REAL *vec_tmp = NULL;
01015
01016     vec_tmp = (REAL *)fasp_mem_calloc(nc, sizeof(REAL));
01017
01018     if (nc == 1) {
01019         for (point = 0; point < ngrid; point++) {
01020             rhs = b_val[point];
01021             for (band = 0; band < nband; band++) {
01022                 width = offsets[band];
01023                 column = point + width;
01024                 if (width < 0) {
01025                     if (column >= 0) {
01026                         rhs -= offdiag[band][column]*u_val[column];
01027                     }
01028                 }
01029                 else { // width > 0
01030                     if (column < ngrid) {
01031                         rhs -= offdiag[band][point]*u_val[column];
01032                     }
01033                 }
01034             } // end for band
01035
01036             // zero-diagonal should be tested previously
01037             u_val[point] = one_minus_weight*u_val[point] +
01038                 weight*(rhs / diag[point]);
01039
01040         } // end for point
01041     } // end if (nc == 1)
01042
01043     else if (nc > 1) {
01044         for (block = 0; block < ngrid; block++) {
01045             ncb = nc*block;
01046             for (point = 0; point < nc; point++) {
01047                 vec_tmp[point] = b_val[ncb+point];
01048             }
01049             start_DATA = nc2*block;
01050             for (band = 0; band < nband; band++) {
01051                 width = offsets[band];
01052                 column = block + width;
01053                 if (width < 0) {
01054                     if (column >= 0) {
01055                         start_data = nc2*column;
01056                         start_vecu = nc*column;
01057                         blkcontr2( start_data, start_vecu, 0, nc,
01058                             offdiag[band], u_val, vec_tmp );
01059                     }
01060                 }
01061                 else { // width > 0
01062                     if (column < ngrid) {
01063                         start_vecu = nc*column;
01064                         blkcontr2( start_DATA, start_vecu, 0, nc,
01065                             offdiag[band], u_val, vec_tmp );
01066                     }
01067                 }
01068             } // end for band
01069         } // end for block
01070
01071         // subblock smoothing
01072         aAxpby(weight, one_minus_weight, nc,

```

```

01073         diaginv+start_DATA, vec_tmp, u_val+nc*block);
01074     } // end for block
01075 } // end else if (nc > 1)
01076 else {
01077     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
01078     return;
01079 }
01080 fasp_mem_free(vec_tmp); vec_tmp = NULL;
01081 }
01082
01083 void fasp_smoother_dstr_sor_descend (dSTRmat *A,
01084                                     dvector *b,
01085                                     dvector *u,
01086                                     REAL *diaginv,
01087                                     REAL weight)
01088 {
01089     // information of A
01090     INT ngrid = A->ngrid; // number of grids
01091     INT nc = A->nc; // size of each block (number of components)
01092     INT nband = A->nband; // number of off-diag band
01093     INT *offsets = A->offsets; // offsets of the off-diagonals
01094     REAL *diag = A->diag; // Diagonal entries
01095     REAL **offdiag = A->offdiag; // Off-diagonal entries
01096
01097     // values of dvector b and u
01098     REAL *b_val = b->val;
01099     REAL *u_val = u->val;
01100
01101     // local variables
01102     INT block = 0;
01103     INT point = 0;
01104     INT band = 0;
01105     INT width = 0;
01106     INT nc2 = nc*nc;
01107     INT ncb = 0;
01108     INT column = 0;
01109     INT start_data = 0;
01110     INT start_DATA = 0;
01111     INT start_vecu = 0;
01112     REAL rhs = 0.0;
01113     REAL one_minus_weight = 1.0 - weight;
01114
01115     // auxiliary array(nc*1 vector)
01116     REAL *vec_tmp = NULL;
01117
01118     vec_tmp = (REAL *)fasp_mem_calloc(nc, sizeof(REAL));
01119
01120     if (nc == 1) {
01121         for (point = ngrid-1; point >= 0; point --) {
01122             rhs = b_val[point];
01123             for (band = 0; band < nband; band++) {
01124                 width = offsets[band];
01125                 column = point + width;
01126                 if (width < 0) {
01127                     if (column >= 0) {
01128                         rhs -= offdiag[band][column]*u_val[column];
01129                     }
01130                 }
01131                 else { // width > 0
01132                     if (column < ngrid) {
01133                         rhs -= offdiag[band][point]*u_val[column];
01134                     }
01135                 }
01136             } // end for band
01137
01138             // zero-diagonal should be tested previously
01139             u_val[point] = one_minus_weight*u_val[point] +
01140                 weight*(rhs / diag[point]);
01141         } // end for point
01142     } // end if (nc == 1)
01143
01144     else if (nc > 1) {
01145         for (block = ngrid-1; block >= 0; block --) {
01146             ncb = nc*block;
01147             for (point = 0; point < nc; point++) {
01148                 vec_tmp[point] = b_val[ncb+point];
01149             }
01150         }
01151     }
01152 }

```

```

01170     }
01171     start_DATA = nc2*block;
01172     for (band = 0; band < nband; band++) {
01173         width = offsets[band];
01174         column = block + width;
01175         if (width < 0) {
01176             if (column >= 0) {
01177                 start_data = nc2*column;
01178                 start_vecu = nc*column;
01179                 blkcontr2( start_data, start_vecu, 0, nc,
01180                     offdiag[band], u_val, vec_tmp );
01181             }
01182         }
01183         else { // width > 0
01184             if (column < ngrid) {
01185                 start_vecu = nc*column;
01186                 blkcontr2( start_DATA, start_vecu, 0, nc,
01187                     offdiag[band], u_val, vec_tmp );
01188             }
01189         }
01190     } // end for band
01191
01192     // subblock smoothing
01193     aAxpby(weight, one_minus_weight, nc,
01194         diaginv+start_DATA, vec_tmp, u_val+nc*block);
01195
01196 } // end for block
01197
01198 } // end else if (nc > 1)
01199 else {
01200     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
01201     return;
01202 }
01203
01204 fasp_mem_free(vec_tmp); vec_tmp = NULL;
01205 }
01206
01224 void fasp_smoother_dstr_sor_order (dSTRmat *A,
01225     dvector *b,
01226     dvector *u,
01227     REAL *diaginv,
01228     INT *mark,
01229     REAL weight)
01230 {
01231     // information of A
01232     INT ngrid = A->ngrid; // number of grids
01233     INT nc = A->nc; // size of each block (number of components)
01234     INT nband = A->nband; // number of off-diag band
01235     INT *offsets = A->offsets; // offsets of the off-diagals
01236     REAL *diag = A->diag; // Diagonal entries
01237     REAL **offdiag = A->offdiag; // Off-diagonal entries
01238
01239     // values of dvector b and u
01240     REAL *b_val = b->val;
01241     REAL *u_val = u->val;
01242
01243     // local variables
01244     INT block = 0;
01245     INT point = 0;
01246     INT band = 0;
01247     INT width = 0;
01248     INT nc2 = nc*nc;
01249     INT ncb = 0;
01250     INT column = 0;
01251     INT index = 0;
01252     INT start_data = 0;
01253     INT start_DATA = 0;
01254     INT start_vecu = 0;
01255     REAL rhs = 0.0;
01256     REAL one_minus_weight = 1.0 - weight;
01257
01258     // auxiliary array(nc+1 vector)
01259     REAL *vec_tmp = NULL;
01260
01261     vec_tmp = (REAL *)fasp_mem_calloc(nc, sizeof(REAL));
01262
01263     if (nc == 1) {
01264         for (index = 0; index < ngrid; index++) {
01265             point = mark[index];
01266             rhs = b_val[point];
01267             for (band = 0; band < nband; band++) {

```

```

01268         width = offsets[band];
01269         column = point + width;
01270         if (width < 0) {
01271             if (column >= 0) {
01272                 rhs -= offdiag[band][column]*u_val[column];
01273             }
01274         }
01275         else { // width > 0
01276             if (column < ngrid) {
01277                 rhs -= offdiag[band][point]*u_val[column];
01278             }
01279         }
01280     } // end for band
01281
01282     // zero-diagonal should be tested previously
01283     u_val[point] = one_minus_weight*u_val[point] +
01284         weight*(rhs / diag[point]);
01285
01286 } // end for index
01287
01288 } // end if (nc == 1)
01289
01290 else if (nc > 1) {
01291     for (index = 0; index < ngrid; index++) {
01292         block = mark[index];
01293         ncb = nc*block;
01294         for (point = 0; point < nc; point++) {
01295             vec_tmp[point] = b_val[ncb+point];
01296         }
01297         start_DATA = nc2*block;
01298         for (band = 0; band < nband; band++) {
01299             width = offsets[band];
01300             column = block + width;
01301             if (width < 0) {
01302                 if (column >= 0) {
01303                     start_data = nc2*column;
01304                     start_vecu = nc*column;
01305                     blkcontr2( start_data, start_vecu, 0, nc,
01306                             offdiag[band], u_val, vec_tmp );
01307                 }
01308             }
01309             else { // width > 0
01310                 if (column < ngrid) {
01311                     start_vecu = nc*column;
01312                     blkcontr2( start_DATA, start_vecu, 0, nc,
01313                             offdiag[band], u_val, vec_tmp );
01314                 }
01315             }
01316         } // end for band
01317
01318         // subblock smoothing
01319         aAxpby(weight, one_minus_weight, nc,
01320             diaginv+start_DATA, vec_tmp, u_val+nc*block);
01321
01322     } // end for index
01323
01324 } // end else if (nc > 1)
01325
01326 else {
01327     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
01328     return;
01329 }
01330
01331 fasp_mem_free(vec_tmp); vec_tmp = NULL;
01332 }
01333
01355 void fasp_smoother_dstr_sor_cf (dSTRmat *A,
01356                                dvector *b,
01357                                dvector *u,
01358                                REAL *diaginv,
01359                                INT *mark,
01360                                const INT order,
01361                                const REAL weight)
01362 {
01363     // information of A
01364     INT ngrid = A->ngrid; // number of grids
01365     INT nc = A->nc; // size of each block (number of components)
01366     INT nband = A->nband; // number of off-diag band
01367     INT *offsets = A->offsets; // offsets of the off-diagals
01368     REAL *diag = A->diag; // Diagonal entries
01369     REAL **offdiag = A->offdiag; // Off-diagonal entries

```

```

01370
01371 // values of dvector b and u
01372 REAL *b_val = b->val;
01373 REAL *u_val = u->val;
01374
01375 // local variables
01376 INT block = 0;
01377 INT point = 0;
01378 INT band = 0;
01379 INT width = 0;
01380 INT nc2 = nc*nc;
01381 INT ncb = 0;
01382 INT column = 0;
01383 INT start_data = 0;
01384 INT start_DATA = 0;
01385 INT start_vecu = 0;
01386 REAL rhs = 0.0;
01387 REAL one_minus_weight = 1.0 - weight;
01388 INT FIRST = order; // which kind of points to be smoothed firstly?
01389 INT SECOND = -order; // which kind of points to be smoothed secondly?
01390
01391 // auxiliary array(nc*1 vector)
01392 REAL *vec_tmp = NULL;
01393
01394 vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));
01395
01396 if (nc == 1) {
01397     // deal with the points marked FIRST
01398     for (point = 0; point < ngrid; point++) {
01399         if (mark[point] == FIRST) {
01400             rhs = b_val[point];
01401             for (band = 0; band < nband; band++) {
01402                 width = offsets[band];
01403                 column = point + width;
01404                 if (width < 0) {
01405                     if (column >= 0) {
01406                         rhs -= offdiag[band][column]*u_val[column];
01407                     }
01408                 }
01409                 else { // width > 0
01410                     if (column < ngrid) {
01411                         rhs -= offdiag[band][point]*u_val[column];
01412                     }
01413                 }
01414             } // end for band
01415
01416             // zero-diagonal should be tested previously
01417             u_val[point] = one_minus_weight*u_val[point] +
01418                 weight*(rhs / diag[point]);
01419
01420         } // end if (mark[point] == FIRST)
01421     } // end for point
01422
01423     // deal with the points marked SECOND
01424     for (point = 0; point < ngrid; point++) {
01425         if (mark[point] == SECOND) {
01426             rhs = b_val[point];
01427             for (band = 0; band < nband; band++) {
01428                 width = offsets[band];
01429                 column = point + width;
01430                 if (width < 0) {
01431                     if (column >= 0) {
01432                         rhs -= offdiag[band][column]*u_val[column];
01433                     }
01434                 }
01435                 else { // width > 0
01436                     if (column < ngrid) {
01437                         rhs -= offdiag[band][point]*u_val[column];
01438                     }
01439                 }
01440             } // end for band
01441
01442             // zero-diagonal should be tested previously
01443             u_val[point] = rhs / diag[point];
01444         } // end if (mark[point] == SECOND)
01445     } // end for point
01446
01447 } // end if (nc == 1)
01448
01449 else if (nc > 1) {
01450     // deal with the blocks marked FIRST

```

```

01451     for (block = 0; block < ngrid; block++) {
01452         if (mark[block] == FIRST) {
01453             ncb = nc*block;
01454             for (point = 0; point < nc; point++) {
01455                 vec_tmp[point] = b_val[ncb+point];
01456             }
01457             start_DATA = nc2*block;
01458             for (band = 0; band < nband; band++) {
01459                 width = offsets[band];
01460                 column = block + width;
01461                 if (width < 0) {
01462                     if (column >= 0) {
01463                         start_data = nc2*column;
01464                         start_vecu = nc*column;
01465                         blkcontr2( start_data, start_vecu, 0, nc,
01466                                 offdiag[band], u_val, vec_tmp );
01467                     }
01468                 }
01469                 else { // width > 0
01470                     if (column < ngrid) {
01471                         start_vecu = nc*column;
01472                         blkcontr2( start_DATA, start_vecu, 0, nc,
01473                                 offdiag[band], u_val, vec_tmp );
01474                     }
01475                 }
01476             } // end for band
01477
01478             // subblock smoothing
01479             aAxpby(weight, one_minus_weight, nc,
01480                   diaginv+start_DATA, vec_tmp, u_val+nc*block);
01481         } // end if (mark[block] == FIRST)
01482
01483     } // end for block
01484
01485     // deal with the blocks marked SECOND
01486     for (block = 0; block < ngrid; block++) {
01487         if (mark[block] == SECOND) {
01488             ncb = nc*block;
01489             for (point = 0; point < nc; point++) {
01490                 vec_tmp[point] = b_val[ncb+point];
01491             }
01492             start_DATA = nc2*block;
01493             for (band = 0; band < nband; band++) {
01494                 width = offsets[band];
01495                 column = block + width;
01496                 if (width < 0) {
01497                     if (column >= 0) {
01498                         start_data = nc2*column;
01499                         start_vecu = nc*column;
01500                         blkcontr2( start_data, start_vecu, 0, nc,
01501                                 offdiag[band], u_val, vec_tmp );
01502                     }
01503                 }
01504                 else { // width > 0
01505                     if (column < ngrid) {
01506                         start_vecu = nc*column;
01507                         blkcontr2( start_DATA, start_vecu, 0, nc,
01508                                 offdiag[band], u_val, vec_tmp );
01509                     }
01510                 }
01511             } // end for band
01512
01513             // subblock smoothing
01514             aAxpby(weight, one_minus_weight, nc,
01515                   diaginv+start_DATA, vec_tmp, u_val+nc*block);
01516         } // end if (mark[block] == SECOND)
01517
01518     } // end for block
01519
01520 } // end else if (nc > 1)
01521 else {
01522     printf("### ERROR: nc is illegal! [%s:%d]\n", __FILE__, __LINE__);
01523     return;
01524 }
01525
01526 fasp_mem_free(vec_tmp); vec_tmp = NULL;
01527 }
01528
01543 void fasp_generate_diaginv_block (dSTRmat *A,
01544                                  ivector *neigh,
01545                                  dvector *diaginv,

```

```

01546                                     ivector *pivot)
01547 {
01548     // information about A
01549     const INT nc = A->nc;
01550     const INT ngrid = A->ngrid;
01551     const INT nband = A->nband;
01552
01553     INT *offsets = A->offsets;
01554     REAL *diag = A->diag;
01555     REAL **offdiag = A->offdiag;
01556
01557     // information about neighbors
01558     INT nneigh;
01559     if (!neigh) {
01560         nneigh = 0;
01561     }
01562     else {
01563         nneigh = neigh->row/ngrid;
01564     }
01565
01566     // local variable
01567     INT i, j, k, l, m, n, nbd, p;
01568     INT count;
01569     INT block_size;
01570     INT mem_inv = 0;
01571     INT mem_pivot = 0;
01572
01573     // allocation
01574     REAL *temp = (REAL *)fasp_mem_calloc(((nneigh+1)*nc)*((nneigh+1)*nc)*ngrid, sizeof(REAL));
01575     INT *tmp = (INT *)fasp_mem_calloc(((nneigh+1)*nc)*ngrid, sizeof(INT));
01576
01577     // main loop
01578     for (i=0; i<ngrid; ++i) {
01579         // count number of neighbors of node i
01580         count = 1;
01581         for (l=0; l<nneigh; ++l) {
01582             if (neigh->val[i*nneigh+l] >= 0) count++ ;
01583         }
01584
01585         // prepare the inverse of diagonal block i
01586         block_size = count*nc;
01587
01588         diaginv[i].row = block_size*block_size;
01589         diaginv[i].val = temp + mem_inv;
01590         mem_inv += diaginv[i].row;
01591
01592         pivot[i].row = block_size;
01593         pivot[i].val = tmp + mem_pivot;
01594         mem_pivot += pivot[i].row;
01595
01596         // put the diagonal block corresponding to node i
01597         for (j=0; j<nc; ++j) {
01598             for (k=0; k<nc; ++k) {
01599                 diaginv[i].val[j*block_size+k] = diag[i*nc*nc + j*nc + k];
01600             }
01601         }
01602
01603         // put the blocks corresponding to the neighbor of node i
01604         count = 1;
01605         for (l=0; l<nneigh; ++l) {
01606             p = neigh->val[i*nneigh+l];
01607             if (p >= 0) {
01608                 // put the diagonal block corresponding to this neighbor
01609                 for (j=0; j<nc; ++j) {
01610                     for (k=0; k<nc; ++k) {
01611                         m = count*nc + j; n = count*nc+k;
01612                         diaginv[i].val[m*block_size+n] = diag[p*nc*nc+j*nc+k];
01613                     }
01614                 }
01615
01616                 for (nbd=0; nbd<nband; nbd++) {
01617                     // put the block corresponding to (i, p)
01618                     if ( offsets[nbd] == (p-i) ) {
01619                         for (j=0; j<nc; ++j) {
01620                             for (k=0; k<nc; ++k) {
01621                                 m = j; n = count*nc + k;
01622                                 diaginv[i].val[m*block_size+n] = offdiag[nbd][(p-MAX(p-i,
01623                                     0))*nc*nc+j*nc+k];
01624                             }
01625                         }

```

```

01626
01627         // put the block corresponding to (p, i)
01628         if ( offsets[nbd] == (i-p) ) {
01629             for (j=0; j<nc; ++j) {
01630                 for(k=0; k<nc; ++k) {
01631                     m = count*nc + j; n = k;
01632                     diaginv[i].val[m*block_size+n] = offdiag[nbd] [(i-MAX(i-p,
01633 0))*nc*nc+j*nc+k];
01634                 }
01635             }
01636         }
01637         count++;
01638     } //end if
01639 } // end for (l=0; l<nneigh; ++l)
01640
01641 //fasp_smat_inv(diaginv[i].val, block_size);
01642 fasp_smat_lu_decomp(diaginv[i].val, pivot[i].val, block_size);
01643
01644 } // end of main loop
01645 }
01646
01647 void fasp_smoother_dstr_swz (dSTRmat *A,
01648                             dvector *b,
01649                             dvector *u,
01650                             dvector *diaginv,
01651                             ivector *pivot,
01652                             ivector *neigh,
01653                             ivector *order)
01654 {
01655     // information about A
01656     const INT ngrid = A->ngrid;
01657     const INT nc = A->nc;
01658
01659     // information about neighbors
01660     INT nneigh;
01661     if (!neigh) {
01662         nneigh = 0;
01663     }
01664     else {
01665         nneigh= neigh->row/ngrid;
01666     }
01667
01668     // local variable
01669     INT i, j, k, l, p, ti;
01670
01671     // work space
01672     REAL *temp = (REAL *)fasp_mem_malloc(b->row + (nneigh+1)*nc + (nneigh+1)*nc, sizeof(REAL));
01673     dvector r, e, ri;
01674     r.row = b->row; r.val = temp;
01675     e.row = (nneigh+1)*nc; e.val = temp + b->row;
01676     ri.row = (nneigh+1)*nc; ri.val = temp + b->row + (nneigh+1)*nc;
01677
01678     // initial residual
01679     fasp_dvec_cp(b,&r); fasp_blas_dstr_aApy(-1.0,A,u->val,r.val);
01680
01681     // main loop
01682     if (!order) {
01683         for (i=0; i<ngrid; ++i) {
01684             //-----
01685             // right hand side for A_ii e_i = r_i
01686             // rhs corresponding to node i
01687             for (j=0; j<nc; ++j) {
01688                 ri.val[j] = r.val[i*nc + j];
01689             }
01690             // rhs corresponding to the neighbors of node i
01691             k = 1;
01692             for (l=0; l<nneigh; ++l) {
01693                 p=neigh->val[nneigh*i+l];
01694                 if ( p>=0 ) {
01695                     for (j=0; j<nc; ++j) {
01696                         ri.val[k*nc+j] = r.val[p*nc+j];
01697                     }
01698                     ++k;
01699                 } // end if
01700             }
01701
01702             ri.row = k*nc;
01703             //-----
01704             //-----
01705         }
01706     }

```



```

01724         // solve local problem
01725         e.row = k*nc;
01726         //fasp_blas_smat_mxv(diaginv[ti].val, ri.val, k*nc, e.val);
01727         fasp_smat_lu_solve(diaginv[i].val, ri.val, pivot[i].val, e.val, k*nc);
01728         //-----
01729         //-----
01730         // update solution
01731         // solution corresponding to node i
01732         for (j=0; j<nc; ++j) {
01733             u->val[i*nc + j] += e.val[j];
01734         }
01735         // solution corresponding to the neighbor of node i
01736         k = 1;
01737         for (l=0; l<nneigh; ++l) {
01738             p=neigh->val[nneigh*i+l];
01739             if ( p>=0 ) {
01740                 for (j=0; j<nc; ++j) {
01741                     u->val[p*nc+j] += e.val[k*nc+j];
01742                 }
01743             }
01744             ++k;
01745         } // end if
01746     }
01747     //-----
01748     //-----
01749     // update residule
01750     fasp_dvec_cp(b,&r); fasp_blas_dstr_aAxy(-1.0,A,u->val,r.val);
01751 }
01752 }
01753 else {
01754     for (i=0; i<ngrid; ++i) {
01755         ti = order->val[i];
01756         //-----
01757         // right hand side for A_ii e_i = r_i
01758         // rhs corresponding to node i
01759         for (j=0; j<nc; ++j) {
01760             ri.val[j] = r.val[ti*nc + j];
01761         }
01762         // rhs corresponding to the neighbors of node i
01763         k = 1;
01764         for (l=0; l<nneigh; ++l) {
01765             p=neigh->val[nneigh*ti+l];
01766             if ( p>=0 ) {
01767                 for (j=0; j<nc; ++j) {
01768                     ri.val[k*nc+j] = r.val[p*nc+j];
01769                 }
01770             }
01771             ++k;
01772         } // end if
01773     }
01774
01775     ri.row = k*nc;
01776     //-----
01777     //-----
01778     // solve local problem
01779     e.row = k*nc;
01780     //fasp_blas_smat_mxv(diaginv[ti].val, ri.val, k*nc, e.val);
01781     fasp_smat_lu_solve(diaginv[ti].val, ri.val, pivot[ti].val, e.val, k*nc);
01782     //-----
01783     //-----
01784     // update solution
01785     // solution corresponding to node i
01786     for (j=0; j<nc; ++j) {
01787         u->val[ti*nc + j] += e.val[j];
01788     }
01789     // solution corresponding to the neighbor of node i
01790     k = 1;
01791     for (l=0; l<nneigh; ++l) {
01792         p=neigh->val[nneigh*ti+l];
01793         if ( p>=0 ) {
01794             for (j=0; j<nc; ++j) {
01795                 u->val[p*nc+j] += e.val[k*nc+j];
01796             }
01797         }
01798         ++k;
01799     } // end if
01800 }
01801 //-----
01802 //-----
01803 // update residule
01804 fasp_dvec_cp(b,&r); fasp_blas_dstr_aAxy(-1.0,A,u->val,r.val);

```

```

01805     }
01806     } // end of main loop
01807 }
01808
01809
01810 /*-----*/
01811 /*--      Private Functions      --*/
01812 /*-----*/
01813
01833 static void blkcontr2 (INT  start_data,
01834                       INT  start_vecx,
01835                       INT  start_vecy,
01836                       INT  nc,
01837                       REAL *data,
01838                       REAL *x,
01839                       REAL *y)
01840 {
01841     INT i,j,k,m;
01842     if (start_vecy == 0) {
01843         for (i = 0; i < nc; i++) {
01844             k = start_data + i*nc;
01845             for (j = 0; j < nc; j++) {
01846                 y[i] -= data[k+j]*x[start_vecx+j];
01847             }
01848         }
01849     }
01850     else {
01851         for (i = 0; i < nc; i++) {
01852             k = start_data + i*nc;
01853             m = start_vecy + i;
01854             for (j = 0; j < nc; j++) {
01855                 y[m] -= data[k+j]*x[start_vecx+j];
01856             }
01857         }
01858     }
01859 }
01860
01876 static void aAxpby (REAL  alpha,
01877                    REAL  beta,
01878                    INT   size,
01879                    REAL *A,
01880                    REAL *x,
01881                    REAL *y)
01882 {
01883     INT i,j;
01884     REAL tmp = 0.0;
01885     if (alpha == 0) {
01886         for (i = 0; i < size; i++) {
01887             y[i] *= beta;
01888         }
01889         return;
01890     }
01891     tmp = beta / alpha;
01892     // y:=(beta/alpha)y
01893     for (i = 0; i < size; i++) {
01894         y[i] *= tmp;
01895     }
01896     // y:=y+Ax
01897     for (i = 0; i < size; i++) {
01898         for (j = 0; j < size; j++) {
01899             y[i] += A[i*size+j]*x[j];
01900         }
01901     }
01902     // y:=alpha*y
01903     for (i = 0; i < size; i++) {
01904         y[i] *= alpha;
01905     }
01906 }
01907
01908 /*-----*/
01909 /*--      End of File      --*/
01910 /*-----*/

```

9.109 KryPbcgs.c File Reference

Krylov subspace methods – Preconditioned BiCGstab.

```
#include <math.h>
#include <float.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"
```

Functions

- [INT fasp_solver_dcsr_pbcgs](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ for CSR matrix.
- [INT fasp_solver_dbsr_pbcgs](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ for BSR matrix.
- [INT fasp_solver_dblc_pbcgs](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ for BLC matrix.
- [INT fasp_solver_dstr_pbcgs](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ for STR matrix.
- [INT fasp_solver_pbcgs](#) ([mxv_matfree](#) *mf, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$.

9.109.1 Detailed Description

Krylov subspace methods – Preconditioned BiCGstab.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

This version is based on Matlab 2011a – Chunsheng Feng

See [KrySPbcgs.c](#) for a safer version

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2016–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPbcgs.c](#).

9.109.2 Function Documentation

9.109.2.1 fasp_solver_dblc_pbcgs()

```
INT fasp_solver_dblc_pbcgs (
    dBLCmat * A,
    dvector * b,
    dvector * u,
```

```

precond * pc,
const REAL tol,
const INT MaxIt,
const SHORT StopType,
const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ for BLC matrix.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>u</i>	Pointer to dvector of DOFs
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chunsheng Feng

Date

03/04/2016

Definition at line 713 of file [KryPbcgs.c](#).

9.109.2.2 fasp_solver_dbsr_pbcgs()

```

INT fasp_solver_dbsr_pbcgs (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ for BSR matrix.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>u</i>	Pointer to dvector of DOFs
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping

Parameters

<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chunsheng Feng

Date

03/04/2016

Definition at line 387 of file [KryPbcgs.c](#).

9.109.2.3 fasp_solver_dcsr_pbcgs()

```
INT fasp_solver_dcsr_pbcgs (
    dCSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving $Au=b$ for CSR matrix.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>u</i>	Pointer to dvector of DOFs
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chunsheng Feng

Date

03/04/2016

Definition at line 62 of file [KryPbcgs.c](#).**9.109.2.4 fasp_solver_dstr_pbcgs()**

```

INT fasp_solver_dstr_pbcgs (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ for STR matrix.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>u</i>	Pointer to dvector of DOFs
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chunsheng Feng

Date

03/04/2016

Definition at line 1039 of file [KryPbcgs.c](#).**9.109.2.5 fasp_solver_pbcgs()**

```

INT fasp_solver_pbcgs (
    mxv_matfree * mf,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,

```

```
const SHORT StopType,
const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving $Au=b$.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector of right hand side
<i>u</i>	Pointer to dvector of DOFs
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chunsheng Feng

Date

03/04/2016

Definition at line 1365 of file [KryPbcgs.c](#).

9.110 KryPbcgs.c

[Go to the documentation of this file.](#)

```
00001
00025 #include <math.h>
00026 #include <float.h>
00027
00028 #include "fasp.h"
00029 #include "fasp_funcs.h"
00030
00031 /*-----*/
00032 /*-- Declare Private Functions --*/
00033 /*-----*/
00034
00035 #include "KryUtil.inl"
00036
00037 /*-----*/
00038 /*-- Public Functions --*/
00039 /*-----*/
00040
00062 INT fasp_solver_dcsr_pbcgs (dCSRmat *A,
00063                             dvector *b,
00064                             dvector *u,
00065                             precondition *pc,
00066                             const REAL tol,
00067                             const INT MaxIt,
00068                             const SHORT StopType,
00069                             const SHORT PrtLvl)
00070 {
00071     const INT m = b->row;
00072
00073     // local variables
00074     REAL n2b, tol_b;
00075     INT iter=0, stag = 1, moresteps = 1, maxmsteps=1;
```

```

00076     INT      flag, maxstagsteps, half_step=0;
00077     REAL      absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
00078     REAL      alpha,beta,omega,rho,rhol,rtv,tt;
00079     REAL      normr,normr_act,normph,normx,imin;
00080     REAL      norm_sh,norm_xhalf,normrmin,factor;
00081     REAL      *x = u->val, *bval=b->val;
00082
00083     // allocate temp memory (need 10*m REAL)
00084     REAL *work=(REAL *)fasp_mem_malloc(10*m,sizeof(REAL));
00085     REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
00086     REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
00087     REAL *t = sh+m, *xmin = t+m;
00088
00089     // Output some info for debugging
00090     if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (CSR) ...\n");
00091
00092     #if DEBUG_MODE > 0
00093     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00094     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00095     #endif
00096
00097     // r = b-A*u
00098     fasp_darray_cp(m,bval,r);
00099     n2b = fasp_blas_darray_norm2(m,r);
00100
00101     flag = 1;
00102     fasp_darray_cp(m,x,xmin);
00103     imin = 0;
00104
00105     iter = 0;
00106
00107     tol_b = n2b*tol;
00108
00109     fasp_blas_dcsr_aAxy(-1.0, A, x, r);
00110     normr = fasp_blas_darray_norm2(m,r);
00111     normr_act = normr;
00112     relres = normr/n2b;
00113
00114     // if initial residual is small, no need to iterate!
00115     if ( normr <= tol_b ) {
00116         flag = 0;
00117         iter = 0;
00118         goto FINISHED;
00119     }
00120
00121     // output iteration information if needed
00122     fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);
00123
00124     // shadow residual rt = r* := r
00125     fasp_darray_cp(m,r,rt);
00126     normrmin = normr;
00127
00128     rho = 1.0;
00129     omega = 1.0;
00130     stag = 0;
00131     alpha = 0.0;
00132
00133     moresteps = 0;
00134     maxmsteps = 10;
00135     maxstagsteps = 3;
00136
00137     // loop over maxit iterations (unless convergence or failure)
00138     for (iter=1;iter <= MaxIt;iter++) {
00139
00140         rhol = rho;
00141         rho = fasp_blas_darray_dotprod(m,rt,r);
00142
00143         if ((rho ==0.0) || (ABS(rho) >= DBL_MAX )) {
00144             flag = 4;
00145             goto FINISHED;
00146         }
00147
00148         if (iter==1) {
00149             fasp_darray_cp(m,r,p);
00150         }
00151         else {
00152             beta = (rho/rhol)*(alpha/omega);
00153
00154             if ((beta == 0) || (ABS(beta) > DBL_MAX )) {
00155                 flag = 4;
00156                 goto FINISHED;

```



```

00157     }
00158
00159     // p = r + beta * (p - omega * v);
00160     fasp_blas_darray_axpy(m,-omega,v,p); //p=p - omega*v
00161     fasp_blas_darray_axpy(m,1.0, r, beta, p); //p = 1.0*r +beta*p
00162 }
00163
00164 // pp = precondition(p) ,ph
00165 if ( pc != NULL )
00166     pc->fct(p,ph,pc->data); /* Apply preconditioner */
00167 // if ph all is infinite then exit need add
00168 else
00169     fasp_darray_cp(m,p,ph); /* No preconditioner */
00170
00171 // v = A*ph
00172 fasp_blas_dcsr_mxv(A,ph,v);
00173 rtv = fasp_blas_darray_dotprod(m,rt,v);
00174
00175 if ( ( rtv==0.0 ) || ( ABS(rtv) > DBL_MAX ) ){
00176     flag = 4;
00177     goto FINISHED;
00178 }
00179
00180 alpha = rho/rtv;
00181
00182 if ( ABS(alpha) > DBL_MAX ){
00183     flag = 4;
00184     ITS_DIVZERO;
00185     goto FINISHED;
00186 }
00187
00188 normx = fasp_blas_darray_norm2(m,x);
00189 normph = fasp_blas_darray_norm2(m,ph);
00190 if (ABS(alpha)*normph < DBL_EPSILON*normx )
00191     stag = stag + 1;
00192 else
00193     stag = 0;
00194
00195 // xhalf = x + alpha * ph; // form the "half" iterate
00196 // s = r - alpha * v; // residual associated with xhalf
00197 fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf); // z= ax + y
00198 fasp_blas_darray_axpyz(m, -alpha, v, r, s);
00199 normr = fasp_blas_darray_norm2(m,s); // normr = norm(s);
00200 normr_act = normr;
00201
00202 // compute reduction factor of residual ||r||
00203 absres = normr_act;
00204 factor = absres/absres0;
00205 fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);
00206
00207 // check for convergence
00208 if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
00209 {
00210     fasp_darray_cp(m,bval,s);
00211     fasp_blas_dcsr_aAxy(-1.0,A,xhalf,s);
00212     normr_act = fasp_blas_darray_norm2(m,s);
00213
00214     if (normr_act <= tol) {
00215         // x = xhalf;
00216         fasp_darray_cp(m,xhalf,x); // x = xhalf;
00217         flag = 0;
00218         imin = iter - 0.5;
00219         half_step++;
00220         if ( PrtLvl >= PRINT_MORE )
00221             printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00222                 flag,stag,imin,half_step);
00223         goto FINISHED;
00224     }
00225     else {
00226         if ( (stag >= maxstagsteps) && (moresteps == 0) ) stag = 0;
00227
00228         moresteps = moresteps + 1;
00229         if (moresteps >= maxmsteps) {
00230             // if ~warned
00231             flag = 3;
00232             fasp_darray_cp(m,xhalf,x);
00233             goto FINISHED;
00234         }
00235     }
00236 }
00237

```

```

00238     if ( stag >= maxstagsteps ) {
00239         flag = 3;
00240         goto FINISHED;
00241     }
00242
00243     if ( normr_act < normrmin ) // update minimal norm quantities
00244     {
00245         normrmin = normr_act;
00246         fasp_darray_cp(m,xhalf,xmin);
00247         imin = iter - 0.5;
00248         half_step++;
00249         if ( PrtLvl >= PRINT_MORE )
00250             printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00251                 flag,stag,imin,half_step);
00252     }
00253
00254     // sh = precondition(s)
00255     if ( pc != NULL ) {
00256         pc->fct(s,sh,pc->data); /* Apply preconditioner */
00257     }
00258     else
00259         fasp_darray_cp(m,s,sh); /* No preconditioner */
00260
00261     // t = A*sh;
00262     fasp_blas_dcsr_mxv(A,sh,t);
00263     // tt = t' * t;
00264     tt = fasp_blas_darray_dotprod(m,t,t);
00265     if ( (tt == 0) || (tt >= DBL_MAX) ) {
00266         flag = 4;
00267         goto FINISHED;
00268     }
00269
00270     // omega = (t' * s) / tt;
00271     omega = fasp_blas_darray_dotprod(m,s,t)/tt;
00272     if ( ABS(omega) > DBL_MAX ) {
00273         flag = 4;
00274         goto FINISHED;
00275     }
00276
00277     norm_sh = fasp_blas_darray_norm2(m,sh);
00278     norm_xhalf = fasp_blas_darray_norm2(m,xhalf);
00279
00280     if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
00281         stag = stag + 1;
00282     else
00283         stag = 0;
00284
00285     fasp_blas_darray_axpyz(m, omega,sh,xhalf, x); // x = xhalf + omega * sh;
00286     fasp_blas_darray_axpyz(m, -omega, t, s, r); // r = s - omega * t;
00287     normr = fasp_blas_darray_norm2(m,r); // normr = norm(r);
00288     normr_act = normr;
00289
00290     // check for convergence
00291     if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
00292     {
00293         fasp_darray_cp(m,bval,r);
00294         fasp_blas_dcsr_aAxy(-1.0,A,x,r);
00295         normr_act = fasp_blas_darray_norm2(m,r);
00296         if ( normr_act <= tol ) {
00297             flag = 0;
00298             goto FINISHED;
00299         }
00300         else {
00301             if ( (stag >= maxstagsteps) && (moresteps == 0) ) stag = 0;
00302
00303             moresteps = moresteps + 1;
00304             if ( moresteps >= maxmsteps ) {
00305                 flag = 3;
00306                 goto FINISHED;
00307             }
00308         }
00309     }
00310
00311     // update minimal norm quantities
00312     if ( normr_act < normrmin ) {
00313         normrmin = normr_act;
00314         fasp_darray_cp(m,x,xmin);
00315         imin = iter;
00316     }
00317
00318     if ( stag >= maxstagsteps ) {

```

```

00319         flag = 3;
00320         goto FINISHED;
00321     }
00322
00323     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00324
00325     absres0 = absres;
00326 } // for iter = 1 : maxit
00327
00328 FINISHED: // finish iterative method
00329 // returned solution is first with minimal residual
00330 if (flag == 0)
00331     relres = normr_act / n2b;
00332 else {
00333     fasp_darray_cp(m, bval,r);
00334     fasp_blas_dcsr_aAxy(-1.0,A,xmin,r);
00335     normr = fasp_blas_darray_norm2(m,r);
00336
00337     if ( normr <= normr_act ) {
00338         fasp_darray_cp(m, xmin, x);
00339         iter = imin;
00340         relres = normr/n2b;
00341     }
00342     else {
00343         relres = normr_act/n2b;
00344     }
00345 }
00346
00347 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00348
00349 if ( PrtLvl >= PRINT_MORE )
00350     printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00351         flag,stag,imin,half_step);
00352
00353 // clean up temp memory
00354 fasp_mem_free(work); work = NULL;
00355
00356 #if DEBUG_MODE > 0
00357 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00358 #endif
00359
00360 if ( iter > MaxIt )
00361     return ERROR_SOLVER_MAXIT;
00362 else
00363     return iter;
00364 }
00365
00387 INT fasp_solver_dbsr_pbcgs (dBSRmat      *A,
00388                             dvector      *b,
00389                             dvector      *u,
00390                             precondition *pc,
00391                             const REAL    tol,
00392                             const INT     MaxIt,
00393                             const SHORT   StopType,
00394                             const SHORT   PrtLvl)
00395 {
00396     const INT    m = b->row;
00397
00398     // local variables
00399     REAL    n2b,tolb;
00400     INT     iter=0, stag = 1, moresteps = 1, maxmsteps=1;
00401     INT     flag, maxstagsteps, half_step=0;
00402     REAL    absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
00403     REAL    alpha,beta,omega,rho,rhol,rtv,tt;
00404     REAL    normr,normr_act,normph,normx,imin;
00405     REAL    norm_sh,norm_xhalf,normrmin,factor;
00406     REAL    *x = u->val, *bval=b->val;
00407
00408     // allocate temp memory (need 10*m REAL)
00409     REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
00410     REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
00411     REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
00412     REAL *t = sh+m, *xmin = t+m;
00413
00414     // Output some info for debugging
00415     if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (BSR) ...\n");
00416
00417 #if DEBUG_MODE > 0
00418     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00419     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00420 #endif

```

```

00421
00422 // r = b-A*u
00423 fasp_darray_cp(m,bval,r);
00424 n2b = fasp_blas_darray_norm2(m,r);
00425
00426 flag = 1;
00427 fasp_darray_cp(m,x,xmin);
00428 imin = 0;
00429
00430 iter = 0;
00431
00432 tol = n2b*tol;
00433
00434 fasp_blas_dbsr_aApy(-1.0, A, x, r);
00435 normr = fasp_blas_darray_norm2(m,r);
00436 normr_act = normr;
00437 relres = normr/n2b;
00438
00439 // if initial residual is small, no need to iterate!
00440 if ( normr <= tol ) {
00441     flag = 0;
00442     iter = 0;
00443     goto FINISHED;
00444 }
00445
00446 // output iteration information if needed
00447 fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);
00448
00449 // shadow residual rt = r* := r
00450 fasp_darray_cp(m,r,rt);
00451 normrmin = normr;
00452
00453 rho = 1.0;
00454 omega = 1.0;
00455 stag = 0;
00456 alpha = 0.0;
00457
00458 moresteps = 0;
00459 maxmsteps = 10;
00460 maxstagsteps = 3;
00461
00462 // loop over maxit iterations (unless convergence or failure)
00463 for (iter=1;iter <= MaxIt;iter++) {
00464
00465     rho1 = rho;
00466     rho = fasp_blas_darray_dotprod(m,rt,r);
00467
00468     if ((rho==0.0) || (ABS(rho) >= DBL_MAX )) {
00469         flag = 4;
00470         goto FINISHED;
00471     }
00472
00473     if (iter==1) {
00474         fasp_darray_cp(m,r,p);
00475     }
00476     else {
00477         beta = (rho/rho1)*(alpha/omega);
00478
00479         if ((beta == 0) || (ABS(beta) > DBL_MAX )) {
00480             flag = 4;
00481             goto FINISHED;
00482         }
00483
00484         // p = r + beta * (p - omega * v);
00485         fasp_blas_darray_axpy(m,-omega,v,p); //p=p - omega*v
00486         fasp_blas_darray_axpby(m,1.0, r, beta, p); //p = 1.0*r +beta*p
00487     }
00488
00489     // pp = precondition(p) ,ph
00490     if ( pc != NULL )
00491         pc->fct(p,ph,pc->data); /* Apply preconditioner */
00492     // if ph all is infinite then exit need add
00493     else
00494         fasp_darray_cp(m,p,ph); /* No preconditioner */
00495
00496     // v = A*ph
00497     fasp_blas_dbsr_mxv(A,ph,v);
00498     rtv = fasp_blas_darray_dotprod(m,rt,v);
00499
00500     if (( rtv==0.0 ) || ( ABS(rtv) > DBL_MAX )) {
00501         flag = 4;

```

```

00502         goto FINISHED;
00503     }
00504
00505     alpha = rho/rtv;
00506
00507     if ( ABS(alpha) > DBL_MAX ) {
00508         flag = 4;
00509         ITS_DIVZERO;
00510         goto FINISHED;
00511     }
00512
00513     normx = fasp_blas_darray_norm2(m,x);
00514     normph = fasp_blas_darray_norm2(m,ph);
00515     if (ABS(alpha)*normph < DBL_EPSILON*normx )
00516         stag = stag + 1;
00517     else
00518         stag = 0;
00519
00520     // xhalf = x + alpha * ph;          // form the "half" iterate
00521     // s = r - alpha * v;              // residual associated with xhalf
00522     fasp_blas_darray_axpyz(m, alpha, ph, x, xhalf); // z= ax + y
00523     fasp_blas_darray_axpyz(m, -alpha, v, r, s);
00524     normr = fasp_blas_darray_norm2(m,s); // normr = norm(s);
00525     normr_act = normr;
00526
00527     // compute reduction factor of residual ||r||
00528     absres = normr_act;
00529     factor = absres/absres0;
00530     fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);
00531
00532     // check for convergence
00533     if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
00534     {
00535         fasp_darray_cp(m,bval,s);
00536         fasp_blas_dbsr_aAxy(-1.0,A,xhalf,s);
00537         normr_act = fasp_blas_darray_norm2(m,s);
00538
00539         if (normr_act <= tol) {
00540             // x = xhalf;
00541             fasp_darray_cp(m,xhalf,x); // x = xhalf;
00542             flag = 0;
00543             imin = iter - 0.5;
00544             half_step++;
00545             if ( PrtLvl >= PRINT_MORE )
00546                 printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00547                     flag,stag,imin,half_step);
00548             goto FINISHED;
00549         }
00550         else {
00551             if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
00552
00553             moresteps = moresteps + 1;
00554             if (moresteps >= maxmsteps){
00555                 // if ~warned
00556                 flag = 3;
00557                 fasp_darray_cp(m,xhalf,x);
00558                 goto FINISHED;
00559             }
00560         }
00561     }
00562
00563     if ( stag >= maxstagsteps ) {
00564         flag = 3;
00565         goto FINISHED;
00566     }
00567
00568     if ( normr_act < normrmin ) // update minimal norm quantities
00569     {
00570         normrmin = normr_act;
00571         fasp_darray_cp(m,xhalf,xmin);
00572         imin = iter - 0.5;
00573         half_step++;
00574         if ( PrtLvl >= PRINT_MORE )
00575             printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00576                 flag,stag,imin,half_step);
00577     }
00578
00579     // sh = precondition(s)
00580     if ( pc != NULL ) {
00581         pc->fct(s,sh,pc->data); /* Apply preconditioner */
00582     }

```

```

00583     else
00584         fasp_darray_cp(m,s,sh); /* No preconditioner */
00585
00586         // t = A*sh;
00587         fasp_blas_dbsr_mxv(A,sh,t);
00588         // tt = t' * t;
00589         tt = fasp_blas_darray_dotprod(m,t,t);
00590         if ( (tt == 0) || (tt >= DBL_MAX) ) {
00591             flag = 4;
00592             goto FINISHED;
00593         }
00594
00595         // omega = (t' * s) / tt;
00596         omega = fasp_blas_darray_dotprod(m,s,t)/tt;
00597         if ( ABS(omega) > DBL_MAX ) {
00598             flag = 4;
00599             goto FINISHED;
00600         }
00601
00602         norm_sh = fasp_blas_darray_norm2(m,sh);
00603         norm_xhalf = fasp_blas_darray_norm2(m,xhalf);
00604
00605         if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
00606             stag = stag + 1;
00607         else
00608             stag = 0;
00609
00610         fasp_blas_darray_axpyz(m, omega,sh,xhalf, x); // x = xhalf + omega * sh;
00611         fasp_blas_darray_axpyz(m, -omega, t, s, r); // r = s - omega * t;
00612         normr = fasp_blas_darray_norm2(m,r); // normr = norm(r);
00613         normr_act = normr;
00614
00615         // check for convergence
00616         if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
00617         {
00618             fasp_darray_cp(m,bval,r);
00619             fasp_blas_dbsr_aAxy(-1.0,A,x,r);
00620             normr_act = fasp_blas_darray_norm2(m,r);
00621             if ( normr_act <= tol ) {
00622                 flag = 0;
00623                 goto FINISHED;
00624             }
00625             else {
00626                 if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
00627
00628                 moresteps = moresteps + 1;
00629                 if ( moresteps >= maxmsteps ) {
00630                     flag = 3;
00631                     goto FINISHED;
00632                 }
00633             }
00634         }
00635
00636         // update minimal norm quantities
00637         if ( normr_act < normrmin ) {
00638             normrmin = normr_act;
00639             fasp_darray_cp(m,x,xmin);
00640             imin = iter;
00641         }
00642
00643         if ( stag >= maxstagsteps )
00644         {
00645             flag = 3;
00646             goto FINISHED;
00647         }
00648
00649         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00650
00651         absres0 = absres;
00652     } // for iter = 1 : maxit
00653
00654 FINISHED: // finish iterative method
00655 // returned solution is first with minimal residual
00656 if (flag == 0)
00657     relres = normr_act / n2b;
00658 else {
00659     fasp_darray_cp(m, bval,r);
00660     fasp_blas_dbsr_aAxy(-1.0,A,xmin,r);
00661     normr = fasp_blas_darray_norm2(m,r);
00662
00663     if ( normr <= normr_act ) {

```

```

00664         fasp_darray_cp(m, xmin,x);
00665         iter = imin;
00666         relres = normr/n2b;
00667     }
00668     else {
00669         relres = normr_act/n2b;
00670     }
00671 }
00672
00673 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00674
00675 if ( PrtLvl >= PRINT_MORE )
00676     printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00677         flag,stag,imin,half_step);
00678
00679 // clean up temp memory
00680 fasp_mem_free(work); work = NULL;
00681
00682 #if DEBUG_MODE > 0
00683     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00684 #endif
00685
00686 if ( iter > MaxIt )
00687     return ERROR_SOLVER_MAXIT;
00688 else
00689     return iter;
00690 }
00691
00713 INT fasp_solver_dblc_pbcgs (dBLMat      *A,
00714                             dvector     *b,
00715                             dvector     *u,
00716                             precondition *pc,
00717                             const REAL  tol,
00718                             const INT    MaxIt,
00719                             const SHORT  StopType,
00720                             const SHORT  PrtLvl)
00721 {
00722     const INT    m = b->row;
00723
00724     // local variables
00725     REAL  n2b,tolb;
00726     INT    iter=0, stag = 1, moresteps = 1, maxmsteps=1;
00727     INT    flag, maxstagsteps, half_step=0;
00728     REAL  absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
00729     REAL  alpha,beta,omega,rho,rhol,rtv,tt;
00730     REAL  normr,normr_act,normph,normx,imin;
00731     REAL  norm_sh,norm_xhalf,normrmin,factor;
00732     REAL  *x = u->val, *bval=b->val;
00733
00734     // allocate temp memory (need 10*m REAL)
00735     REAL *work=(REAL *)fasp_mem_malloc(10*m,sizeof(REAL));
00736     REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
00737     REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
00738     REAL *t = sh+m, *xmin = t+m;
00739
00740     // Output some info for debugging
00741     if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (BLC) ...\n");
00742
00743 #if DEBUG_MODE > 0
00744     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00745     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00746 #endif
00747
00748     // r = b-A*u
00749     fasp_darray_cp(m,bval,r);
00750     n2b = fasp_blas_darray_norm2(m,r);
00751
00752     flag = 1;
00753     fasp_darray_cp(m,x,xmin);
00754     imin = 0;
00755
00756     iter = 0;
00757
00758     tolb = n2b*tol;
00759
00760     fasp_blas_dblc_aAxy(-1.0, A, x, r);
00761     normr = fasp_blas_darray_norm2(m,r);
00762     normr_act = normr;
00763     relres = normr/n2b;
00764
00765     // if initial residual is small, no need to iterate!

```

```

00766     if ( normr <= tolB ) {
00767         flag = 0;
00768         iter = 0;
00769         goto FINISHED;
00770     }
00771
00772     // output iteration information if needed
00773     fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);
00774
00775     // shadow residual rt = r* := r
00776     fasp_darray_cp(m,r,rt);
00777     normrmin = normr;
00778
00779     rho = 1.0;
00780     omega = 1.0;
00781     stag = 0;
00782     alpha = 0.0;
00783
00784     moresteps = 0;
00785     maxmsteps = 10;
00786     maxstagsteps = 3;
00787
00788     // loop over maxit iterations (unless convergence or failure)
00789     for (iter=1;iter <= MaxIt;iter++) {
00790
00791         rho1 = rho;
00792         rho = fasp_blas_darray_dotprod(m,rt,r);
00793
00794         if ((rho == 0.0) || (ABS(rho) >= DBL_MAX )) {
00795             flag = 4;
00796             goto FINISHED;
00797         }
00798
00799         if (iter==1) {
00800             fasp_darray_cp(m,r,p);
00801         }
00802         else {
00803             beta = (rho/rho1)*(alpha/omega);
00804
00805             if ((beta == 0) || (ABS(beta) > DBL_MAX )) {
00806                 flag = 4;
00807                 goto FINISHED;
00808             }
00809
00810             // p = r + beta * (p - omega * v);
00811             fasp_blas_darray_axpy(m,-omega,v,p); //p=p - omega*v
00812             fasp_blas_darray_axpy(m,1.0, r, beta, p); //p = 1.0*r +beta*p
00813         }
00814
00815         // pp = precondition(p) ,ph
00816         if ( pc != NULL )
00817             pc->fct(p,ph,pc->data); /* Apply preconditioner */
00818         // if ph all is infinite then exit need add
00819         else
00820             fasp_darray_cp(m,p,ph); /* No preconditioner */
00821
00822         // v = A*ph
00823         fasp_blas_dblc_mxv(A,ph,v);
00824         rtv = fasp_blas_darray_dotprod(m,rt,v);
00825
00826         if ((rtv==0.0) || (ABS(rtv) > DBL_MAX )) {
00827             flag = 4;
00828             goto FINISHED;
00829         }
00830
00831         alpha = rho/rtv;
00832
00833         if (ABS(alpha) > DBL_MAX ) {
00834             flag = 4;
00835             ITS_DIVZERO;
00836             goto FINISHED;
00837         }
00838
00839         normx = fasp_blas_darray_norm2(m,x);
00840         normph = fasp_blas_darray_norm2(m,ph);
00841         if (ABS(alpha)*normph < DBL_EPSILON*normx )
00842             stag = stag + 1;
00843         else
00844             stag = 0;
00845
00846         // xhalf = x + alpha * ph;          // form the "half" iterate

```



```

00847      // s = r - alpha * v;          // residual associated with xhalf
00848      fasp_blas_darray_axpyz(m, alpha, ph, x, xhalf); // z = ax + y
00849      fasp_blas_darray_axpyz(m, -alpha, v, r, s);
00850      normr = fasp_blas_darray_norm2(m,s); // normr = norm(s);
00851      normr_act = normr;
00852
00853      // compute reduction factor of residual ||r||
00854      absres = normr_act;
00855      factor = absres/absres0;
00856      fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);
00857
00858      // check for convergence
00859      if ((normr <= tol) || (stag >= maxstagsteps) || moresteps)
00860      {
00861          fasp_darray_cp(m,bval,s);
00862          fasp_blas_dblc_aApy(-1.0,A,xhalf,s);
00863          normr_act = fasp_blas_darray_norm2(m,s);
00864
00865          if (normr_act <= tol) {
00866              // x = xhalf;
00867              fasp_darray_cp(m,xhalf,x); // x = xhalf;
00868              flag = 0;
00869              imin = iter - 0.5;
00870              half_step++;
00871              if (PrtLvl >= PRINT_MORE)
00872                  printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00873                      flag,stag,imin,half_step);
00874              goto FINISHED;
00875          }
00876          else {
00877              if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
00878
00879              moresteps = moresteps + 1;
00880              if (moresteps >= maxmsteps) {
00881                  // if ~warned
00882                  flag = 3;
00883                  fasp_darray_cp(m,xhalf,x);
00884                  goto FINISHED;
00885              }
00886          }
00887      }
00888
00889      if (stag >= maxstagsteps) {
00890          flag = 3;
00891          goto FINISHED;
00892      }
00893
00894      if (normr_act < normrmin) // update minimal norm quantities
00895      {
00896          normrmin = normr_act;
00897          fasp_darray_cp(m,xhalf,xmin);
00898          imin = iter - 0.5;
00899          half_step++;
00900          if (PrtLvl >= PRINT_MORE)
00901              printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
00902                  flag,stag,imin,half_step);
00903      }
00904
00905      // sh = precondition(s)
00906      if (pc != NULL) {
00907          pc->fct(s,sh,pc->data); /* Apply preconditioner */
00908      }
00909      else
00910          fasp_darray_cp(m,s,sh); /* No preconditioner */
00911
00912      // t = A*sh;
00913      fasp_blas_dblc_mxv(A,sh,t);
00914      // tt = t' * t;
00915      tt = fasp_blas_darray_dotprod(m,t,t);
00916      if ((tt == 0) || (tt >= DBL_MAX)) {
00917          flag = 4;
00918          goto FINISHED;
00919      }
00920
00921      // omega = (t' * s) / tt;
00922      omega = fasp_blas_darray_dotprod(m,s,t)/tt;
00923      if (ABS(omega) > DBL_MAX) {
00924          flag = 4;
00925          goto FINISHED;
00926      }
00927

```

```

00928     norm_sh = fasp_blas_darray_norm2(m,sh);
00929     norm_xhalf = fasp_blas_darray_norm2(m,xhalf);
00930
00931     if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
00932         stag = stag + 1;
00933     else
00934         stag = 0;
00935
00936     fasp_blas_darray_axpyz(m, omega,sh,xhalf, x); // x = xhalf + omega * sh;
00937     fasp_blas_darray_axpyz(m, -omega, t, s, r); // r = s - omega * t;
00938     normr = fasp_blas_darray_norm2(m,r); // normr = norm(r);
00939     normr_act = normr;
00940
00941     // check for convergence
00942     if ( ( normr <= tol ) || ( stag >= maxstagsteps ) || moresteps )
00943     {
00944         fasp_darray_cp(m,bval,r);
00945         fasp_blas_dblc_aApy(-1.0,A,x,r);
00946         normr_act = fasp_blas_darray_norm2(m,r);
00947         if ( normr_act <= tol ) {
00948             flag = 0;
00949             goto FINISHED;
00950         }
00951         else {
00952             if ( (stag >= maxstagsteps) && (moresteps == 0) ) stag = 0;
00953
00954             moresteps = moresteps + 1;
00955             if ( moresteps >= maxmsteps ) {
00956                 flag = 3;
00957                 goto FINISHED;
00958             }
00959         }
00960     }
00961
00962     // update minimal norm quantities
00963     if ( normr_act < normrmin ) {
00964         normrmin = normr_act;
00965         fasp_darray_cp(m,x,xmin);
00966         imin = iter;
00967     }
00968
00969     if ( stag >= maxstagsteps )
00970     {
00971         flag = 3;
00972         goto FINISHED;
00973     }
00974
00975     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00976
00977     absres0 = absres;
00978     } // for iter = 1 : maxit
00979
00980 FINISHED: // finish iterative method
00981 // returned solution is first with minimal residual
00982 if (flag == 0)
00983     relres = normr_act / n2b;
00984 else {
00985     fasp_darray_cp(m, bval,r);
00986     fasp_blas_dblc_aApy(-1.0,A,xmin,r);
00987     normr = fasp_blas_darray_norm2(m,r);
00988
00989     if ( normr <= normr_act ) {
00990         fasp_darray_cp(m, xmin,x);
00991         iter = imin;
00992         relres = normr/n2b;
00993     }
00994     else {
00995         relres = normr_act/n2b;
00996     }
00997 }
00998
00999 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01000
01001 if ( PrtLvl >= PRINT_MORE )
01002     printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01003         flag,stag,imin, half_step);
01004
01005 // clean up temp memory
01006 fasp_mem_free(work); work = NULL;
01007
01008 #if DEBUG_MODE > 0

```

```

01009     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01010 #endif
01011
01012     if ( iter > MaxIt )
01013         return ERROR_SOLVER_MAXIT;
01014     else
01015         return iter;
01016 }
01017
01039 INT fasp_solver_dstr_pbcgs (dSTRmat      *A,
01040                             dvector      *b,
01041                             dvector      *u,
01042                             precondition *pc,
01043                             const REAL    tol,
01044                             const INT     MaxIt,
01045                             const SHORT   StopType,
01046                             const SHORT   PrtLvl)
01047 {
01048     const INT    m = b->row;
01049
01050     // local variables
01051     REAL    n2b,tolb;
01052     INT     iter=0, stag = 1, moresteps = 1, maxmsteps=1;
01053     INT     flag, maxstagsteps, half_step=0;
01054     REAL    absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
01055     REAL    alpha,beta,omega,rho,rhol,rtv,tt;
01056     REAL    normr,normr_act,normph,normx,imin;
01057     REAL    norm_sh,norm_xhalf,normrmin,factor;
01058     REAL    *x = u->val, *bval=b->val;
01059
01060     // allocate temp memory (need 10*m REAL)
01061     REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
01062     REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
01063     REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
01064     REAL *t = sh+m, *xmin = t+m;
01065
01066     // Output some info for debugging
01067     if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (STR) ...\n");
01068
01069 #if DEBUG_MODE > 0
01070     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01071     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01072 #endif
01073
01074     // r = b-A*u
01075     fasp_darray_cp(m,bval,r);
01076     n2b = fasp_blas_darray_norm2(m,r);
01077
01078     flag = 1;
01079     fasp_darray_cp(m,x,xmin);
01080     imin = 0;
01081
01082     iter = 0;
01083
01084     tolb = n2b*tol;
01085
01086     fasp_blas_dstr_aAxy(-1.0, A, x, r);
01087     normr = fasp_blas_darray_norm2(m,r);
01088     normr_act = normr;
01089     relres = normr/n2b;
01090
01091     // if initial residual is small, no need to iterate!
01092     if ( normr <= tolb ) {
01093         flag = 0;
01094         iter = 0;
01095         goto FINISHED;
01096     }
01097
01098     // output iteration information if needed
01099     fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);
01100
01101     // shadow residual rt = r* := r
01102     fasp_darray_cp(m,r,rt);
01103     normrmin = normr;
01104
01105     rho = 1.0;
01106     omega = 1.0;
01107     stag = 0;
01108     alpha = 0.0;
01109
01110     moresteps = 0;

```

```

01111     maxmsteps = 10;
01112     maxstagsteps = 3;
01113
01114     // loop over maxit iterations (unless convergence or failure)
01115     for (iter=1; iter <= MaxIt; iter++) {
01116
01117         rho1 = rho;
01118         rho = fasp_blas_darray_dotprod(m,rt,r);
01119
01120         if ((rho ==0.0 ) || ( ABS(rho) >= DBL_MAX )) {
01121             flag = 4;
01122             goto FINISHED;
01123         }
01124
01125         if (iter==1) {
01126             fasp_darray_cp(m,r,p);
01127         }
01128         else {
01129             beta = (rho/rho1)*(alpha/omega);
01130
01131             if ((beta == 0) || ( ABS(beta) > DBL_MAX )) {
01132                 flag = 4;
01133                 goto FINISHED;
01134             }
01135
01136             // p = r + beta * (p - omega * v);
01137             fasp_blas_darray_axpy(m,-omega,v,p); //p=p - omega*v
01138             fasp_blas_darray_axpby(m,1.0, r, beta, p); //p = 1.0*r +beta*p
01139         }
01140
01141         // pp = precondition(p) ,ph
01142         if ( pc != NULL )
01143             pc->fct(p,ph,pc->data); /* Apply preconditioner */
01144         // if ph all is infinite then exit need add
01145         else
01146             fasp_darray_cp(m,p,ph); /* No preconditioner */
01147
01148         // v = A*ph
01149         fasp_blas_dstr_mxv(A,ph,v);
01150         rtv = fasp_blas_darray_dotprod(m,rt,v);
01151
01152         if (( rtv==0.0 ) || ( ABS(rtv) > DBL_MAX )) {
01153             flag = 4;
01154             goto FINISHED;
01155         }
01156
01157         alpha = rho/rtv;
01158
01159         if ( ABS(alpha) > DBL_MAX ) {
01160             flag = 4;
01161             ITS_DIVZERO;
01162             goto FINISHED;
01163         }
01164
01165         normx = fasp_blas_darray_norm2(m,x);
01166         normph = fasp_blas_darray_norm2(m,ph);
01167         if (ABS(alpha)*normph < DBL_EPSILON*normx )
01168             stag = stag + 1;
01169         else
01170             stag = 0;
01171
01172         // xhalf = x + alpha * ph; // form the "half" iterate
01173         // s = r - alpha * v; // residual associated with xhalf
01174         fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf); // z= ax + y
01175         fasp_blas_darray_axpyz(m, -alpha, v, r, s);
01176         normr = fasp_blas_darray_norm2(m,s); // normr = norm(s);
01177         normr_act = normr;
01178
01179         // compute reduction factor of residual ||r||
01180         absres = normr_act;
01181         factor = absres/absres0;
01182         fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);
01183
01184         // check for convergence
01185         if ((normr <= tolb) || (stag >= maxstagsteps) || moresteps)
01186         {
01187             fasp_darray_cp(m,bval,s);
01188             fasp_blas_dstr_aAxy(-1.0,A,xhalf,s);
01189             normr_act = fasp_blas_darray_norm2(m,s);
01190
01191             if (normr_act <= tolb){

```

```

01192         // x = xhalf;
01193         fasp_darray_cp(m,xhalf,x);      // x = xhalf;
01194         flag = 0;
01195         imin = iter - 0.5;
01196         half_step++;
01197         if ( PrtLvl >= PRINT_MORE )
01198             printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01199                   flag,stag,imin,half_step);
01200         goto FINISHED;
01201     }
01202     else {
01203         if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
01204
01205         moresteps = moresteps + 1;
01206         if (moresteps >= maxmsteps){
01207             // if ~warned
01208             flag = 3;
01209             fasp_darray_cp(m,xhalf,x);
01210             goto FINISHED;
01211         }
01212     }
01213 }
01214
01215 if ( stag >= maxstagsteps ) {
01216     flag = 3;
01217     goto FINISHED;
01218 }
01219
01220 if ( normr_act < normrmin )      // update minimal norm quantities
01221 {
01222     normrmin = normr_act;
01223     fasp_darray_cp(m,xhalf,xmin);
01224     imin = iter - 0.5;
01225     half_step++;
01226     if ( PrtLvl >= PRINT_MORE )
01227         printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01228               flag,stag,imin,half_step);
01229 }
01230
01231 // sh = precondition(s)
01232 if ( pc != NULL ) {
01233     pc->fct(s,sh,pc->data); /* Apply preconditioner */
01234 }
01235 else
01236     fasp_darray_cp(m,s,sh); /* No preconditioner */
01237
01238 // t = A*sh;
01239 fasp_blas_dstr_mxv(A,sh,t);
01240 // tt = t' * t;
01241 tt = fasp_blas_darray_dotprod(m,t,t);
01242 if ( (tt == 0) || (tt >= DBL_MAX) ) {
01243     flag = 4;
01244     goto FINISHED;
01245 }
01246
01247 // omega = (t' * s) / tt;
01248 omega = fasp_blas_darray_dotprod(m,s,t)/tt;
01249 if ( ABS(omega) > DBL_MAX ) {
01250     flag = 4;
01251     goto FINISHED;
01252 }
01253
01254 norm_sh = fasp_blas_darray_norm2(m,sh);
01255 norm_xhalf = fasp_blas_darray_norm2(m,xhalf);
01256
01257 if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
01258     stag = stag + 1;
01259 else
01260     stag = 0;
01261
01262 fasp_blas_darray_axpyz(m, omega,sh,xhalf, x); // x = xhalf + omega * sh;
01263 fasp_blas_darray_axpyz(m, -omega, t, s, r); // r = s - omega * t;
01264 normr = fasp_blas_darray_norm2(m,r); // normr = norm(r);
01265 normr_act = normr;
01266
01267 // check for convergence
01268 if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
01269 {
01270     fasp_darray_cp(m,bval,r);
01271     fasp_blas_dstr_aAxy(-1.0,A,x,r);
01272     normr_act = fasp_blas_darray_norm2(m,r);

```

```

01273         if ( normr_act <= tol_b ) {
01274             flag = 0;
01275             goto FINISHED;
01276         }
01277         else {
01278             if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
01279
01280             moresteps = moresteps + 1;
01281             if ( moresteps >= maxmsteps ) {
01282                 flag = 3;
01283                 goto FINISHED;
01284             }
01285         }
01286     }
01287
01288     // update minimal norm quantities
01289     if ( normr_act < normrmin ) {
01290         normrmin = normr_act;
01291         fasp_darray_cp(m,x,xmin);
01292         imin = iter;
01293     }
01294
01295     if ( stag >= maxstagsteps )
01296     {
01297         flag = 3;
01298         goto FINISHED;
01299     }
01300
01301     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01302
01303     absres0 = absres;
01304 } // for iter = 1 : maxit
01305
01306 FINISHED: // finish iterative method
01307 // returned solution is first with minimal residual
01308 if (flag == 0)
01309     relres = normr_act / n2b;
01310 else {
01311     fasp_darray_cp(m, bval,r);
01312     fasp_blas_dstr_aAxy(-1.0,A,xmin,r);
01313     normr = fasp_blas_darray_norm2(m,r);
01314
01315     if ( normr <= normr_act ) {
01316         fasp_darray_cp(m, xmin,x);
01317         iter = imin;
01318         relres = normr/n2b;
01319     }
01320     else {
01321         relres = normr_act/n2b;
01322     }
01323 }
01324
01325 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01326
01327 if ( PrtLvl >= PRINT_MORE )
01328     printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01329           flag,stag,imin,half_step);
01330
01331 // clean up temp memory
01332 fasp_mem_free(work); work = NULL;
01333
01334 #if DEBUG_MODE > 0
01335     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
01336 #endif
01337
01338 if ( iter > MaxIt )
01339     return ERROR_SOLVER_MAXIT;
01340 else
01341     return iter;
01342 }
01343
01365 INT fasp_solver_pbcgs (mxv_matfree *mf,
01366                       dvector      *b,
01367                       dvector      *u,
01368                       precondition *pc,
01369                       const REAL    tol,
01370                       const INT     MaxIt,
01371                       const SHORT   StopType,
01372                       const SHORT   PrtLvl)
01373 {
01374     const INT    m = b->row;

```

```

01375
01376 // local variables
01377 REAL    n2b,tolb;
01378 INT     iter=0, stag = 1, moresteps = 1, maxmsteps=1;
01379 INT     flag, maxstagsteps, half_step=0;
01380 REAL    absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
01381 REAL    alpha,beta,omega,rho,rhol,rtv,tt;
01382 REAL    normr,normr_act,normph,normx,imin;
01383 REAL    norm_sh,norm_xhalf,normrmin,factor;
01384 REAL    *x = u->val, *bval=b->val;
01385
01386 // allocate temp memory (need 10*m REAL)
01387 REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
01388 REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
01389 REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
01390 REAL *t = sh+m, *xmin = t+m;
01391
01392 // Output some info for debugging
01393 if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (MatFree) ...\n");
01394
01395 #if DEBUG_MODE > 0
01396 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01397 printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01398 #endif
01399
01400 // r = b-A*u
01401 fasp_darray_cp(m,bval,r);
01402 n2b = fasp_blas_darray_norm2(m,r);
01403
01404 flag = 1;
01405 fasp_darray_cp(m,x,xmin);
01406 imin = 0;
01407
01408 iter = 0;
01409
01410 tolb = n2b*tol;
01411
01412 // r = b-A*x
01413 mf->fct(mf->data, x, r);
01414 fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
01415 normr = fasp_blas_darray_norm2(m,r);
01416 normr_act = normr;
01417
01418 relres = normr/n2b;
01419 // if initial residual is small, no need to iterate!
01420 if (normr <= tolb) {
01421     flag =0;
01422     iter =0;
01423     goto FINISHED;
01424 }
01425
01426 // output iteration information if needed
01427
01428 fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);
01429
01430 // shadow residual rt = r* := r
01431 fasp_darray_cp(m,r,rt);
01432 normrmin = normr;
01433
01434 rho = 1.0;
01435 omega = 1.0;
01436 stag = 0;
01437 alpha =0.0;
01438
01439 moresteps = 0;
01440 maxmsteps = 10;
01441 maxstagsteps = 3;
01442
01443 // loop over maxit iterations (unless convergence or failure)
01444 for (iter=1;iter <= MaxIt;iter++) {
01445
01446     rhol = rho;
01447     rho = fasp_blas_darray_dotprod(m,rt,r);
01448
01449     if ((rho ==0.0) || (ABS(rho) >= DBL_MAX )) {
01450         flag = 4;
01451         goto FINISHED;
01452     }
01453
01454     if (iter==1) {
01455         fasp_darray_cp(m,r,p);

```

```

01456     }
01457     else {
01458         beta = (rho/rho1)*(alpha/omega);
01459
01460         if ((beta == 0) || ( ABS(beta) > DBL_MAX )) {
01461             flag = 4;
01462             goto FINISHED;
01463         }
01464
01465         // p = r + beta * (p - omega * v);
01466         fasp_blas_darray_axpy(m,-omega,v,p); //p=p - omega*v
01467         fasp_blas_darray_axpby(m,1.0, r, beta, p); //p = 1.0*r +beta*p
01468     }
01469
01470     // pp = precondition(p) ,ph
01471     if ( pc != NULL )
01472         pc->fct(p,ph,pc->data); /* Apply preconditioner */
01473     // if ph all is infinite then exit need add
01474     else
01475         fasp_darray_cp(m,p,ph); /* No preconditioner */
01476
01477     // v = A*ph
01478     mf->fct(mf->data, ph, v);
01479     rtv = fasp_blas_darray_dotprod(m,rt,v);
01480
01481     if (( rtv==0.0 ) || ( ABS(rtv) > DBL_MAX )) {
01482         flag = 4;
01483         goto FINISHED;
01484     }
01485
01486     alpha = rho/rtv;
01487
01488     if ( ABS(alpha) > DBL_MAX ) {
01489         flag = 4;
01490         ITS_DIVZERO;
01491         goto FINISHED;
01492     }
01493
01494     normx = fasp_blas_darray_norm2(m,x);
01495     normph = fasp_blas_darray_norm2(m,ph);
01496     if (ABS(alpha)*normph < DBL_EPSILON*normx )
01497         stag = stag + 1;
01498     else
01499         stag = 0;
01500
01501     // xhalf = x + alpha * ph; // form the "half" iterate
01502     // s = r - alpha * v; // residual associated with xhalf
01503     fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf); // z= ax + y
01504     fasp_blas_darray_axpyz(m, -alpha, v, r, s);
01505     normr = fasp_blas_darray_norm2(m,s); // normr = norm(s);
01506     normr_act = normr;
01507
01508     // compute reduction factor of residual ||r||
01509     absres = normr_act;
01510     factor = absres/absres0;
01511     fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);
01512
01513     // check for convergence
01514     if ((normr <= tol) || (stag >= maxstagsteps) || moresteps)
01515     {
01516         // s = b-A*xhalf
01517         mf->fct(mf->data, xhalf, s);
01518         fasp_blas_darray_axpby(m, 1.0, bval, -1.0, s);
01519         normr_act = fasp_blas_darray_norm2(m,s);
01520
01521         if (normr_act <= tol){
01522             // x = xhalf;
01523             fasp_darray_cp(m,xhalf,x); // x = xhalf;
01524             flag = 0;
01525             imin = iter - 0.5;
01526             half_step++;
01527             if ( PrtLvl >= PRINT_MORE )
01528                 printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01529                     flag,stag,imin,half_step);
01530             goto FINISHED;
01531         }
01532     }
01533     else {
01534         if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
01535
01536         moresteps = moresteps + 1;
01537         if (moresteps >= maxmsteps){

```



```

01537             //      if ~warned
01538             flag = 3;
01539             fasp_darray_cp(m,xhalf,x);
01540             goto FINISHED;
01541         }
01542     }
01543 }
01544
01545 if ( stag >= maxstagsteps ) {
01546     flag = 3;
01547     goto FINISHED;
01548 }
01549
01550 if ( normr_act < normrmin ) // update minimal norm quantities
01551 {
01552     normrmin = normr_act;
01553     fasp_darray_cp(m,xhalf,xmin);
01554     imin = iter - 0.5;
01555     half_step++;
01556     if ( PrtLvl >= PRINT_MORE )
01557         printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01558             flag,stag,imin,half_step);
01559 }
01560
01561 // sh = precondition(s)
01562 if ( pc != NULL ) {
01563     pc->fct(s,sh,pc->data); /* Apply preconditioner */
01564     //if all is finite
01565 }
01566 else
01567     fasp_darray_cp(m,s,sh); /* No preconditioner */
01568
01569 // t = A*sh;
01570 mf->fct(mf->data, sh, t);
01571 // tt = t' * t;
01572 tt = fasp_blas_darray_dotprod(m,t,t);
01573 if ((tt == 0) || (tt >= DBL_MAX)) {
01574     flag = 4;
01575     goto FINISHED;
01576 }
01577
01578 // omega = (t' * s) / tt;
01579 omega = fasp_blas_darray_dotprod(m,s,t)/tt;
01580 if (ABS(omega) > DBL_MAX )
01581 {
01582     flag = 4;
01583     goto FINISHED;
01584 }
01585
01586 norm_sh = fasp_blas_darray_norm2(m,sh);
01587 norm_xhalf = fasp_blas_darray_norm2(m,xhalf);
01588
01589 if (ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
01590     stag = stag + 1;
01591 else
01592     stag = 0;
01593
01594 fasp_blas_darray_axpyz(m, omega,sh,xhalf, x); // x = xhalf + omega * sh;
01595 fasp_blas_darray_axpyz(m, -omega, t, s, r); // r = s - omega * t;
01596 normr = fasp_blas_darray_norm2(m,r); //normr = norm(r);
01597 normr_act = normr;
01598
01599 // check for convergence
01600 if ( (normr <= tol) || (stag >= maxstagsteps) || moresteps )
01601 {
01602     // normr_act = norm(r);
01603     // r = b-A*x
01604     mf->fct(mf->data, x, r);
01605     fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
01606     normr_act = fasp_blas_darray_norm2(m,r);
01607     if (normr_act <= tol)
01608     {
01609         flag = 0;
01610         goto FINISHED;
01611     }
01612     else
01613     {
01614         if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;
01615
01616         moresteps = moresteps + 1;
01617         if (moresteps >= maxmsteps)

```

```

01618         {
01619             flag = 3;
01620             goto FINISHED;
01621         }
01622     }
01623 }
01624
01625 if (normr_act < normrmin) // update minimal norm quantities
01626 {
01627     normrmin = normr_act;
01628     fasp_darray_cp(m,x,xmin);
01629     imin = iter;
01630 }
01631
01632 if (stag >= maxstagsteps)
01633 {
01634     flag = 3;
01635     goto FINISHED;
01636 }
01637
01638 if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01639
01640 absres0 = absres;
01641 } // for iter = 1 : maxit
01642
01643 FINISHED: // finish iterative method
01644 // returned solution is first with minimal residual
01645 if (flag == 0)
01646     relres = normr_act / n2b;
01647 else {
01648     // r = b-A*xmin
01649     mf->fct(mf->data, xmin, r);
01650     fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
01651     normr = fasp_blas_darray_norm2(m,r);
01652
01653     if ( normr <= normr_act ) {
01654         fasp_darray_cp(m, xmin,x);
01655         iter = imin;
01656         relres = normr/n2b;
01657     }
01658     else {
01659         relres = normr_act/n2b;
01660     }
01661 }
01662
01663 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01664
01665 if ( PrtLvl >= PRINT_MORE )
01666     printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
01667         flag,stag,imin, half_step);
01668
01669 // clean up temp memory
01670 fasp_mem_free(work); work = NULL;
01671
01672 #if DEBUG_MODE > 0
01673     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01674 #endif
01675
01676 if ( iter > MaxIt )
01677     return ERROR_SOLVER_MAXIT;
01678 else
01679     return iter;
01680 }
01681
01682 /*-----*/
01683 /*--          End of File          --*/
01684 /*-----*/

```

9.111 KryPcg.c File Reference

Krylov subspace methods – Preconditioned CG.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- `INT fasp_solver_dcsr_pcg` (`dCSRmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
Preconditioned conjugate gradient method for solving $Au=b$.
- `INT fasp_solver_dbsr_pcg` (`dBSRmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
Preconditioned conjugate gradient method for solving $Au=b$.
- `INT fasp_solver_dblc_pcg` (`dBLCmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
Preconditioned conjugate gradient method for solving $Au=b$.
- `INT fasp_solver_dstr_pcg` (`dSTRmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
Preconditioned conjugate gradient method for solving $Au=b$.
- `INT fasp_solver_pcg` (`mxv_matfree *mf`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
Preconditioned conjugate gradient (CG) method for solving $Au=b$.

9.111.1 Detailed Description

Krylov subspace methods – Preconditioned CG.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

See [KrySPcg.c](#) for a safer version

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Abstract algorithm

PCG method to solve $A*x=b$ is to generate $\{x_k\}$ to approximate x

Step 0. Given A , b , x_0 , M

Step 1. Compute residual $r_0 = b - A*x_0$ and convergence check;

Step 2. Initialization $z_0 = M^{-1}*r_0$, $p_0 = z_0$;

Step 3. Main loop ...

FOR $k = 0:MaxIt$

- get step size $\alpha = f(r_k, z_k, p_k)$;
- update solution: $x_{k+1} = x_k + \alpha*p_k$;
- perform stagnation check;
- update residual: $r_{k+1} = r_k - \alpha*(A*p_k)$;
- perform residual check;
- obtain p_{k+1} using $\{p_0, p_1, \dots, p_k\}$;
- prepare for next iteration;

- print the result of k-th iteration; END FOR

Convergence check: $\text{norm}(r)/\text{norm}(b) < \text{tol}$

Stagnation check:

- IF $\text{norm}(\alpha * p_k)/\text{norm}(x_{k+1}) < \text{tol_stag}$
 1. compute $r = b - A * x_{k+1}$;
 2. convergence check;
 3. IF (not converged & restart_number < Max_Stag_Check) restart;
- END IF

Residual check:

- IF $\text{norm}(r_{k+1})/\text{norm}(b) < \text{tol}$
 1. compute the real residual $r = b - A * x_{k+1}$;
 2. convergence check;
 3. IF (not converged & restart_number < Max_Res_Check) restart;
- END IF

Definition in file [KryPcg.c](#).

9.111.2 Function Documentation

9.111.2.1 fasp_solver_dblc_pcg()

```

INT fasp_solver_dblc_pcg (
    dBLCMat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBLCMat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/24/2010

Modified by Chensong Zhang on 03/28/2013

Definition at line 684 of file [KryPcg.c](#).

9.111.2.2 fasp_solver_dbsr_pcg()

```
INT fasp_solver_dbsr_pcg (
    dBSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 390 of file [KryPcg.c](#).

9.111.2.3 fasp_solver_dcsr_pcg()

```

INT fasp_solver_dcsr_pcg (
    dCSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang, Xiaozhe Hu, Shiquan Zhang

Date

05/06/2010

Definition at line 98 of file [KryPcg.c](#).

9.111.2.4 fasp_solver_dstr_pcg()

```

INT fasp_solver_dstr_pcg (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

04/25/2010

Modified by Chensong Zhang on 03/28/2013

Definition at line [978](#) of file [KryPcg.c](#).

9.111.2.5 fasp_solver_pcg()

```

INT fasp_solver_pcg (
    mxv_matfree * mf,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient (CG) method for solving $Au=b$.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang, Xiaozhe Hu, Shiquan Zhang

Date

05/06/2010

Modified by Feiteng Huang on 09/19/2012: matrix free
Definition at line 1272 of file [KryPcg.c](#).

9.112 KryPcg.c

[Go to the documentation of this file.](#)

```

00001
00062 #include <math.h>
00063
00064 #include "fasp.h"
00065 #include "fasp_functs.h"
00066
00067 /*-----*/
00068 /*--  Declare Private Functions  --*/
00069 /*-----*/
00070
00071 #include "KryUtil.inl"
00072
00073 /*-----*/
00074 /*--      Public Functions      --*/
00075 /*-----*/
00076
00098 INT fasp_solver_dcsr_pcg (dCSRmat      *A,
00099                          dvector      *b,
00100                          dvector      *u,
00101                          precondition  *pc,
00102                          const REAL    tol,
00103                          const INT     MaxIt,
00104                          const SHORT   StopType,
00105                          const SHORT   PrtLvl)
00106 {
00107     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00108     const INT    m = b->row;
00109     const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
00110     const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
00111
00112     // local variables
00113     INT          iter = 0, stag = 1, more_step = 1;
00114     REAL          absres0 = BIGREAL, absres = BIGREAL;
00115     REAL          relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00116     REAL          reldiff, factor, normuin;
00117     REAL          alpha, beta, templ, temp2;
00118
00119     // allocate temp memory (need 4*m REAL numbers)
00120     REAL *work = (REAL *)fasp_mem_calloc(4*m, sizeof(REAL));
00121     REAL *p = work, *z = work+m, *r = z+m, *t = r+m;
00122
00123     // Output some info for debugging
00124     if (PrtLvl > PRINT_NONE) printf("\nCalling CG solver (CSR) ...\n");
00125
00126     #if DEBUG_MODE > 0
00127         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00128         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00129     #endif
00130
00131     // r = b-A*u
00132     fasp_darray_cp(m, b->val, r);
00133     fasp_blas_dcsr_aAxy(-1.0, A, u->val, r);
00134
00135     if (pc != NULL)
00136         pc->fct(r, z, pc->data); /* Apply preconditioner */

```



```

00137     else
00138         fasp_darray_cp(m,r,z); /* No preconditioner */
00139
00140     // compute initial residuals
00141     switch ( StopType ) {
00142     case STOP_REL_RES:
00143         absres0 = fasp_blas_darray_norm2(m,r);
00144         normr0  = MAX(SMALLREAL,absres0);
00145         relres  = absres0/normr0;
00146         break;
00147     case STOP_REL_PRECRES:
00148         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
00149         normr0  = MAX(SMALLREAL,absres0);
00150         relres  = absres0/normr0;
00151         break;
00152     case STOP_MOD_REL_RES:
00153         absres0 = fasp_blas_darray_norm2(m,r);
00154         normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00155         relres  = absres0/normu;
00156         break;
00157     default:
00158         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00159         goto FINISHED;
00160     }
00161
00162     // if initial residual is small, no need to iterate!
00163     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00164
00165     // output iteration information if needed
00166     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00167
00168     fasp_darray_cp(m,z,p);
00169     temp1 = fasp_blas_darray_dotprod(m,z,r);
00170
00171     // main PCG loop
00172     while ( iter++ < MaxIt ) {
00173
00174         // t = A*p
00175         fasp_blas_dcsr_mxv(A,p,t);
00176
00177         // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00178         temp2 = fasp_blas_darray_dotprod(m,t,p);
00179         if ( ABS(temp2) > SMALLREAL2 ) {
00180             alpha = temp1/temp2;
00181         }
00182         else { // Possible breakdown
00183             ITS_DIVZERO; goto FINISHED;
00184         }
00185
00186         // u_k = u_{k-1} + alpha_k*p_{k-1}
00187         fasp_blas_darray_axpy(m,alpha,p,u->val);
00188
00189         // r_k = r_{k-1} - alpha_k*A*p_{k-1}
00190         fasp_blas_darray_axpy(m,-alpha,t,r);
00191
00192         // compute norm of residual
00193         switch ( StopType ) {
00194         case STOP_REL_RES:
00195             absres = fasp_blas_darray_norm2(m,r);
00196             relres = absres/normr0;
00197             break;
00198         case STOP_REL_PRECRES:
00199             // z = B(r)
00200             if ( pc != NULL )
00201                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00202             else
00203                 fasp_darray_cp(m,r,z); /* No preconditioner */
00204             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00205             relres = absres/normr0;
00206             break;
00207         case STOP_MOD_REL_RES:
00208             absres = fasp_blas_darray_norm2(m,r);
00209             relres = absres/normu;
00210             break;
00211         }
00212
00213         // compute reduction factor of residual ||r||
00214         factor = absres/absres0;
00215
00216         // output iteration information if needed
00217         fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

```

```

00218
00219     if ( factor > 0.9 ) { // Only check when converge slowly
00220
00221         // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00222         normuinf = fasp_blas_darray_norminf(m, u->val);
00223         if ( normuinf <= sol_inf_tol ) {
00224             if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00225             iter = ERROR_SOLVER_SOLSTAG;
00226             break;
00227         }
00228
00229         // Check II: if stagnated, try to restart
00230         normu = fasp_blas_darray_norm2(m, u->val);
00231
00232         // compute relative difference
00233         reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00234         if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00235
00236             if ( PrtLvl >= PRINT_MORE ) {
00237                 ITS_DIFFRES(reldiff, relres);
00238                 ITS_RESTART;
00239             }
00240
00241             fasp_darray_cp(m,b->val,r);
00242             fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00243
00244             // compute residual norms
00245             switch ( StopType ) {
00246                 case STOP_REL_RES:
00247                     absres = fasp_blas_darray_norm2(m,r);
00248                     relres = absres/normr0;
00249                     break;
00250                 case STOP_REL_PRECRES:
00251                     // z = B(r)
00252                     if ( pc != NULL )
00253                         pc->fct(r,z,pc->data); /* Apply preconditioner */
00254                     else
00255                         fasp_darray_cp(m,r,z); /* No preconditioner */
00256                     absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00257                     relres = absres/normr0;
00258                     break;
00259                 case STOP_MOD_REL_RES:
00260                     absres = fasp_blas_darray_norm2(m,r);
00261                     relres = absres/normu;
00262                     break;
00263             }
00264
00265             if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00266
00267             if ( relres < tol )
00268                 break;
00269             else {
00270                 if ( stag >= MaxStag ) {
00271                     if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00272                     iter = ERROR_SOLVER_STAG;
00273                     break;
00274                 }
00275                 fasp_darray_set(m,p,0.0);
00276                 ++stag;
00277             }
00278
00279             } // end of stagnation check!
00280
00281     } // end of check I and II
00282
00283     // Check III: prevent false convergence
00284     if ( relres < tol ) {
00285
00286         REAL updated_relres = relres;
00287
00288         // compute true residual r = b - Ax and update residual
00289         fasp_darray_cp(m,b->val,r);
00290         fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00291
00292         // compute residual norms
00293         switch ( StopType ) {
00294             case STOP_REL_RES:
00295                 absres = fasp_blas_darray_norm2(m,r);
00296                 relres = absres/normr0;
00297                 break;
00298             case STOP_REL_PRECRES:

```

```

00299         // z = B(r)
00300         if ( pc != NULL )
00301             pc->fct(r,z,pc->data); /* Apply preconditioner */
00302         else
00303             fasp_darray_cp(m,r,z); /* No preconditioner */
00304         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00305         relres = absres/normr0;
00306         break;
00307     case STOP_MOD_REL_RES:
00308         absres = fasp_blas_darray_norm2(m,r);
00309         relres = absres/normu;
00310         break;
00311     }
00312
00313     // check convergence
00314     if ( relres < tol ) break;
00315
00316     if ( PrtLvl >= PRINT_MORE ) {
00317         ITS_COMPRES(updated_relres); ITS_REALRES(relres);
00318     }
00319
00320     if ( more_step >= MaxRestartStep ) {
00321         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00322         iter = ERROR_SOLVER_TOLSMALL;
00323         break;
00324     }
00325
00326     // prepare for restarting method
00327     fasp_darray_set(m,p,0.0);
00328     ++more_step;
00329 } // end of safe-guard check!
00330
00331 // save residual for next iteration
00332 absres0 = absres;
00333
00334 // compute z_k = B(r_k)
00335 if ( StopType != STOP_REL_PRECRES ) {
00336     if ( pc != NULL )
00337         pc->fct(r,z,pc->data); /* Apply preconditioner */
00338     else
00339         fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00340 }
00341
00342 // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
00343 temp2 = fasp_blas_darray_dotprod(m,z,r);
00344 beta = temp2/temp1;
00345 temp1 = temp2;
00346
00347 // compute p_k = z_k + beta_k*p_{k-1}
00348 fasp_blas_darray_axpby(m,1.0,z,beta,p);
00349
00350 } // end of main PCG loop.
00351
00352 FINISHED: // finish iterative method
00353 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00354
00355 // clean up temp memory
00356 fasp_mem_free(work); work = NULL;
00357
00358 #if DEBUG_MODE > 0
00359     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00360 #endif
00361
00362 if ( iter > MaxIt )
00363     return ERROR_SOLVER_MAXIT;
00364 else
00365     return iter;
00366 }
00367
00368
00390 INT fasp_solver_dbsr_pcg (dBSRmat      *A,
00391                          dvector      *b,
00392                          dvector      *u,
00393                          precondition *pc,
00394                          const REAL   tol,
00395                          const INT    MaxIt,
00396                          const SHORT  StopType,
00397                          const SHORT  PrtLvl)
00398 {
00399     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00400     const INT   m = b->row;

```

```

00401     const REAL    maxdiff = tol*STAG_RATIO; // stagnation tolerance
00402     const REAL    sol_inf_tol = SMALLREAL; // infinity norm tolerance
00403
00404     // local variables
00405     INT           iter = 0, stag = 1, more_step = 1;
00406     REAL          absres0 = BIGREAL, absres = BIGREAL;
00407     REAL          relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00408     REAL          reldiff, factor, normuinf;
00409     REAL          alpha, beta, temp1, temp2;
00410
00411     // allocate temp memory (need 4*m REAL numbers)
00412     REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
00413     REAL *p = work, *z = work+m, *r = z+m, *t = r+m;
00414
00415     // Output some info for debugging
00416     if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (BSR) ...\n");
00417
00418     #if DEBUG_MODE > 0
00419     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00420     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00421     #endif
00422
00423     // r = b-A*u
00424     fasp_darray_cp(m,b->val,r);
00425     fasp_blas_dbsr_aAxy(-1.0,A,u->val,r);
00426
00427     if ( pc != NULL )
00428         pc->fct(r,z,pc->data); /* Apply preconditioner */
00429     else
00430         fasp_darray_cp(m,r,z); /* No preconditioner */
00431
00432     // compute initial residuals
00433     switch ( StopType ) {
00434     case STOP_REL_RES:
00435         absres0 = fasp_blas_darray_norm2(m,r);
00436         normr0 = MAX(SMALLREAL,absres0);
00437         relres = absres0/normr0;
00438         break;
00439     case STOP_REL_PRECRES:
00440         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
00441         normr0 = MAX(SMALLREAL,absres0);
00442         relres = absres0/normr0;
00443         break;
00444     case STOP_MOD_REL_RES:
00445         absres0 = fasp_blas_darray_norm2(m,r);
00446         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00447         relres = absres0/normu;
00448         break;
00449     default:
00450         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00451         goto FINISHED;
00452     }
00453
00454     // if initial residual is small, no need to iterate!
00455     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00456
00457     // output iteration information if needed
00458     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00459
00460     fasp_darray_cp(m,z,p);
00461     temp1 = fasp_blas_darray_dotprod(m,z,r);
00462
00463     // main PCG loop
00464     while ( iter++ < MaxIt ) {
00465
00466         // t = A*p
00467         fasp_blas_dbsr_mxv(A,p,t);
00468
00469         // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00470         temp2 = fasp_blas_darray_dotprod(m,t,p);
00471         if ( ABS(temp2) > SMALLREAL2 ) {
00472             alpha = temp1/temp2;
00473         }
00474         else { // Possible breakdown
00475             ITS_DIVZERO; goto FINISHED;
00476         }
00477
00478         // u_k = u_{k-1} + alpha_k*p_{k-1}
00479         fasp_blas_darray_axpy(m,alpha,p,u->val);
00480
00481         // r_k = r_{k-1} - alpha_k*A*p_{k-1}

```

```

00482     fasp_blas_darray_axpy(m,-alpha,t,r);
00483
00484     // compute norm of residual
00485     switch ( StopType ) {
00486     case STOP_REL_RES:
00487         absres = fasp_blas_darray_norm2(m,r);
00488         relres = absres/normr0;
00489         break;
00490     case STOP_REL_PRECRES:
00491         // z = B(r)
00492         if ( pc != NULL )
00493             pc->fct(r,z,pc->data); /* Apply preconditioner */
00494         else
00495             fasp_darray_cp(m,r,z); /* No preconditioner */
00496         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00497         relres = absres/normr0;
00498         break;
00499     case STOP_MOD_REL_RES:
00500         absres = fasp_blas_darray_norm2(m,r);
00501         relres = absres/normu;
00502         break;
00503     }
00504
00505     // compute reduction factor of residual ||r||
00506     factor = absres/absres0;
00507
00508     // output iteration information if needed
00509     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00510
00511     if ( factor > 0.9 ) { // Only check when converge slowly
00512
00513         // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00514         normuinf = fasp_blas_darray_norminf(m, u->val);
00515         if ( normuinf <= sol_inf_tol ) {
00516             if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00517             iter = ERROR_SOLVER_SOLSTAG;
00518             break;
00519         }
00520
00521         // Check II: if stagnated, try to restart
00522         normu = fasp_blas_darray_norm2(m, u->val);
00523
00524         // compute relative difference
00525         reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00526         if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00527
00528             if ( PrtLvl >= PRINT_MORE ) {
00529                 ITS_DIFFRES(reldiff,relres);
00530                 ITS_RESTART;
00531             }
00532
00533             fasp_darray_cp(m,b->val,r);
00534             fasp_blas_dbsr_aAxy(-1.0,A,u->val,r);
00535
00536             // compute residual norms
00537             switch ( StopType ) {
00538             case STOP_REL_RES:
00539                 absres = fasp_blas_darray_norm2(m,r);
00540                 relres = absres/normr0;
00541                 break;
00542             case STOP_REL_PRECRES:
00543                 // z = B(r)
00544                 if ( pc != NULL )
00545                     pc->fct(r,z,pc->data); /* Apply preconditioner */
00546                 else
00547                     fasp_darray_cp(m,r,z); /* No preconditioner */
00548                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00549                 relres = absres/normr0;
00550                 break;
00551             case STOP_MOD_REL_RES:
00552                 absres = fasp_blas_darray_norm2(m,r);
00553                 relres = absres/normu;
00554                 break;
00555             }
00556
00557             if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00558
00559             if ( relres < tol )
00560                 break;
00561             else {
00562                 if ( stag >= MaxStag ) {

```

```

00563         if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00564         iter = ERROR_SOLVER_STAG;
00565         break;
00566     }
00567     fasp_darray_set(m,p,0.0);
00568     ++stag;
00569 }
00570
00571 } // end of stagnation check!
00572
00573 } // end of check I and II
00574
00575 // Check III: prevent false convergence
00576 if ( relres < tol ) {
00577     REAL updated_relres = relres;
00578
00579     // compute true residual r = b - Ax and update residual
00580     fasp_darray_cp(m,b->val,r);
00581     fasp_blas_dbsr_aAxy(-1.0,A,u->val,r);
00582
00583     // compute residual norms
00584     switch ( StopType ) {
00585     case STOP_REL_RES:
00586         absres = fasp_blas_darray_norm2(m,r);
00587         relres = absres/normr0;
00588         break;
00589     case STOP_REL_PRECRES:
00590         // z = B(r)
00591         if ( pc != NULL )
00592             pc->fct(r,z,pc->data); /* Apply preconditioner */
00593         else
00594             fasp_darray_cp(m,r,z); /* No preconditioner */
00595         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00596         relres = absres/normr0;
00597         break;
00598     case STOP_MOD_REL_RES:
00599         absres = fasp_blas_darray_norm2(m,r);
00600         relres = absres/normu;
00601         break;
00602     }
00603
00604     // check convergence
00605     if ( relres < tol ) break;
00606
00607     if ( PrtLvl >= PRINT_MORE ) {
00608         ITS_COMPRES(updated_relres); ITS_REALRES(relres);
00609     }
00610
00611     if ( more_step >= MaxRestartStep ) {
00612         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00613         iter = ERROR_SOLVER_TOLSMALL;
00614         break;
00615     }
00616
00617     // prepare for restarting method
00618     fasp_darray_set(m,p,0.0);
00619     ++more_step;
00620
00621 } // end of safe-guard check!
00622
00623 // save residual for next iteration
00624 absres0 = absres;
00625
00626 // compute z_k = B(r_k)
00627 if ( StopType != STOP_REL_PRECRES ) {
00628     if ( pc != NULL )
00629         pc->fct(r,z,pc->data); /* Apply preconditioner */
00630     else
00631         fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00632 }
00633
00634 // compute beta_k = (z_k, r_k) / (z_{k-1}, r_{k-1})
00635 temp2 = fasp_blas_darray_dotprod(m,z,r);
00636 beta = temp2/temp1;
00637 temp1 = temp2;
00638
00639 // compute p_k = z_k + beta_k*p_{k-1}
00640 fasp_blas_darray_axpy(m,1.0,z,beta,p);
00641
00642 } // end of main PCG loop.
00643

```

```

00644
00645 FINISHED: // finish iterative method
00646     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00647
00648     // clean up temp memory
00649     fasp_mem_free(work); work = NULL;
00650
00651 #if DEBUG_MODE > 0
00652     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00653 #endif
00654
00655     if ( iter > MaxIt )
00656         return ERROR_SOLVER_MAXIT;
00657     else
00658         return iter;
00659 }
00660
00684 INT fasp_solver_dblc_pcg (dBLCmat      *A,
00685                          dvector      *b,
00686                          dvector      *u,
00687                          precondition *pc,
00688                          const REAL    tol,
00689                          const INT      MaxIt,
00690                          const SHORT    StopType,
00691                          const SHORT    PrtLvl)
00692 {
00693     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00694     const INT    m = b->row;
00695     const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
00696     const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
00697
00698     // local variables
00699     INT          iter = 0, stag = 1, more_step = 1;
00700     REAL          absres0 = BIGREAL, absres = BIGREAL;
00701     REAL          relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00702     REAL          reldiff, factor, normuinf;
00703     REAL          alpha, beta, temp1, temp2;
00704
00705     // allocate temp memory (need 4*m REAL numbers)
00706     REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
00707     REAL *p = work, *z = work+m, *r = z+m, *t = r+m;
00708
00709     // Output some info for debugging
00710     if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (BLC) ...\n");
00711
00712 #if DEBUG_MODE > 0
00713     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00714     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00715 #endif
00716
00717     // r = b-A*u
00718     fasp_darray_cp(m,b->val,r);
00719     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00720
00721     if ( pc != NULL )
00722         pc->fct(r,z,pc->data); /* Apply preconditioner */
00723     else
00724         fasp_darray_cp(m,r,z); /* No preconditioner */
00725
00726     // compute initial residuals
00727     switch ( StopType ) {
00728     case STOP_REL_RES:
00729         absres0 = fasp_blas_darray_norm2(m,r);
00730         normr0 = MAX(SMALLREAL,absres0);
00731         relres = absres0/normr0;
00732         break;
00733     case STOP_REL_PRECRES:
00734         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
00735         normr0 = MAX(SMALLREAL,absres0);
00736         relres = absres0/normr0;
00737         break;
00738     case STOP_MOD_REL_RES:
00739         absres0 = fasp_blas_darray_norm2(m,r);
00740         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00741         relres = absres0/normu;
00742         break;
00743     default:
00744         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00745         goto FINISHED;
00746     }
00747

```

```

00748 // if initial residual is small, no need to iterate!
00749 if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00750
00751 // output iteration information if needed
00752 fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00753
00754 fasp_darray_cp(m,z,p);
00755 temp1 = fasp_blas_darray_dotprod(m,z,r);
00756
00757 // main PCG loop
00758 while ( iter++ < MaxIt ) {
00759
00760     // t = A*p
00761     fasp_blas_dblc_mxv(A,p,t);
00762
00763     // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00764     temp2 = fasp_blas_darray_dotprod(m,t,p);
00765     if ( ABS(temp2) > SMALLREAL2 ) {
00766         alpha = temp1/temp2;
00767     }
00768     else { // Possible breakdown
00769         ITS_DIVZERO; goto FINISHED;
00770     }
00771
00772     // u_k = u_{k-1} + alpha_k*p_{k-1}
00773     fasp_blas_darray_axpy(m,alpha,p,u->val);
00774
00775     // r_k = r_{k-1} - alpha_k*A*p_{k-1}
00776     fasp_blas_darray_axpy(m,-alpha,t,r);
00777
00778     // compute norm of residual
00779     switch ( StopType ) {
00780         case STOP_REL_RES:
00781             absres = fasp_blas_darray_norm2(m,r);
00782             relres = absres/normr0;
00783             break;
00784         case STOP_REL_PRECRES:
00785             // z = B(r)
00786             if ( pc != NULL )
00787                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00788             else
00789                 fasp_darray_cp(m,r,z); /* No preconditioner */
00790             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00791             relres = absres/normr0;
00792             break;
00793         case STOP_MOD_REL_RES:
00794             absres = fasp_blas_darray_norm2(m,r);
00795             relres = absres/normu;
00796             break;
00797     }
00798
00799     // compute reduction factor of residual ||r||
00800     factor = absres/absres0;
00801
00802     // output iteration information if needed
00803     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00804
00805     if ( factor > 0.9 ) { // Only check when converge slowly
00806
00807         // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00808         normuinf = fasp_blas_darray_norminf(m, u->val);
00809         if ( normuinf <= sol_inf_tol ) {
00810             if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00811             iter = ERROR_SOLVER_SOLSTAG;
00812             break;
00813         }
00814
00815         // Check II: if stagnated, try to restart
00816         normu = fasp_blas_darray_norm2(m, u->val);
00817
00818         // compute relative difference
00819         reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00820         if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00821
00822             if ( PrtLvl >= PRINT_MORE ) {
00823                 ITS_DIFFRES(reldiff,relres);
00824                 ITS_RESTART;
00825             }
00826
00827             fasp_darray_cp(m,b->val,r);
00828             fasp_blas_dblc_aAxy(-1.0,A,u->val,r);

```



```

00829
00830 // compute residual norms
00831 switch ( StopType ) {
00832     case STOP_REL_RES:
00833         absres = fasp_blas_darray_norm2(m,r);
00834         relres = absres/normr0;
00835         break;
00836     case STOP_REL_PRECRES:
00837         // z = B(r)
00838         if ( pc != NULL )
00839             pc->fct(r,z,pc->data); /* Apply preconditioner */
00840         else
00841             fasp_darray_cp(m,r,z); /* No preconditioner */
00842         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00843         relres = absres/normr0;
00844         break;
00845     case STOP_MOD_REL_RES:
00846         absres = fasp_blas_darray_norm2(m,r);
00847         relres = absres/normu;
00848         break;
00849 }
00850
00851 if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00852
00853 if ( relres < tol )
00854     break;
00855 else {
00856     if ( stag >= MaxStag ) {
00857         if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00858         iter = ERROR_SOLVER_STAG;
00859         break;
00860     }
00861     fasp_darray_set(m,p,0.0);
00862     ++stag;
00863 }
00864
00865 } // end of stagnation check!
00866
00867 } // end of check I and II
00868
00869 // Check III: prevent false convergence
00870 if ( relres < tol ) {
00871
00872     REAL updated_relres = relres;
00873
00874     // compute true residual r = b - Ax and update residual
00875     fasp_darray_cp(m,b->val,r);
00876     fasp_blas_dblc_aApy(-1.0,A,u->val,r);
00877
00878     // compute residual norms
00879     switch ( StopType ) {
00880         case STOP_REL_RES:
00881             absres = fasp_blas_darray_norm2(m,r);
00882             relres = absres/normr0;
00883             break;
00884         case STOP_REL_PRECRES:
00885             // z = B(r)
00886             if ( pc != NULL )
00887                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00888             else
00889                 fasp_darray_cp(m,r,z); /* No preconditioner */
00890             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00891             relres = absres/normr0;
00892             break;
00893         case STOP_MOD_REL_RES:
00894             absres = fasp_blas_darray_norm2(m,r);
00895             relres = absres/normu;
00896             break;
00897     }
00898
00899     // check convergence
00900     if ( relres < tol ) break;
00901
00902     if ( PrtLvl >= PRINT_MORE ) {
00903         ITS_COMPRES(updated_relres); ITS_REALRES(relres);
00904     }
00905
00906     if ( more_step >= MaxRestartStep ) {
00907         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00908         iter = ERROR_SOLVER_TOLSMALL;
00909         break;

```

```

00910         }
00911
00912         // prepare for restarting method
00913         fasp_darray_set(m,p,0.0);
00914         ++more_step;
00915
00916     } // end of safe-guard check!
00917
00918     // save residual for next iteration
00919     absres0 = absres;
00920
00921     // compute z_k = B(r_k)
00922     if ( StopType != STOP_REL_PRECRES ) {
00923         if ( pc != NULL )
00924             pc->fct(r,z,pc->data); /* Apply preconditioner */
00925         else
00926             fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00927     }
00928
00929     // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
00930     temp2 = fasp_blas_darray_dotprod(m,z,r);
00931     beta = temp2/temp1;
00932     temp1 = temp2;
00933
00934     // compute p_k = z_k + beta_k*p_{k-1}
00935     fasp_blas_darray_axpy(m,1.0,z,beta,p);
00936
00937 } // end of main PCG loop.
00938
00939 FINISHED: // finish iterative method
00940 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00941
00942 // clean up temp memory
00943 fasp_mem_free(work); work = NULL;
00944
00945 #if DEBUG_MODE > 0
00946 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00947 #endif
00948
00949 if ( iter > MaxIt )
00950     return ERROR_SOLVER_MAXIT;
00951 else
00952     return iter;
00953 }
00954
00978 INT fasp_solver_dstr_pcg (dSTRmat      *A,
00979                          dvector      *b,
00980                          dvector      *u,
00981                          precondition *pc,
00982                          const REAL   tol,
00983                          const INT    MaxIt,
00984                          const SHORT  StopType,
00985                          const SHORT  PrtLvl)
00986 {
00987     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00988     const INT   m = b->row;
00989     const REAL  maxdiff = tol*STAG_RATIO; // stagnation tolerance
00990     const REAL  sol_inf_tol = SMALLREAL; // infinity norm tolerance
00991
00992     // local variables
00993     INT         iter = 0, stag = 1, more_step = 1;
00994     REAL        absres0 = BIGREAL, absres = BIGREAL;
00995     REAL        relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00996     REAL        reldiff, factor, normuinf;
00997     REAL        alpha, beta, temp1, temp2;
00998
00999     // allocate temp memory (need 4*m REAL numbers)
01000     REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
01001     REAL *p = work, *z = work+m, *r = z+m, *t = r+m;
01002
01003     // Output some info for debugging
01004     if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (STR) ...\n");
01005
01006     #if DEBUG_MODE > 0
01007     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01008     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01009     #endif
01010
01011     // r = b-A*u
01012     fasp_darray_cp(m,b->val,r);
01013     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);

```

```

01014
01015     if ( pc != NULL )
01016         pc->fct(r,z,pc->data); /* Apply preconditioner */
01017     else
01018         fasp_darray_cp(m,r,z); /* No preconditioner */
01019
01020     // compute initial residuals
01021     switch ( StopType ) {
01022     case STOP_REL_RES:
01023         absres0 = fasp_blas_darray_norm2(m,r);
01024         normr0 = MAX(SMALLREAL,absres0);
01025         relres = absres0/normr0;
01026         break;
01027     case STOP_REL_PRECRES:
01028         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
01029         normr0 = MAX(SMALLREAL,absres0);
01030         relres = absres0/normr0;
01031         break;
01032     case STOP_MOD_REL_RES:
01033         absres0 = fasp_blas_darray_norm2(m,r);
01034         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
01035         relres = absres0/normu;
01036         break;
01037     default:
01038         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01039         goto FINISHED;
01040     }
01041
01042     // if initial residual is small, no need to iterate!
01043     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
01044
01045     // output iteration information if needed
01046     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
01047
01048     fasp_darray_cp(m,z,p);
01049     temp1 = fasp_blas_darray_dotprod(m,z,r);
01050
01051     // main PCG loop
01052     while ( iter++ < MaxIt ) {
01053
01054         // t = A*p
01055         fasp_blas_dstr_mxv(A,p,t);
01056
01057         // alpha_k = (z_{k-1}, r_{k-1}) / (A*p_{k-1}, p_{k-1})
01058         temp2 = fasp_blas_darray_dotprod(m,t,p);
01059         if ( ABS(temp2) > SMALLREAL2 ) {
01060             alpha = temp1/temp2;
01061         }
01062         else { // Possible breakdown
01063             ITS_DIVZERO; goto FINISHED;
01064         }
01065
01066         // u_k = u_{k-1} + alpha_k*p_{k-1}
01067         fasp_blas_darray_axpy(m,alpha,p,u->val);
01068
01069         // r_k = r_{k-1} - alpha_k*A*p_{k-1}
01070         fasp_blas_darray_axpy(m,-alpha,t,r);
01071
01072         // compute norm of residual
01073         switch ( StopType ) {
01074         case STOP_REL_RES:
01075             absres = fasp_blas_darray_norm2(m,r);
01076             relres = absres/normr0;
01077             break;
01078         case STOP_REL_PRECRES:
01079             // z = B(r)
01080             if ( pc != NULL )
01081                 pc->fct(r,z,pc->data); /* Apply preconditioner */
01082             else
01083                 fasp_darray_cp(m,r,z); /* No preconditioner */
01084             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01085             relres = absres/normr0;
01086             break;
01087         case STOP_MOD_REL_RES:
01088             absres = fasp_blas_darray_norm2(m,r);
01089             relres = absres/normu;
01090             break;
01091         }
01092
01093         // compute reduction factor of residual ||r||
01094         factor = absres/absres0;

```

```

01095
01096 // output iteration information if needed
01097 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01098
01099 if ( factor > 0.9 ) { // Only check when converge slowly
01100
01101     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
01102     normuinf = fasp_blas_darray_norminf(m, u->val);
01103     if ( normuinf <= sol_inf_tol ) {
01104         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01105         iter = ERROR_SOLVER_SOLSTAG;
01106         break;
01107     }
01108
01109     // Check II: if stagnated, try to restart
01110     normu = fasp_blas_darray_norm2(m, u->val);
01111
01112     // compute relative difference
01113     reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
01114     if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
01115
01116         if ( PrtLvl >= PRINT_MORE ) {
01117             ITS_DIFFRES(reldiff,relres);
01118             ITS_RESTART;
01119         }
01120
01121         fasp_darray_cp(m,b->val,r);
01122         fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01123
01124         // compute residual norms
01125         switch ( StopType ) {
01126             case STOP_REL_RES:
01127                 absres = fasp_blas_darray_norm2(m,r);
01128                 relres = absres/normr0;
01129                 break;
01130             case STOP_REL_PRECRES:
01131                 // z = B(r)
01132                 if ( pc != NULL )
01133                     pc->fct(r,z,pc->data); /* Apply preconditioner */
01134                 else
01135                     fasp_darray_cp(m,r,z); /* No preconditioner */
01136                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01137                 relres = absres/normr0;
01138                 break;
01139             case STOP_MOD_REL_RES:
01140                 absres = fasp_blas_darray_norm2(m,r);
01141                 relres = absres/normu;
01142                 break;
01143         }
01144
01145         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01146
01147         if ( relres < tol )
01148             break;
01149         else {
01150             if ( stag >= MaxStag ) {
01151                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01152                 iter = ERROR_SOLVER_STAG;
01153                 break;
01154             }
01155             fasp_darray_set(m,p,0.0);
01156             ++stag;
01157         }
01158     } // end of stagnation check!
01159 } // end of check I and II
01160
01161 // Check III: prevent false convergence
01162 if ( relres < tol ) {
01163     REAL updated_relres = relres;
01164
01165     // compute true residual r = b - Ax and update residual
01166     fasp_darray_cp(m,b->val,r);
01167     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01168
01169     // compute residual norms
01170     switch ( StopType ) {
01171         case STOP_REL_RES:
01172             absres = fasp_blas_darray_norm2(m,r);

```

```

01176         relres = absres/normr0;
01177         break;
01178     case STOP_REL_PRECRES:
01179         // z = B(r)
01180         if ( pc != NULL )
01181             pc->fct(r,z,pc->data); /* Apply preconditioner */
01182         else
01183             fasp_darray_cp(m,r,z); /* No preconditioner */
01184         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01185         relres = absres/normr0;
01186         break;
01187     case STOP_MOD_REL_RES:
01188         absres = fasp_blas_darray_norm2(m,r);
01189         relres = absres/normu;
01190         break;
01191     }
01192
01193     // check convergence
01194     if ( relres < tol ) break;
01195
01196     if ( PrtLvl >= PRINT_MORE ) {
01197         ITS_COMPRES(updated_relres); ITS_REALRES(relres);
01198     }
01199
01200     if ( more_step >= MaxRestartStep ) {
01201         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01202         iter = ERROR_SOLVER_TOLSMALL;
01203         break;
01204     }
01205
01206     // prepare for restarting method
01207     fasp_darray_set(m,p,0.0);
01208     ++more_step;
01209
01210 } // end of safe-guard check!
01211
01212 // save residual for next iteration
01213 absres0 = absres;
01214
01215 // compute z_k = B(r_k)
01216 if ( StopType != STOP_REL_PRECRES ) {
01217     if ( pc != NULL )
01218         pc->fct(r,z,pc->data); /* Apply preconditioner */
01219     else
01220         fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
01221 }
01222
01223 // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
01224 temp2 = fasp_blas_darray_dotprod(m,z,r);
01225 beta = temp2/temp1;
01226 temp1 = temp2;
01227
01228 // compute p_k = z_k + beta_k*p_{k-1}
01229 fasp_blas_darray_axpby(m,1.0,z,beta,p);
01230
01231 } // end of main PCG loop.
01232
01233 FINISHED: // finish iterative method
01234 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01235
01236 // clean up temp memory
01237 fasp_mem_free(work); work = NULL;
01238
01239 #if DEBUG_MODE > 0
01240 printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01241 #endif
01242
01243 if ( iter > MaxIt )
01244     return ERROR_SOLVER_MAXIT;
01245 else
01246     return iter;
01247 }
01248
01272 INT fasp_solver_pcg (mxv_matfree *mf,
01273                     dvector *b,
01274                     dvector *u,
01275                     precondition *pc,
01276                     const REAL tol,
01277                     const INT MaxIt,
01278                     const SHORT StopType,
01279                     const SHORT PrtLvl)

```

```

01280 {
01281     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
01282     const INT m=b->row;
01283     const REAL maxdiff = tol*STAG_RATIO; // stagantation tolerance
01284     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
01285
01286     // local variables
01287     INT iter = 0, stag, more_step, restart_step;
01288     REAL absres0 = BIGREAL, absres = BIGREAL;
01289     REAL relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
01290     REAL reldiff, factor, infnormu;
01291     REAL alpha, beta, templ, temp2;
01292
01293     // allocate temp memory (need 4*m REAL numbers)
01294     REAL *work=(REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
01295     REAL *p=work, *z=work+m, *r=z+m, *t=r+m;
01296
01297     // Output some info for debugging
01298     if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (MatFree) ...\n");
01299
01300 #if DEBUG_MODE > 0
01301     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01302     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01303 #endif
01304
01305     // initialize counters
01306     stag=1; more_step=1; restart_step=1;
01307
01308     // r = b-A*u
01309     mf->fct(mf->data, u->val, r);
01310     fasp_blas_darray_axpy(m, 1.0, b->val, -1.0, r);
01311
01312     if (pc != NULL)
01313         pc->fct(r,z,pc->data); /* Apply preconditioner */
01314     else
01315         fasp_darray_cp(m,r,z); /* No preconditioner */
01316
01317     // compute initial relative residual
01318     switch (StopType) {
01319     case STOP_REL_PRECRES:
01320         absres0=sqrt(fasp_blas_darray_dotprod(m,r,z));
01321         normr0=MAX(SMALLREAL,absres0);
01322         relres=absres0/normr0;
01323         break;
01324     case STOP_MOD_REL_RES:
01325         absres0=fasp_blas_darray_norm2(m,r);
01326         normu=MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
01327         relres=absres0/normu;
01328         break;
01329     default:
01330         absres0=fasp_blas_darray_norm2(m,r);
01331         normr0=MAX(SMALLREAL,absres0);
01332         relres=absres0/normr0;
01333         break;
01334     }
01335
01336     // if initial residual is small, no need to iterate!
01337     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
01338
01339     fasp_darray_cp(m,z,p);
01340     templ=fasp_blas_darray_dotprod(m,z,r);
01341
01342     while ( iter++ < MaxIt ) {
01343
01344         // t=A*p
01345         mf->fct(mf->data, p, t);
01346
01347         // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
01348         temp2=fasp_blas_darray_dotprod(m,t,p);
01349         alpha=templ/temp2;
01350
01351         // u_k=u_{k-1} + alpha_k*p_{k-1}
01352         fasp_blas_darray_axpy(m,alpha,p,u->val);
01353
01354         // r_k=r_{k-1} - alpha_k*A*p_{k-1}
01355         fasp_blas_darray_axpy(m,-alpha,t,r);
01356         absres=fasp_blas_darray_norm2(m,r);
01357
01358         // compute reduction factor of residual ||r||
01359         factor=absres/absres0;
01360

```

```

01361         // compute relative residual
01362         switch (StopType) {
01363             case STOP_REL_PRECRES:
01364                 // z = B(r)
01365                 if (pc != NULL)
01366                     pc->fct(r,z,pc->data); /* Apply preconditioner */
01367                 else
01368                     fasp_darray_cp(m,r,z); /* No preconditioner */
01369                 temp2=fasp_blas_darray_dotprod(m,z,r);
01370                 relres=sqrt(ABS(temp2))/normr0;
01371                 break;
01372             case STOP_MOD_REL_RES:
01373                 relres=absres/normu;
01374                 break;
01375             default:
01376                 relres=absres/normr0;
01377                 break;
01378         }
01379
01380         // output iteration information if needed
01381         fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01382
01383         // solution check, if solution is too small, return ERROR_SOLVER_SOLSTAG.
01384         infnormu = fasp_blas_darray_norminf(m, u->val);
01385         if ( infnormu <= sol_inf_tol ) {
01386             if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01387             iter = ERROR_SOLVER_SOLSTAG;
01388             break;
01389         }
01390
01391         // compute relative difference
01392         normu = fasp_blas_darray_norm2(m, u->val);
01393         reldiff = ABS(alpha)*fasp_blas_darray_norm2(m, p)/normu;
01394
01395         // stagnation check
01396         if ( (stag<=MaxStag) & (reldiff<maxdiff) ) {
01397
01398             if ( PrtLvl >= PRINT_MORE ) {
01399                 ITS_DIFFRES(reldiff,relres);
01400                 ITS_RESTART;
01401             }
01402
01403             mf->fct(mf->data, u->val, r);
01404             fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
01405             absres = fasp_blas_darray_norm2(m,r);
01406
01407             // relative residual
01408             switch (StopType) {
01409                 case STOP_REL_PRECRES:
01410                     // z = B(r)
01411                     if (pc != NULL)
01412                         pc->fct(r,z,pc->data); /* Apply preconditioner */
01413                     else
01414                         fasp_darray_cp(m,r,z); /* No preconditioner */
01415                     temp2=fasp_blas_darray_dotprod(m,z,r);
01416                     relres=sqrt(ABS(temp2))/normr0;
01417                     break;
01418                 case STOP_MOD_REL_RES:
01419                     relres=absres/normu;
01420                     break;
01421                 default:
01422                     relres=absres/normr0;
01423                     break;
01424             }
01425
01426             if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01427
01428             if ( relres < tol )
01429                 break;
01430             else {
01431                 if ( stag >= MaxStag ) {
01432                     if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01433                     iter = ERROR_SOLVER_STAG;
01434                     break;
01435                 }
01436                 fasp_darray_set(m,p,0.0);
01437                 ++stag;
01438                 ++restart_step;
01439             }
01440         } // end of stagnation check!
01441

```

```

01442         // safe-guard check
01443         if ( relres < tol ) {
01444             if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01445
01446             mf->fct(mf->data, u->val, r);
01447             fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
01448
01449             // relative residual
01450             switch (StopType) {
01451                 case STOP_REL_PRECRES:
01452                     // z = B(r)
01453                     if (pc != NULL)
01454                         pc->fct(r, z, pc->data); /* Apply preconditioner */
01455                     else
01456                         fasp_darray_cp(m, r, z); /* No preconditioner */
01457                     temp2=fasp_blas_darray_dotprod(m, z, r);
01458                     relres=sqrt(ABS(temp2))/normr0;
01459                     break;
01460                 case STOP_MOD_REL_RES:
01461                     absres=fasp_blas_darray_norm2(m, r);
01462                     relres=absres/normu;
01463                     break;
01464                 default:
01465                     absres=fasp_blas_darray_norm2(m, r);
01466                     relres=absres/normr0;
01467                     break;
01468             }
01469
01470             if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01471
01472             // check convergence
01473             if ( relres < tol ) break;
01474
01475             if ( more_step >= MaxRestartStep ) {
01476                 if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01477                 iter = ERROR_SOLVER_TOLSMALL;
01478                 break;
01479             }
01480
01481             // prepare for restarting method
01482             fasp_darray_set(m, p, 0.0);
01483             ++more_step;
01484             ++restart_step;
01485
01486         } // end of safe-guard check!
01487
01488         // update relative residual here
01489         absres0 = absres;
01490
01491         // compute z_k = B(r_k)
01492         if ( StopType != STOP_REL_PRECRES ) {
01493             if ( pc != NULL )
01494                 pc->fct(r, z, pc->data); /* Apply preconditioner */
01495             else
01496                 fasp_darray_cp(m, r, z); /* No preconditioner, B=I */
01497         }
01498
01499         // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
01500         temp2=fasp_blas_darray_dotprod(m, z, r);
01501         beta=temp2/temp1;
01502         temp1=temp2;
01503
01504         // compute p_k = z_k + beta_k*p_{k-1}
01505         fasp_blas_darray_axpby(m, 1.0, z, beta, p);
01506
01507     } // end of main PCG loop.
01508
01509     FINISHED: // finish iterative method
01510     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter, MaxIt, relres);
01511
01512     // clean up temp memory
01513     fasp_mem_free(work); work = NULL;
01514
01515     #if DEBUG_MODE > 0
01516     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01517     #endif
01518
01519     if (iter>MaxIt)
01520         return ERROR_SOLVER_MAXIT;
01521     else
01522         return iter;

```



```

01523 }
01524
01525 /*-----*/
01526 /*--      End of File      --*/
01527 /*-----*/

```

9.113 KryPgcg.c File Reference

Krylov subspace methods – Preconditioned generalized CG.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dcsr_pgcg](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned generalilzed conjugate gradient (GCG) method for solving $Au=b$.
- [INT fasp_solver_pgcg](#) ([mxv_matfree](#) *mf, [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned generalilzed conjugate gradient (GCG) method for solving $Au=b$.

9.113.1 Detailed Description

Krylov subspace methods – Preconditioned generalized CG.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), and [BlaSpmvCSR.c](#)

Reference: Concus, P. and Golub, G.H. and O'Leary, D.P. A Generalized Conjugate Gradient Method for the Numerical: Solution of Elliptic Partial Differential Equations, Computer Science Department, Stanford University, 1976
Copyright (C) 2012–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPgcg.c](#).

9.113.2 Function Documentation

9.113.2.1 fasp_solver_dcsr_pgcg()

```

INT fasp_solver_dcsr_pgcg (
    dCSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,

```

```
const SHORT StopType,
const SHORT PrtLvl )
```

Preconditioned generalized conjugate gradient (GCG) method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/01/2012

Modified by Chensong Zhang on 05/01/2012

Definition at line 60 of file [KryPgcg.c](#).

9.113.2.2 fasp_solver_pgcg()

```
INT fasp_solver_pgcg (
    mxv_matfree * mf,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned generalized conjugate gradient (GCG) method for solving $Au=b$.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type – DOES not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/01/2012

Note

Not completely implemented yet! –Chensong

Modified by Feiteng Huang on 09/26/2012: matrix free
Definition at line 213 of file KryPgcg.c.

9.114 KryPgcg.c

[Go to the documentation of this file.](#)

```

00001
00022 #include <math.h>
00023
00024 #include "fasp.h"
00025 #include "fasp_functs.h"
00026
00027 /*-----*/
00028 /*--  Declare Private Functions  --*/
00029 /*-----*/
00030
00031 #include "KryUtil.inl"
00032
00033 /*-----*/
00034 /*--      Public Functions      --*/
00035 /*-----*/
00036
00060 INT fasp_solver_dcsr_pgcg (dCSRmat      *A,
00061                          dvector      *b,
00062                          dvector      *u,
00063                          precondition *pc,
00064                          const REAL    tol,
00065                          const INT     MaxIt,
00066                          const SHORT   StopType,
00067                          const SHORT   PrtLvl)
00068 {
00069     INT    iter=0, m=A->row, i;
00070     REAL    absres0 = BIGREAL, absres = BIGREAL;
00071     REAL    relres  = BIGREAL, normb  = BIGREAL;
00072     REAL    alpha, factor;
00073
00074     // allocate temp memory
00075     REAL *work = (REAL *)fasp_mem_calloc(2*m+MaxIt+MaxIt*m, sizeof(REAL));
00076
00077     REAL *r, *Br, *beta, *p;
00078     r = work; Br = r + m; beta = Br + m; p = beta + MaxIt;
00079
00080     // Output some info for debugging
00081     if (PrtLvl > PRINT_NONE) printf("\nCalling GCG solver (CSR) ...\n");
00082
00083     #if DEBUG_MODE > 0
00084         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00085         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00086     #endif
00087
00088     normb=fasp_blas_darray_norm2(m,b->val);
00089
00090     // -----
00091     // 1st iteration (Steepest descent)
00092     // -----
00093     // r = b-A*u

```

```

00094     fasp_darray_cp(m,b->val,r);
00095     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00096
00097     // Br
00098     if (pc != NULL)
00099         pc->fct(r,p,pc->data); /* Preconditioning */
00100     else
00101         fasp_darray_cp(m,r,p); /* No preconditioner, B=I */
00102
00103     // alpha = (p'r)/(p'Ap)
00104     alpha = fasp_blas_darray_dotprod(m,r,p) / fasp_blas_dcsr_vmv(A, p, p);
00105
00106     // u = u + alpha *p
00107     fasp_blas_darray_axpy(m, alpha, p, u->val);
00108
00109     // r = r - alpha *Ap
00110     fasp_blas_dcsr_aAxy((-1.0*alpha),A,p,r);
00111
00112     // norm(r), factor
00113     absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;
00114
00115     // compute relative residual
00116     relres = absres/normb;
00117
00118     // output iteration information if needed
00119     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00120
00121     // update relative residual here
00122     absres0 = absres;
00123
00124     for ( iter = 1; iter < MaxIt ; iter++) {
00125
00126         // Br
00127         if (pc != NULL)
00128             pc->fct(r, Br ,pc->data); // Preconditioning
00129         else
00130             fasp_darray_cp(m,r, Br); // No preconditioner, B=I
00131
00132         // form p
00133         fasp_darray_cp(m, Br, p+iter*m);
00134
00135         for (i=0; i<iter; i++) {
00136             beta[i] = (-1.0) * ( fasp_blas_dcsr_vmv(A, Br, p+i*m)
00137                               /fasp_blas_dcsr_vmv(A, p+i*m, p+i*m) );
00138
00139             fasp_blas_darray_axpy(m, beta[i], p+i*m, p+iter*m);
00140         }
00141
00142         // -----
00143         // next iteration
00144         // -----
00145
00146         // alpha = (p'r)/(p'Ap)
00147         alpha = fasp_blas_darray_dotprod(m,r,p+iter*m)
00148             / fasp_blas_dcsr_vmv(A, p+iter*m, p+iter*m);
00149
00150         // u = u + alpha *p
00151         fasp_blas_darray_axpy(m, alpha, p+iter*m, u->val);
00152
00153         // r = r - alpha *Ap
00154         fasp_blas_dcsr_aAxy((-1.0*alpha),A,p+iter*m,r);
00155
00156         // norm(r), factor
00157         absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;
00158
00159         // compute relative residual
00160         relres = absres/normb;
00161
00162         // output iteration information if needed
00163         fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00164
00165         if (relres < tol) break;
00166
00167         // update relative residual here
00168         absres0 = absres;
00169     } // end of main GCG loop.
00170
00171     // finish iterative method
00172     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00173
00174

```

```

00175 // clean up temp memory
00176 fasp_mem_free(work); work = NULL;
00177
00178 #if DEBUG_MODE > 0
00179 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00180 #endif
00181
00182 if (iter>MaxIt)
00183     return ERROR_SOLVER_MAXIT;
00184 else
00185     return iter;
00186 }
00187
00213 INT fasp_solver_pgcg (mxv_matfree *mf,
00214                      dvector *b,
00215                      dvector *u,
00216                      precondition *pc,
00217                      const REAL tol,
00218                      const INT MaxIt,
00219                      const SHORT StopType,
00220                      const SHORT PrtLvl)
00221 {
00222     INT iter=0, m=b->row, i;
00223     REAL absres0 = BIGREAL, absres = BIGREAL;
00224     REAL relres = BIGREAL, normb = BIGREAL;
00225     REAL alpha, factor, gama_1, gama_2;
00226
00227 // allocate temp memory
00228 REAL *work = (REAL *)fasp_mem_calloc(3*m+MaxIt+MaxIt*m, sizeof(REAL));
00229
00230 REAL *r, *Br, *beta, *p, *q;
00231 q = work; r = q + m; Br = r + m; beta = Br + m; p = beta + MaxIt;
00232
00233 // Output some info for debugging
00234 if (PrtLvl > PRINT_NONE) printf("\nCalling GCG solver (MatFree) ...\n");
00235
00236 #if DEBUG_MODE > 0
00237 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00238 printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00239 #endif
00240
00241 normb=fasp_blas_darray_norm2(m,b->val);
00242
00243 // -----
00244 // 1st iteration (Steepest descent)
00245 // -----
00246 // r = b-A*u
00247 mf->fct(mf->data, u->val, r);
00248 fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
00249
00250 // Br
00251 if (pc != NULL)
00252     pc->fct(r,p,pc->data); /* Preconditioning */
00253 else
00254     fasp_darray_cp(m,r,p); /* No preconditioner, B=I */
00255
00256 // alpha = (p'r)/(p'Ap)
00257 mf->fct(mf->data, p, q);
00258 alpha = fasp_blas_darray_dotprod (m,r,p) / fasp_blas_darray_dotprod (m, p, q);
00259
00260 // u = u + alpha *p
00261 fasp_blas_darray_axpy(m, alpha, p, u->val);
00262
00263 // r = r - alpha *Ap
00264 mf->fct(mf->data, p, q);
00265 fasp_blas_darray_axpby(m, (-1.0*alpha), q, 1.0, r);
00266
00267 // norm(r), factor
00268 absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;
00269
00270 // compute relative residual
00271 relres = absres/normb;
00272
00273 // output iteration information if needed
00274 fasp_itinfo(PrtLvl,StopType,iter+1,relres,absres,factor);
00275
00276 // update relative residual here
00277 absres0 = absres;
00278
00279 for ( iter = 1; iter < MaxIt ; iter++) {
00280

```

```

00281 // Br
00282 if (pc != NULL)
00283     pc->fct(r, Br ,pc->data); // Preconditioning
00284 else
00285     fasp_darray_cp(m,r, Br); // No preconditioner, B=I
00286
00287 // form p
00288 fasp_darray_cp(m, Br, p+iter*m);
00289
00290 for (i=0; i<iter; i++) {
00291     mf->fct(mf->data, Br, q);
00292     gama_1 = fasp_blas_darray_dotprod(m, p+i*m, q);
00293     mf->fct(mf->data, p+i*m, q);
00294     gama_2 = fasp_blas_darray_dotprod(m, p+i*m, q);
00295     beta[i] = (-1.0) * ( gama_1 / gama_2 );
00296
00297     fasp_blas_darray_axpy(m, beta[i], p+i*m, p+iter*m);
00298 }
00299
00300 // -----
00301 // next iteration
00302 // -----
00303
00304 // alpha = (p'r)/(p'Ap)
00305 mf->fct(mf->data, p+iter*m, q);
00306 alpha = fasp_blas_darray_dotprod(m,r,p+iter*m)
00307 / fasp_blas_darray_dotprod (m, q, p+iter*m);
00308
00309 // u = u + alpha *p
00310 fasp_blas_darray_axpy(m, alpha , p+iter*m, u->val);
00311
00312 // r = r - alpha *Ap
00313 mf->fct(mf->data, p+iter*m, q);
00314 fasp_blas_darray_axpby(m, (-1.0*alpha), q, 1.0, r);
00315
00316 // norm(r), factor
00317 absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;
00318
00319 // compute relative residual
00320 relres = absres/normb;
00321
00322 // output iteration information if needed
00323 fasp_itinfo(PrtLvl,StopType,iter+1,relres,absres,factor);
00324
00325 if (relres < tol) break;
00326
00327 // update relative residual here
00328 absres0 = absres;
00329
00330 } // end of main GCG loop.
00331
00332 // finish iterative method
00333 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00334
00335 // clean up temp memory
00336 fasp_mem_free(work); work = NULL;
00337
00338 #if DEBUG_MODE > 0
00339     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00340 #endif
00341
00342 if (iter>MaxIt)
00343     return ERROR_SOLVER_MAXIT;
00344 else
00345     return iter;
00346 }
00347
00348 /*-----*/
00349 /*--          End of File          --*/
00350 /*-----*/

```

9.115 KryPgcr.c File Reference

Krylov subspace methods – Preconditioned GCR.

```

#include <math.h>
#include "fasp.h"

```

```
#include "fasp_funcs.h"
#include "KryUtil.inl"
```

Functions

- `INT fasp_solver_dcsr_pgcr` (`dCSRmat *A`, `dvector *b`, `dvector *x`, `precond *pc`, const `REAL` `tol`, const `INT` `MaxIt`, const `SHORT` `restart`, const `SHORT` `StopType`, const `SHORT` `PrtLvl`)
A preconditioned GCR method for solving $Au=b$.
- `INT fasp_solver_dblc_pgcr` (`dBLCmat *A`, `dvector *b`, `dvector *x`, `precond *pc`, const `REAL` `tol`, const `INT` `MaxIt`, const `SHORT` `restart`, const `SHORT` `StopType`, const `SHORT` `PrtLvl`)
A preconditioned GCR method for solving $Au=b$.

9.115.1 Detailed Description

Krylov subspace methods – Preconditioned GCR.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvCSR.c](#), and [BlaVector.c](#)

Copyright (C) 2014–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPgcr.c](#).

9.115.2 Function Documentation

9.115.2.1 `fasp_solver_dblc_pgcr()`

```
INT fasp_solver_dblc_pgcr (
    dBLCmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

A preconditioned GCR method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>x</i>	Pointer to dvector of dofs
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopage
<i>MaxIt</i>	Maximal number of iterations

Parameters

<i>restart</i>	Restart number for GCR
<i>StopType</i>	Stopping type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Reference: YVAN NOTAY "AN AGGREGATION-BASED ALGEBRAIC MULTIGRID METHOD"

Author

Zheng Li

Date

12/23/2014

Definition at line 249 of file [KryPgcr.c](#).

9.115.2.2 fasp_solver_dcsr_pgcr()

```

INT fasp_solver_dcsr_pgcr (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

A preconditioned GCR method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to dvector of right hand side
<i>x</i>	Pointer to dvector of dofs
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stoppage
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restart number for GCR
<i>StopType</i>	Stopping type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Reference: YVAN NOTAY "AN AGGREGATION-BASED ALGEBRAIC MULTIGRID METHOD"

Author

Zheng Li

Date

12/23/2014

Definition at line 55 of file KryPgcr.c.

9.116 KryPgcr.c

[Go to the documentation of this file.](#)

```

00001
00017 #include <math.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 #include "KryUtil.inl"
00027
00028 static void dense_aAtxpy (INT, INT, REAL *, REAL, REAL *, REAL, REAL *);
00029
00055 INT fasp_solver_dcsr_pgcr (dCSRmat      *A,
00056                          dvector      *b,
00057                          dvector      *x,
00058                          precondition *pc,
00059                          const REAL   tol,
00060                          const INT     MaxIt,
00061                          const SHORT  restart,
00062                          const SHORT  StopType,
00063                          const SHORT  PrtLvl)
00064 {
00065     const INT    n = b->row;
00066
00067     // local variables
00068     INT    iter = 0;
00069     int     i, j, k, rst = -1; // must be signed!  -zcs
00070
00071     REAL    gamma, alpha, beta, checktol;
00072     REAL    absres0 = BIGREAL, absres = BIGREAL;
00073     REAL    relres  = BIGREAL;
00074
00075     // allocate temp memory (need about (restart+4)*n REAL numbers)
00076     REAL    *c = NULL, *z = NULL, *alp = NULL, *tmpx = NULL;
00077     REAL    *norms = NULL, *r = NULL, *work = NULL;
00078     REAL    **h = NULL;
00079
00080     INT      Restart = MIN(restart, MaxIt);
00081     LONG     worksize = n+2*Restart*n+Restart+Restart;
00082
00083     // Output some info for debugging
00084     if ( PrtLvl > PRINT_NONE ) printf("\nCalling GCR solver (CSR) ...\n");
00085
00086 #if DEBUG_MODE > 0
00087     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00088     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00089 #endif
00090
00091     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00092
00093     /* check whether memory is enough for GCR */
00094     while ( (work == NULL) && (Restart > 5) ) {
00095         Restart = Restart - 5;
00096         worksize = n+2*Restart*n+Restart+Restart;
00097         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00098     }
00099
00100     if ( work == NULL ) {
00101         printf("### ERROR: No enough memory for GCR! [%s:%d]\n",
00102             __FILE__, __LINE__ );
00103         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);

```

```

00104     }
00105
00106     if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00107         printf("### WARNING: GCR restart number set to %d!\n", Restart);
00108     }
00109
00110     r = work; z = r+n; c = z + Restart*n; alp = c + Restart*n; tmpx = alp + Restart;
00111
00112     h = (REAL **)fasp_mem_calloc(Restart, sizeof(REAL *));
00113     for (i = 0; i < Restart; i++) h[i] = (REAL*)fasp_mem_calloc(Restart, sizeof(REAL));
00114
00115     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00116
00117     // r = b-A*x
00118     fasp_darray_cp(n, b->val, r);
00119     fasp_blas_dcsr_aApy(-1.0, A, x->val, r);
00120
00121     absres = fasp_blas_darray_dotprod(n, r, r);
00122
00123     absres0 = MAX(SMALLREAL, absres);
00124
00125     relres = absres/absres0;
00126
00127     // output iteration information if needed
00128     fasp_itinfo(PrtLvl, StopType, 0, relres, sqrt(absres0), 0.0);
00129
00130     // store initial residual
00131     norms[0] = relres;
00132
00133     checktol = MAX(tol*tol*absres0, absres*1.0e-4);
00134
00135     while ( iter < MaxIt && sqrt(relres) > tol ) {
00136
00137         i = -1; rst ++;
00138
00139         while ( i < Restart-1 && iter < MaxIt ) {
00140
00141             i++; iter++;
00142
00143             // z = B^-1r
00144             if ( pc == NULL )
00145                 fasp_darray_cp(n, r, &z[i*n]);
00146             else
00147                 pc->fct(r, &z[i*n], pc->data);
00148
00149             // c = Az
00150             fasp_blas_dcsr_mxv(A, &z[i*n], &c[i*n]);
00151
00152             /* Modified Gram_Schmidt orthogonalization */
00153             for ( j = 0; j < i; j++ ) {
00154                 gamma = fasp_blas_darray_dotprod(n, &c[j*n], &c[i*n]);
00155                 h[i][j] = gamma/h[j][j];
00156                 fasp_blas_darray_axpy(n, -h[i][j], &c[j*n], &c[i*n]);
00157             }
00158             // gamma = (c, c)
00159             gamma = fasp_blas_darray_dotprod(n, &c[i*n], &c[i*n]);
00160
00161             h[i][i] = gamma;
00162
00163             // alpha = (c, r)
00164             alpha = fasp_blas_darray_dotprod(n, &c[i*n], r);
00165
00166             beta = alpha/gamma;
00167
00168             alp[i] = beta;
00169
00170             // r = r - beta*c
00171             fasp_blas_darray_axpy(n, -beta, &c[i*n], r);
00172
00173             // equivalent to ||r||_2
00174             absres = absres - alpha*alpha/gamma;
00175
00176             if (absres < checktol) {
00177                 absres = fasp_blas_darray_dotprod(n, r, r);
00178                 checktol = MAX(tol*tol*absres0, absres*1.0e-4);
00179             }
00180
00181             relres = absres / absres0;
00182
00183             norms[iter] = relres;
00184

```

```

00185         fasp_itinfo(PrtLvl, StopType, iter, sqrt(relres), sqrt(absres),
00186                     sqrt(norms[iter]/norms[iter-1]));
00187
00188         if (sqrt(relres) < tol) break;
00189     }
00190
00191     for ( k = i; k >=0; k-- ) {
00192         tmpx[k] = alp[k];
00193         for (j=0; j<k; ++j) {
00194             alp[j] -= h[k][j]*tmpx[k];
00195         }
00196     }
00197
00198     if (rst==0) dense_aAtxpby(n, i+1, z, 1.0, tmpx, 0.0, x->val);
00199     else dense_aAtxpby(n, i+1, z, 1.0, tmpx, 1.0, x->val);
00200
00201 }
00202
00203 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,sqrt(relres));
00204
00205 // clean up memory
00206 for (i = 0; i < Restart; i++) {
00207     fasp_mem_free(h[i]); h[i] = NULL;
00208 }
00209 fasp_mem_free(h); h = NULL;
00210
00211 fasp_mem_free(work); work = NULL;
00212 fasp_mem_free(norms); norms = NULL;
00213
00214 #if DEBUG_MODE > 0
00215 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00216 #endif
00217
00218 if ( iter >= MaxIt )
00219     return ERROR_SOLVER_MAXIT;
00220 else
00221     return iter;
00222 }
00223
00249 INT fasp_solver_dblc_pgcr (dBLMat      *A,
00250                          dvector      *b,
00251                          dvector      *x,
00252                          precondition *pc,
00253                          const REAL   tol,
00254                          const INT     MaxIt,
00255                          const SHORT   restart,
00256                          const SHORT   StopType,
00257                          const SHORT   PrtLvl)
00258 {
00259     const INT    n = b->row;
00260
00261     // local variables
00262     INT         iter = 0;
00263     int          i, j, k, rst = -1; // must be signed! -zcs
00264
00265     REAL         gamma, alpha, beta, checktol;
00266     REAL         absres0 = BIGREAL, absres = BIGREAL;
00267     REAL         relres = BIGREAL;
00268
00269     // allocate temp memory (need about (restart+4)*n REAL numbers)
00270     REAL         *c = NULL, *z = NULL, *alp = NULL, *tmpx = NULL;
00271     REAL         *norms = NULL, *r = NULL, *work = NULL;
00272     REAL         **h = NULL;
00273
00274     INT          Restart = MIN(restart, MaxIt);
00275     LONG         worksize = n+2*Restart*n+Restart+Restart;
00276
00277     // Output some info for debugging
00278     if ( PrtLvl > PRINT_NONE ) printf("\nCalling GCR solver (BLC) ...\n");
00279
00280 #if DEBUG_MODE > 0
00281     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00282     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00283 #endif
00284
00285     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00286
00287     /* check whether memory is enough for GCR */
00288     while ( (work == NULL) && (Restart > 5) ) {
00289         Restart = Restart - 5;
00290         worksize = n+2*Restart*n+Restart+Restart;

```

```

00291     work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
00292 }
00293
00294 if ( work == NULL ) {
00295     printf("### ERROR: No enough memory for GCR! [%s:%d]\n",
00296         __FILE__, __LINE__ );
00297     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00298 }
00299
00300 if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00301     printf("### WARNING: GCR restart number set to %d!\n", Restart);
00302 }
00303
00304 r = work; z = r+n; c = z + Restart*n; alp = c + Restart*n; tmpx = alp + Restart;
00305
00306 h = (REAL **)fasp_mem_malloc(Restart, sizeof(REAL *));
00307 for ( i = 0; i < Restart; i++) h[i] = (REAL*)fasp_mem_malloc(Restart, sizeof(REAL));
00308
00309 norms = (REAL *) fasp_mem_malloc(MaxIt+1, sizeof(REAL));
00310
00311 // r = b-A*x
00312 fasp_darray_cp(n, b->val, r);
00313 fasp_blas_dblc_aAxy(-1.0, A, x->val, r);
00314
00315 absres = fasp_blas_darray_dotprod(n, r, r);
00316
00317 absres0 = MAX(SMALLREAL,absres);
00318
00319 relres = absres/absres0;
00320
00321 // output iteration information if needed
00322 fasp_itinfo(PrtLvl,StopType,0,relres,sqrt(absres0),0.0);
00323
00324 // store initial residual
00325 norms[0] = relres;
00326
00327 checktol = MAX(tol*tol*absres0, absres*1.0e-4);
00328
00329 while ( iter < MaxIt && sqrt(relres) > tol ) {
00330
00331     i = 0; rst ++;
00332     while ( i < Restart && iter < MaxIt ) {
00333
00334         iter++;
00335
00336         // z = B^-1r
00337         if ( pc == NULL )
00338             fasp_darray_cp(n, r, &z[i*n]);
00339         else
00340             pc->fct(r, &z[i*n], pc->data);
00341
00342         // c = Az
00343         fasp_blas_dblc_mxv(A, &z[i*n], &c[i*n]);
00344
00345         /* Modified Gram_Schmidt orthogonalization */
00346         for ( j = 0; j < i; j++ ) {
00347             gamma = fasp_blas_darray_dotprod(n, &c[j*n], &c[i*n]);
00348             h[i][j] = gamma/h[j][j];
00349             fasp_blas_darray_axpy(n, -h[i][j], &c[j*n], &c[i*n]);
00350         }
00351         // gamma = (c,c)
00352         gamma = fasp_blas_darray_dotprod(n, &c[i*n], &c[i*n]);
00353
00354         h[i][i] = gamma;
00355
00356         // alpha = (C, r)
00357         alpha = fasp_blas_darray_dotprod(n, &c[i*n], r);
00358
00359         beta = alpha/gamma;
00360
00361         alp[i] = beta;
00362
00363         // r = r - beta*c
00364         fasp_blas_darray_axpy(n, -beta, &c[i*n], r);
00365
00366         // equivalent to ||r||_2
00367         absres = absres - alpha*alpha/gamma;
00368
00369         if (absres < checktol) {
00370             absres = fasp_blas_darray_dotprod(n, r, r);
00371             checktol = MAX(tol*tol*absres0, absres*1.0e-4);

```

```

00372     }
00373
00374     relres = absres / absres0;
00375
00376     norms[iter] = relres;
00377
00378     fasp_itinfo(PrtLvl, StopType, iter, sqrt(relres), sqrt(absres),
00379               sqrt(norms[iter]/norms[iter-1]));
00380
00381     if (sqrt(relres) < tol) break;
00382
00383     i++;
00384 }
00385
00386 for ( k = i; k >=0; k-- ) {
00387     tmpx[k] = alp[k];
00388     for (j=0; j<k; ++j) {
00389         alp[j] -= h[k][j]*tmpx[k];
00390     }
00391 }
00392
00393 if (rst==0) dense_aAtxpby(n, i+1, z, 1.0, tmpx, 0.0, x->val);
00394 else dense_aAtxpby(n, i+1, z, 1.0, tmpx, 1.0, x->val);
00395 }
00396
00397 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,sqrt(relres));
00398
00399 // clean up memory
00400 for (i = 0; i < Restart; i++) {
00401     fasp_mem_free(h[i]); h[i] = NULL;
00402 }
00403 fasp_mem_free(h); h = NULL;
00404
00405 fasp_mem_free(work); work = NULL;
00406 fasp_mem_free(norms); norms = NULL;
00407
00408 #if DEBUG_MODE > 0
00409 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00410 #endif
00411
00412 if ( iter >= MaxIt )
00413     return ERROR_SOLVER_MAXIT;
00414 else
00415     return iter;
00416 }
00417
00418 /*-----*/
00419 /*--      Private Functions      --*/
00420 /*-----*/
00421
00422 static void dense_aAtxpby (INT    n,
00423                          INT    m,
00424                          REAL    *A,
00425                          REAL    alpha,
00426                          REAL    *x,
00427                          REAL    beta,
00428                          REAL    *y)
00429 {
00430     INT i, j;
00431
00432     for (i=0; i<m; i++) fasp_blas_darray_ax(n, x[i], &A[i*n]);
00433
00434     for (j=1; j<m; j++) {
00435         for (i=0; i<n; i++) {
00436             A[i] += A[i+j*n];
00437         }
00438     }
00439
00440     fasp_blas_darray_axpby(n, alpha, A, beta, y);
00441 }
00442
00443 /*-----*/
00444 /*--      End of File      --*/
00445 /*-----*/

```

9.117 KryPgmres.c File Reference

Krylov subspace methods – Right-preconditioned GMRes.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"
```

Functions

- [INT fasp_solver_dcsr_pgmres](#) (dCSRmat *A, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)
Right preconditioned GMRES method for solving $Au=b$.
- [INT fasp_solver_dbsr_pgmres](#) (dBSRmat *A, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)
Preconditioned GMRES method for solving $Au=b$.
- [INT fasp_solver_dblc_pgmres](#) (dBLCmat *A, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)
Preconditioned GMRES method for solving $Au=b$.
- [INT fasp_solver_dstr_pgmres](#) (dSTRmat *A, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)
Preconditioned GMRES method for solving $Au=b$.
- [INT fasp_solver_pgmres](#) (mxv_matfree *mf, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)
Solve " $Ax=b$ " using PGMRES (right preconditioned) iterative method.

9.117.1 Detailed Description

Krylov subspace methods – Right-preconditioned GMRes.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

See also [KryPvgmres.c](#) for a variable restarting version.

See [KrySPgmres.c](#) for a safer version

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPgmres.c](#).

9.117.2 Function Documentation

9.117.2.1 fasp_solver_dblc_pgmres()

```
INT fasp_solver_dblc_pgmres (
    dBLCmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
```

```

const REAL tol,
const INT MaxIt,
const SHORT restart,
const SHORT StopType,
const SHORT PrtLvl )

```

Preconditioned GMRES method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBLMat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/24/2010

Modified by Chensong Zhang on 04/05/2013: add StopType and safe check
Definition at line 675 of file [KryPgmres.c](#).

9.117.2.2 fasp_solver_dbsr_pgmres()

```

INT fasp_solver_dbsr_pgmres (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned GMRES method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side

Parameters

<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/12/21

Modified by Chensong Zhang on 04/05/2013: add StopType and safe check
Definition at line 370 of file [KryPgmres.c](#).

9.117.2.3 fasp_solver_dcsr_pgmres()

```

INT fasp_solver_dcsr_pgmres (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Right preconditioned GMRES method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/11/28

Modified by Chensong Zhang on 04/05/2013: Add StopType and safe check Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate Modified by Chensong Zhang on 09/21/2014: Add comments and reorganize code

Definition at line 67 of file [KryPgmres.c](#).

9.117.2.4 fasp_solver_dstr_pgmres()

```
INT fasp_solver_dstr_pgmres (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned GMRES method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/11/28

Modified by Chensong Zhang on 04/05/2013: add StopType and safe check
 Definition at line 979 of file [KryPgmres.c](#).

9.117.2.5 fasp_solver_pgmres()

```

INT fasp_solver_pgmres (
    mxv_matfree * mf,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PGMRES (right preconditioned) iterative method.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type – DOES not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/11/28

Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate
 Definition at line 1283 of file [KryPgmres.c](#).

9.118 KryPgmres.c

[Go to the documentation of this file.](#)

```

00001
00025 #include <math.h>

```

```

00026
00027 #include "fasp.h"
00028 #include "fasp_functs.h"
00029
00030 /*-----*/
00031 /*--  Declare Private Functions  --*/
00032 /*-----*/
00033
00034 #include "KryUtil.inl"
00035
00036 /*-----*/
00037 /*--      Public Functions      --*/
00038 /*-----*/
00039
00067 INT fasp_solver_dcsr_pgmres (dCSRmat    *A,
00068                             dvector    *b,
00069                             dvector    *x,
00070                             precondition *pc,
00071                             const REAL  tol,
00072                             const INT   MaxIt,
00073                             const SHORT restart,
00074                             const SHORT StopType,
00075                             const SHORT PrtLvl)
00076 {
00077     const INT    n      = b->row;
00078     const INT    MIN_ITER = 0;
00079
00080     // local variables
00081     INT          iter = 0;
00082     int          i, j, k; // must be signed! -zcs
00083
00084     REAL         r_norm, r_normb, gamma, t;
00085     REAL         absres0 = BIGREAL, absres = BIGREAL;
00086     REAL         relres  = BIGREAL, normu  = BIGREAL;
00087
00088     // allocate temp memory (need about (restart+4)*n REAL numbers)
00089     REAL         *c = NULL, *s = NULL, *rs = NULL;
00090     REAL         *norms = NULL, *r = NULL, *w = NULL;
00091     REAL         *work = NULL;
00092     REAL         **p = NULL, **hh = NULL;
00093
00094     INT          Restart  = MIN(restart, MaxIt);
00095     INT          Restart1 = Restart + 1;
00096     LONG         worksize = (Restart+4)*(Restart+n)+1-n;
00097
00098     /* allocate memory and setup temp work space */
00099     work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
00100
00101     // Output some info for debugging
00102     if (PrtLvl > PRINT_NONE) printf("\nCalling GMRes solver (CSR) ...\n");
00103
00104 #if DEBUG_MODE > 0
00105     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00106     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00107 #endif
00108
00109     /* check whether memory is enough for GMRES */
00110     while ( (work == NULL) && (Restart > 5) ) {
00111         Restart = Restart - 5;
00112         Restart1 = Restart + 1;
00113         worksize = (Restart+4)*(Restart+n)+1-n;
00114         work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
00115     }
00116
00117     if (work == NULL) {
00118         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00119         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00120     }
00121
00122     if (PrtLvl > PRINT_MIN && Restart < restart) {
00123         printf("### WARNING: GMRES restart number set to %d!\n", Restart);
00124     }
00125
00126     p      = (REAL **)fasp_mem_malloc(Restart1, sizeof(REAL *));
00127     hh     = (REAL **)fasp_mem_malloc(Restart1, sizeof(REAL *));
00128     norms  = (REAL *) fasp_mem_malloc(MaxIt+1, sizeof(REAL));
00129
00130     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00131
00132     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00133

```

```

00134     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00135
00136     // compute initial residual: r = b-A*x
00137     fasp_darray_cp(n, b->val, p[0]);
00138     fasp_blas_dcsr_aApy(-1.0, A, x->val, p[0]);
00139     r_norm = fasp_blas_darray_norm2(n,p[0]);
00140
00141     // compute stopping criteria
00142     switch (StopType) {
00143     case STOP_REL_RES:
00144         absres0 = MAX(SMALLREAL,r_norm);
00145         relres = r_norm/absres0;
00146         break;
00147     case STOP_REL_PRECRES:
00148         if ( pc == NULL )
00149             fasp_darray_cp(n, p[0], r);
00150         else
00151             pc->fct(p[0], r, pc->data);
00152         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00153         absres0 = MAX(SMALLREAL,r_normb);
00154         relres = r_normb/absres0;
00155         break;
00156     case STOP_MOD_REL_RES:
00157         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
00158         absres0 = r_norm;
00159         relres = absres0/normu;
00160         break;
00161     default:
00162         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00163         goto FINISHED;
00164     }
00165
00166     // if initial residual is small, no need to iterate!
00167     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00168
00169     // output iteration information if needed
00170     fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0.0);
00171
00172     // store initial residual
00173     norms[0] = relres;
00174
00175     /* GMRES(M) outer iteration */
00176     while ( iter < MaxIt && relres > tol ) {
00177
00178         rs[0] = r_norm;
00179
00180         t = 1.0 / r_norm;
00181
00182         fasp_blas_darray_ax(n, t, p[0]);
00183
00184         /* RESTART CYCLE (right-preconditioning) */
00185         i = 0;
00186         while ( i < Restart && iter < MaxIt ) {
00187
00188             i++; iter++;
00189
00190             /* apply preconditioner */
00191             if ( pc == NULL )
00192                 fasp_darray_cp(n, p[i-1], r);
00193             else
00194                 pc->fct(p[i-1], r, pc->data);
00195
00196             fasp_blas_dcsr_mxv(A, r, p[i]);
00197
00198             /* Modified Gram_Schmidt orthogonalization */
00199             for ( j = 0; j < i; j++ ) {
00200                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00201                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00202             }
00203             t = fasp_blas_darray_norm2(n, p[i]);
00204             hh[i][i-1] = t;
00205
00206             if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
00207                 t = 1.0/t;
00208                 fasp_blas_darray_ax(n, t, p[i]);
00209             }
00210
00211             for ( j = 1; j < i; ++j ) {
00212                 t = hh[j-1][i-1];
00213                 hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00214                 hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];

```

```

00215     }
00216     t = hh[i][i-1]*hh[i][i-1];
00217     t += hh[i-1][i-1]*hh[i-1][i-1];
00218
00219     gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
00220     c[i-1] = hh[i-1][i-1] / gamma;
00221     s[i-1] = hh[i][i-1] / gamma;
00222     rs[i] = -s[i-1]*rs[i-1];
00223     rs[i-1] = c[i-1]*rs[i-1];
00224     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00225
00226     absres = r_norm = fabs(rs[i]);
00227
00228     relres = absres/absres0;
00229
00230     norms[iter] = relres;
00231
00232     // output iteration information if needed
00233     fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00234               norms[iter]/norms[iter-1]);
00235
00236     // exit restart cycle if reaches tolerance
00237     if ( relres < tol && iter >= MIN_ITER ) break;
00238
00239 } /* end of restart cycle */
00240
00241 /* compute solution, first solve upper triangular system */
00242 rs[i-1] = rs[i-1] / hh[i-1][i-1];
00243 for ( k = i-2; k >= 0; k-- ) {
00244     t = 0.0;
00245     for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
00246     t += rs[k];
00247     rs[k] = t / hh[k][k];
00248 }
00249
00250 fasp_darray_cp(n, p[i-1], w);
00251
00252 fasp_blas_darray_ax(n, rs[i-1], w);
00253
00254 for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00255
00256 /* apply preconditioner */
00257 if ( pc == NULL )
00258     fasp_darray_cp(n, w, r);
00259 else
00260     pc->fct(w, r, pc->data);
00261
00262 fasp_blas_darray_axpy(n, 1.0, r, x->val);
00263
00264 // Check: prevent false convergence
00265 if ( relres < tol && iter >= MIN_ITER ) {
00266
00267     REAL computed_relres = relres;
00268
00269     // compute residual
00270     fasp_darray_cp(n, b->val, r);
00271     fasp_blas_dcsr_aAxy(-1.0, A, x->val, r);
00272     r_norm = fasp_blas_darray_norm2(n, r);
00273
00274     switch ( StopType ) {
00275     case STOP_REL_RES:
00276         absres = r_norm;
00277         relres = absres/absres0;
00278         break;
00279     case STOP_REL_PRECRES:
00280         if ( pc == NULL )
00281             fasp_darray_cp(n, r, w);
00282         else
00283             pc->fct(r, w, pc->data);
00284         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00285         relres = absres/absres0;
00286         break;
00287     case STOP_MOD_REL_RES:
00288         absres = r_norm;
00289         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00290         relres = absres/normu;
00291         break;
00292     }
00293
00294     norms[iter] = relres;
00295

```

```

00296         if ( relres < tol ) {
00297             break;
00298         }
00299         else { // Need to restart
00300             fasp_darray_cp(n, r, p[0]); i = 0;
00301         }
00302
00303         if ( PrtLvl >= PRINT_MORE ) {
00304             ITS_COMPRES(computed_relres); ITS_REALRES(relres);
00305         }
00306
00307     } /* end of convergence check */
00308
00309     /* compute residual vector and continue loop */
00310     for ( j = i; j > 0; j-- ) {
00311         rs[j-1] = -s[j-1]*rs[j];
00312         rs[j]   = c[j-1]*rs[j];
00313     }
00314
00315     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00316
00317     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00318
00319     if ( i ) {
00320         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00321         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00322     }
00323
00324 } /* end of main while loop */
00325 FINISHED:
00326     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00327
00328     /*-----
00329 * Clean up workspace
00330 *-----*/
00331     fasp_mem_free(work); work = NULL;
00332     fasp_mem_free(p); p = NULL;
00333     fasp_mem_free(hh); hh = NULL;
00334     fasp_mem_free(norms); norms = NULL;
00335
00336 #if DEBUG_MODE > 0
00337     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00338 #endif
00339
00340     if ( iter >= MaxIt )
00341         return ERROR_SOLVER_MAXIT;
00342     else
00343         return iter;
00344 }
00345
00370 INT fasp_solver_dbsr_pgmres (dBSRmat *A,
00371                             dvector *b,
00372                             dvector *x,
00373                             precondition *pc,
00374                             const REAL tol,
00375                             const INT MaxIt,
00376                             const SHORT restart,
00377                             const SHORT StopType,
00378                             const SHORT PrtLvl)
00379 {
00380     const INT n = b->row;
00381     const INT MIN_ITER = 0;
00382
00383     // local variables
00384     INT iter = 0;
00385     int i, j, k; // must be signed! -zcs
00386
00387     REAL r_norm, r_normb, gamma, t;
00388     REAL absres0 = BIGREAL, absres = BIGREAL;
00389     REAL relres = BIGREAL, normu = BIGREAL;
00390
00391     // allocate temp memory (need about (restart+4)*n REAL numbers)
00392     REAL *c = NULL, *s = NULL, *rs = NULL;
00393     REAL *norms = NULL, *r = NULL, *w = NULL;
00394     REAL *work = NULL;
00395     REAL **p = NULL, **hh = NULL;
00396
00397     INT Restart = MIN(restart, MaxIt);
00398     INT Restart1 = Restart + 1;
00399     LONG worksize = (Restart+4)*(Restart+n)+1-n;
00400

```

```

00401      /* allocate memory and setup temp work space */
00402      work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00403
00404      // Output some info for debugging
00405      if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (BSR) ...\n");
00406
00407      #if DEBUG_MODE > 0
00408          printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00409          printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00410      #endif
00411
00412      /* check whether memory is enough for GMRES */
00413      while ( (work == NULL) && (Restart > 5) ) {
00414          Restart = Restart - 5;
00415          Restart1 = Restart + 1;
00416          worksize = (Restart+4)*(Restart+n)+1-n;
00417          work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00418      }
00419
00420      if ( work == NULL ) {
00421          printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00422          fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00423      }
00424
00425      if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00426          printf("### WARNING: GMRES restart number set to %d!\n", Restart);
00427      }
00428
00429      p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00430      hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00431      norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00432
00433      r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00434
00435      for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00436
00437      for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00438
00439      // compute initial residual: r = b-A*x
00440      fasp_darray_cp(n, b->val, p[0]);
00441      fasp_blas_dbsr_aAxy(-1.0, A, x->val, p[0]);
00442      r_norm = fasp_blas_darray_norm2(n,p[0]);
00443
00444      // compute stopping criteria
00445      switch (StopType) {
00446          case STOP_REL_RES:
00447              absres0 = MAX(SMALLREAL, r_norm);
00448              relres = r_norm/absres0;
00449              break;
00450          case STOP_REL_PRECRES:
00451              if ( pc == NULL )
00452                  fasp_darray_cp(n, p[0], r);
00453              else
00454                  pc->fct(p[0], r, pc->data);
00455              r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00456              absres0 = MAX(SMALLREAL, r_normb);
00457              relres = r_normb/absres0;
00458              break;
00459          case STOP_MOD_REL_RES:
00460              normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00461              absres0 = r_norm;
00462              relres = absres0/normu;
00463              break;
00464          default:
00465              printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00466              goto FINISHED;
00467      }
00468
00469      // if initial residual is small, no need to iterate!
00470      if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00471
00472      // output iteration information if needed
00473      fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0.0);
00474
00475      // store initial residual
00476      norms[0] = relres;
00477
00478      /* GMRES(M) outer iteration */
00479      while ( iter < MaxIt && relres > tol ) {
00480          rs[0] = r_norm;

```

```

00482
00483     t = 1.0 / r_norm;
00484
00485     fasp_blas_darray_ax(n, t, p[0]);
00486
00487     /* RESTART CYCLE (right-preconditioning) */
00488     i = 0;
00489     while ( i < Restart && iter < MaxIt ) {
00490
00491         i++; iter++;
00492
00493         /* apply preconditioner */
00494         if ( pc == NULL )
00495             fasp_darray_cp(n, p[i-1], r);
00496         else
00497             pc->fct(p[i-1], r, pc->data);
00498
00499         fasp_blas_dbsr_mxv(A, r, p[i]);
00500
00501         /* Modified Gram_Schmidt orthogonalization */
00502         for ( j = 0; j < i; j++ ) {
00503             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00504             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00505         }
00506         t = fasp_blas_darray_norm2(n, p[i]);
00507         hh[i][i-1] = t;
00508
00509         if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
00510             t = 1.0/t;
00511             fasp_blas_darray_ax(n, t, p[i]);
00512         }
00513
00514         for ( j = 1; j < i; ++j ) {
00515             t = hh[j-1][i-1];
00516             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00517             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00518         }
00519         t = hh[i][i-1]*hh[i][i-1];
00520         t += hh[i-1][i-1]*hh[i-1][i-1];
00521
00522         gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
00523         c[i-1] = hh[i-1][i-1] / gamma;
00524         s[i-1] = hh[i][i-1] / gamma;
00525         rs[i] = -s[i-1]*rs[i-1];
00526         rs[i-1] = c[i-1]*rs[i-1];
00527         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00528
00529         absres = r_norm = fabs(rs[i]);
00530
00531         relres = absres/absres0;
00532
00533         norms[iter] = relres;
00534
00535         // output iteration information if needed
00536         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00537             norms[iter]/norms[iter-1]);
00538
00539         // exit restart cycle if reaches tolerance
00540         if ( relres < tol && iter >= MIN_ITER ) break;
00541
00542     } /* end of restart cycle */
00543
00544     /* compute solution, first solve upper triangular system */
00545     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00546     for ( k = i-2; k >= 0; k-- ) {
00547         t = 0.0;
00548         for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
00549         t += rs[k];
00550         rs[k] = t / hh[k][k];
00551     }
00552
00553     fasp_darray_cp(n, p[i-1], w);
00554
00555     fasp_blas_darray_ax(n, rs[i-1], w);
00556
00557     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00558
00559     /* apply preconditioner */
00560     if ( pc == NULL )
00561         fasp_darray_cp(n, w, r);
00562     else

```



```

00563     pc->fct(w, r, pc->data);
00564
00565     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00566
00567     // Check: prevent false convergence
00568     if ( relres < tol && iter >= MIN_ITER ) {
00569
00570         REAL computed_relres = relres;
00571
00572         // compute residual
00573         fasp_darray_cp(n, b->val, r);
00574         fasp_blas_dbsr_aApy(-1.0, A, x->val, r);
00575         r_norm = fasp_blas_darray_norm2(n, r);
00576
00577         switch ( StopType ) {
00578             case STOP_REL_RES:
00579                 absres = r_norm;
00580                 relres = absres/absres0;
00581                 break;
00582             case STOP_REL_PRECRES:
00583                 if ( pc == NULL )
00584                     fasp_darray_cp(n, r, w);
00585                 else
00586                     pc->fct(r, w, pc->data);
00587                 absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00588                 relres = absres/absres0;
00589                 break;
00590             case STOP_MOD_REL_RES:
00591                 absres = r_norm;
00592                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00593                 relres = absres/normu;
00594                 break;
00595         }
00596
00597         norms[iter] = relres;
00598
00599         if ( relres < tol ) {
00600             break;
00601         }
00602         else { // Need to restart
00603             fasp_darray_cp(n, r, p[0]); i = 0;
00604         }
00605
00606         if ( PrtLvl >= PRINT_MORE ) {
00607             ITS_COMPRES(computed_relres); ITS_REALRES(relres);
00608         }
00609
00610
00611     } /* end of convergence check */
00612
00613     /* compute residual vector and continue loop */
00614     for ( j = i; j > 0; j-- ) {
00615         rs[j-1] = -s[j-1]*rs[j];
00616         rs[j] = c[j-1]*rs[j];
00617     }
00618
00619     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00620
00621     for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00622
00623     if ( i ) {
00624         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00625         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00626     }
00627
00628     } /* end of main while loop */
00629
00630 FINISHED:
00631     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00632
00633     /*-----
00634  * Clean up workspace
00635  *-----*/
00636     fasp_mem_free(work);    work = NULL;
00637     fasp_mem_free(p);       p = NULL;
00638     fasp_mem_free(hh);      hh = NULL;
00639     fasp_mem_free(norms);   norms = NULL;
00640
00641 #if DEBUG_MODE > 0
00642     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00643 #endif

```

```

00644
00645     if ( iter >= MaxIt )
00646     return ERROR_SOLVER_MAXIT;
00647     else
00648     return iter;
00649 }
00650
00675 INT fasp_solver_dblc_pgmres (dBLcMat      *A,
00676                             dvector      *b,
00677                             dvector      *x,
00678                             precondition *pc,
00679                             const REAL   tol,
00680                             const INT    MaxIt,
00681                             const SHORT  restart,
00682                             const SHORT  StopType,
00683                             const SHORT  PrtLvl)
00684 {
00685     const INT    n      = b->row;
00686     const INT    MIN_ITER = 0;
00687
00688     // local variables
00689     INT          iter = 0;
00690     int          i, j, k; // must be signed! -zcs
00691
00692     REAL         r_norm, r_normb, gamma, t;
00693     REAL         absres0 = BIGREAL, absres = BIGREAL;
00694     REAL         relres  = BIGREAL, normu  = BIGREAL;
00695
00696     // allocate temp memory (need about (restart+4)*n REAL numbers)
00697     REAL         *c = NULL, *s = NULL, *rs = NULL;
00698     REAL         *norms = NULL, *r = NULL, *w = NULL;
00699     REAL         *work = NULL;
00700     REAL         **p = NULL, **hh = NULL;
00701
00702     INT          Restart = MIN(restart, MaxIt);
00703     INT          Restart1 = Restart + 1;
00704     LONG         worksize = (Restart+4)*(Restart+n)+1-n;
00705
00706     /* allocate memory and setup temp work space */
00707     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00708
00709     // Output some info for debugging
00710     if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (BLC) ...\n");
00711
00712     #if DEBUG_MODE > 0
00713     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00714     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00715     #endif
00716
00717     /* check whether memory is enough for GMRES */
00718     while ( (work == NULL) && (Restart > 5) ) {
00719         Restart = Restart - 5;
00720         Restart1 = Restart + 1;
00721         worksize = (Restart+4)*(Restart+n)+1-n;
00722         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00723     }
00724
00725     if ( work == NULL ) {
00726         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00727         fasp_chker(ERROR_ALLOC_MEM, __FUNCTION__);
00728     }
00729
00730     if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00731         printf("### WARNING: GMRES restart number set to %d!\n", Restart);
00732     }
00733
00734     p      = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00735     hh     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00736     norms  = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00737
00738     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00739
00740     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00741
00742     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00743
00744     // compute initial residual: r = b-A*x
00745     fasp_darray_cp(n, b->val, p[0]);
00746     fasp_blas_dblc_aAxy(-1.0, A, x->val, p[0]);
00747     r_norm = fasp_blas_darray_norm2(n, p[0]);
00748

```

```

00749 // compute stopping criteria
00750 switch (StopType) {
00751     case STOP_REL_RES:
00752         absres0 = MAX(SMALLREAL, r_norm);
00753         relres = r_norm/absres0;
00754         break;
00755     case STOP_REL_PRECRES:
00756         if ( pc == NULL )
00757             fasp_darray_cp(n, p[0], r);
00758         else
00759             pc->fct(p[0], r, pc->data);
00760         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
00761         absres0 = MAX(SMALLREAL, r_normb);
00762         relres = r_normb/absres0;
00763         break;
00764     case STOP_MOD_REL_RES:
00765         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00766         absres0 = r_norm;
00767         relres = absres0/normu;
00768         break;
00769     default:
00770         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00771         goto FINISHED;
00772 }
00773
00774 // if initial residual is small, no need to iterate!
00775 if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00776
00777 // output iteration information if needed
00778 fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0.0);
00779
00780 // store initial residual
00781 norms[0] = relres;
00782
00783 /* GMRES(M) outer iteration */
00784 while ( iter < MaxIt && relres > tol ) {
00785     rs[0] = r_norm;
00786
00787     t = 1.0 / r_norm;
00788
00789     fasp_blas_darray_ax(n, t, p[0]);
00790
00791     /* RESTART CYCLE (right-preconditioning) */
00792     i = 0;
00793     while ( i < Restart && iter < MaxIt ) {
00794         i++; iter++;
00795
00796         /* apply preconditioner */
00797         if ( pc == NULL )
00798             fasp_darray_cp(n, p[i-1], r);
00799         else
00800             pc->fct(p[i-1], r, pc->data);
00801
00802         fasp_blas_dblc_mxv(A, r, p[i]);
00803
00804         /* Modified Gram-Schmidt orthogonalization */
00805         for ( j = 0; j < i; j++ ) {
00806             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00807             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00808         }
00809         t = fasp_blas_darray_norm2(n, p[i]);
00810         hh[i][i-1] = t;
00811
00812         if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
00813             t = 1.0/t;
00814             fasp_blas_darray_ax(n, t, p[i]);
00815         }
00816
00817         for ( j = 1; j < i; j++ ) {
00818             t = hh[j-1][i-1];
00819             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00820             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00821         }
00822         t = hh[i][i-1]*hh[i][i-1];
00823         t += hh[i-1][i-1]*hh[i-1][i-1];
00824
00825         gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
00826         c[i-1] = hh[i-1][i-1] / gamma;
00827         s[i-1] = hh[i][i-1] / gamma;
00828     }
00829 }

```

```

00830     rs[i]   = -s[i-1]*rs[i-1];
00831     rs[i-1] = c[i-1]*rs[i-1];
00832     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00833
00834     absres = r_norm = fabs(rs[i]);
00835
00836     relres = absres/absres0;
00837
00838     norms[iter] = relres;
00839
00840     // output iteration information if needed
00841     fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00842               norms[iter]/norms[iter-1]);
00843
00844     // exit restart cycle if reaches tolerance
00845     if ( relres < tol && iter >= MIN_ITER ) break;
00846
00847 } /* end of restart cycle */
00848
00849 /* compute solution, first solve upper triangular system */
00850 rs[i-1] = rs[i-1] / hh[i-1][i-1];
00851 for ( k = i-2; k >= 0; k-- ) {
00852     t = 0.0;
00853     for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
00854     t += rs[k];
00855     rs[k] = t / hh[k][k];
00856 }
00857
00858 fasp_darray_cp(n, p[i-1], w);
00859
00860 fasp_blas_darray_ax(n, rs[i-1], w);
00861
00862 for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00863
00864 /* apply preconditioner */
00865 if ( pc == NULL )
00866     fasp_darray_cp(n, w, r);
00867 else
00868     pc->fct(w, r, pc->data);
00869
00870 fasp_blas_darray_axpy(n, 1.0, r, x->val);
00871
00872 // Check: prevent false convergence
00873 if ( relres < tol && iter >= MIN_ITER ) {
00874
00875     REAL computed_relres = relres;
00876
00877     // compute residual
00878     fasp_darray_cp(n, b->val, r);
00879     fasp_blas_dblc_aAxy(-1.0, A, x->val, r);
00880     r_norm = fasp_blas_darray_norm2(n, r);
00881
00882     switch ( StopType ) {
00883     case STOP_REL_RES:
00884         absres = r_norm;
00885         relres = absres/absres0;
00886         break;
00887     case STOP_REL_PRECRES:
00888         if ( pc == NULL )
00889             fasp_darray_cp(n, r, w);
00890         else
00891             pc->fct(r, w, pc->data);
00892         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00893         relres = absres/absres0;
00894         break;
00895     case STOP_MOD_REL_RES:
00896         absres = r_norm;
00897         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00898         relres = absres/normu;
00899         break;
00900     }
00901
00902     norms[iter] = relres;
00903
00904     if ( relres < tol ) {
00905         break;
00906     }
00907     else { // Need to restart
00908         fasp_darray_cp(n, r, p[0]); i = 0;
00909     }
00910

```

```

00911         if ( PrtLvl >= PRINT_MORE ) {
00912             ITS_COMPRES(computed_relres); ITS_REALRES(relres);
00913         }
00914     } /* end of convergence check */
00915
00916     /* compute residual vector and continue loop */
00917     for ( j = i; j > 0; j-- ) {
00918         rs[j-1] = -s[j-1]*rs[j];
00919         rs[j]   = c[j-1]*rs[j];
00920     }
00921
00922     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00923
00924     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00925
00926     if ( i ) {
00927         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00928         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00929     }
00930
00931     } /* end of main while loop */
00932
00933     FINISHED:
00934     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00935
00936     /*-----
00937     * Clean up workspace
00938     *-----*/
00939     fasp_mem_free(work); work = NULL;
00940     fasp_mem_free(p); p = NULL;
00941     fasp_mem_free(hh); hh = NULL;
00942     fasp_mem_free(norms); norms = NULL;
00943
00944     #if DEBUG_MODE > 0
00945     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00946     #endif
00947
00948     if ( iter >= MaxIt )
00949         return ERROR_SOLVER_MAXIT;
00950     else
00951         return iter;
00952 }
00953
00954
00979 INT fasp_solver_dstr_pgmres (dSTRmat *A,
00980                             dvector *b,
00981                             dvector *x,
00982                             precondition *pc,
00983                             const REAL tol,
00984                             const INT MaxIt,
00985                             const SHORT restart,
00986                             const SHORT StopType,
00987                             const SHORT PrtLvl)
00988 {
00989     const INT n = b->row;
00990     const INT MIN_ITER = 0;
00991
00992     // local variables
00993     INT iter = 0;
00994     int i, j, k; // must be signed! -zcs
00995
00996     REAL r_norm, r_normb, gamma, t;
00997     REAL absres0 = BIGREAL, absres = BIGREAL;
00998     REAL relres = BIGREAL, normu = BIGREAL;
00999
01000     // allocate temp memory (need about (restart+4)*n REAL numbers)
01001     REAL *c = NULL, *s = NULL, *rs = NULL;
01002     REAL *norms = NULL, *r = NULL, *w = NULL;
01003     REAL *work = NULL;
01004     REAL **p = NULL, **hh = NULL;
01005
01006     INT Restart = MIN(restart, MaxIt);
01007     INT Restart1 = Restart + 1;
01008     LONG worksize = (Restart+4)*(Restart+n)+1-n;
01009
01010     // allocate memory and setup temp work space */
01011     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01012
01013     // Output some info for debugging
01014     if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (STR) ...\n");
01015

```

```

01016 #if DEBUG_MODE > 0
01017     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01018     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01019 #endif
01020
01021     /* check whether memory is enough for GMRES */
01022     while ( (work == NULL) && (Restart > 5) ) {
01023         Restart = Restart - 5;
01024         Restart1 = Restart + 1;
01025         worksize = (Restart+4)*(Restart+n)+1-n;
01026         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01027     }
01028
01029     if ( work == NULL ) {
01030         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
01031         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01032     }
01033
01034     if ( PrtLvl > PRINT_MIN && Restart < restart ) {
01035         printf("### WARNING: GMRES restart number set to %d!\n", Restart);
01036     }
01037
01038     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01039     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01040     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01041
01042     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
01043
01044     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
01045
01046     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
01047
01048     // compute initial residual: r = b-A*x
01049     fasp_darray_cp(n, b->val, p[0]);
01050     fasp_blas_dstr_aAxy(-1.0, A, x->val, p[0]);
01051     r_norm = fasp_blas_darray_norm2(n,p[0]);
01052
01053     // compute stopping criteria
01054     switch (StopType) {
01055         case STOP_REL_RES:
01056             absres0 = MAX(SMALLREAL, r_norm);
01057             relres = r_norm/absres0;
01058             break;
01059         case STOP_REL_PRECRES:
01060             if ( pc == NULL )
01061                 fasp_darray_cp(n, p[0], r);
01062             else
01063                 pc->fct(p[0], r, pc->data);
01064             r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
01065             absres0 = MAX(SMALLREAL, r_normb);
01066             relres = r_normb/absres0;
01067             break;
01068         case STOP_MOD_REL_RES:
01069             normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
01070             absres0 = r_norm;
01071             relres = absres0/normu;
01072             break;
01073         default:
01074             printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01075             goto FINISHED;
01076     }
01077
01078     // if initial residual is small, no need to iterate!
01079     if ( relres < tol || absres0 < 1e-312*tol ) goto FINISHED;
01080
01081     // output iteration information if needed
01082     fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0.0);
01083
01084     // store initial residual
01085     norms[0] = relres;
01086
01087     /* GMRES(M) outer iteration */
01088     while ( iter < MaxIt && relres > tol ) {
01089         rs[0] = r_norm;
01090
01091         t = 1.0 / r_norm;
01092
01093         fasp_blas_darray_ax(n, t, p[0]);
01094
01095         /* RESTART CYCLE (right-preconditioning) */

```

```

01097     i = 0;
01098     while ( i < Restart && iter < MaxIt ) {
01099
01100         i++; iter++;
01101
01102         /* apply preconditioner */
01103         if ( pc == NULL )
01104             fasp_darray_cp(n, p[i-1], r);
01105         else
01106             pc->fct(p[i-1], r, pc->data);
01107
01108         fasp_blas_dstr_mxdv(A, r, p[i]);
01109
01110         /* Modified Gram_Schmidt orthogonalization */
01111         for ( j = 0; j < i; j++ ) {
01112             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01113             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01114         }
01115         t = fasp_blas_darray_norm2(n, p[i]);
01116         hh[i][i-1] = t;
01117
01118         if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
01119             t = 1.0/t;
01120             fasp_blas_darray_ax(n, t, p[i]);
01121         }
01122
01123         for ( j = 1; j < i; ++j ) {
01124             t = hh[j-1][i-1];
01125             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01126             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01127         }
01128         t = hh[i][i-1]*hh[i][i-1];
01129         t += hh[i-1][i-1]*hh[i-1][i-1];
01130
01131         gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
01132         c[i-1] = hh[i-1][i-1] / gamma;
01133         s[i-1] = hh[i][i-1] / gamma;
01134         rs[i] = -s[i-1]*rs[i-1];
01135         rs[i-1] = c[i-1]*rs[i-1];
01136         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01137
01138         absres = r_norm = fabs(rs[i]);
01139
01140         relres = absres/absres0;
01141
01142         norms[iter] = relres;
01143
01144         // output iteration information if needed
01145         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
01146                 norms[iter]/norms[iter-1]);
01147
01148         // exit restart cycle if reaches tolerance
01149         if ( relres < tol && iter >= MIN_ITER ) break;
01150
01151     } /* end of restart cycle */
01152
01153     /* compute solution, first solve upper triangular system */
01154     rs[i-1] = rs[i-1] / hh[i-1][i-1];
01155     for ( k = i-2; k >= 0; k-- ) {
01156         t = 0.0;
01157         for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
01158         t += rs[k];
01159         rs[k] = t / hh[k][k];
01160     }
01161
01162     fasp_darray_cp(n, p[i-1], w);
01163
01164     fasp_blas_darray_ax(n, rs[i-1], w);
01165
01166     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01167
01168     /* apply preconditioner */
01169     if ( pc == NULL )
01170         fasp_darray_cp(n, w, r);
01171     else
01172         pc->fct(w, r, pc->data);
01173
01174     fasp_blas_darray_axpy(n, 1.0, r, x->val);
01175
01176     // Check: prevent false convergence
01177     if ( relres < tol && iter >= MIN_ITER ) {

```

```

01178
01179     REAL computed_relres = relres;
01180
01181     // compute residual
01182     fasp_darray_cp(n, b->val, r);
01183     fasp_blas_dstr_aApy(-1.0, A, x->val, r);
01184     r_norm = fasp_blas_darray_norm2(n, r);
01185
01186     switch ( StopType ) {
01187     case STOP_REL_RES:
01188         absres = r_norm;
01189         relres = absres/absres0;
01190         break;
01191     case STOP_REL_PRECRES:
01192         if ( pc == NULL )
01193             fasp_darray_cp(n, r, w);
01194         else
01195             pc->fct(r, w, pc->data);
01196         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
01197         relres = absres/absres0;
01198         break;
01199     case STOP_MOD_REL_RES:
01200         absres = r_norm;
01201         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
01202         relres = absres/normu;
01203         break;
01204     }
01205
01206     norms[iter] = relres;
01207
01208     if ( relres < tol ) {
01209         break;
01210     }
01211     else { // Need to restart
01212         fasp_darray_cp(n, r, p[0]); i = 0;
01213     }
01214
01215     if ( PrtLvl >= PRINT_MORE ) {
01216         ITS_COMPRES(computed_relres); ITS_REALRES(relres);
01217     }
01218
01219     } /* end of convergence check */
01220
01221     /* compute residual vector and continue loop */
01222     for ( j = i; j > 0; j-- ) {
01223         rs[j-1] = -s[j-1]*rs[j];
01224         rs[j] = c[j-1]*rs[j];
01225     }
01226
01227     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01228
01229     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01230
01231     if ( i ) {
01232         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01233         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01234     }
01235
01236     } /* end of main while loop */
01237
01238     FINISHED:
01239     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01240
01241     /*-----
01242     * Clean up workspace
01243     *-----*/
01244     fasp_mem_free(work); work = NULL;
01245     fasp_mem_free(p); p = NULL;
01246     fasp_mem_free(hh); hh = NULL;
01247     fasp_mem_free(norms); norms = NULL;
01248
01249     #if DEBUG_MODE > 0
01250     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01251     #endif
01252
01253     if ( iter >= MaxIt )
01254         return ERROR_SOLVER_MAXIT;
01255     else
01256         return iter;
01257 }
01258

```



```

01283 INT fasp_solver_pgmres (mxv_matfree *mf,
01284                          dvector *b,
01285                          dvector *x,
01286                          precondition *pc,
01287                          const REAL tol,
01288                          const INT MaxIt,
01289                          const SHORT restart,
01290                          const SHORT StopType,
01291                          const SHORT PrtLvl)
01292 {
01293     const INT n = b->row;
01294     const INT min_iter = 0;
01295
01296     // local variables
01297     INT iter = 0;
01298     int i, j, k; // must be signed! -zcs
01299
01300     REAL epsmac = SMALLREAL;
01301     REAL r_norm, b_norm, den_norm;
01302     REAL epsilon, gamma, t;
01303
01304     // allocate temp memory (need about (restart+4)*n REAL numbers)
01305     REAL *c = NULL, *s = NULL, *rs = NULL;
01306     REAL *norms = NULL, *r = NULL, *w = NULL;
01307     REAL *work = NULL;
01308     REAL **p = NULL, **hh = NULL;
01309
01310     INT Restart = restart;
01311     INT Restart1 = Restart + 1;
01312     LONG worksize = (Restart+4)*(Restart+n)+1-n;
01313
01314     // Output some info for debugging
01315     if (PrtLvl > PRINT_NONE) printf("\nCalling GMRes solver (MatFree) ...\n");
01316
01317     #if DEBUG_MODE > 0
01318     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01319     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01320     #endif
01321
01322     /* allocate memory and setup temp work space */
01323     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01324
01325     /* check whether memory is enough for GMRES */
01326     while ( (work == NULL) && (Restart > 5) ) {
01327         Restart = Restart - 5;
01328         worksize = (Restart+4)*(Restart+n)+1-n;
01329         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01330         Restart1 = Restart + 1;
01331     }
01332
01333     if (work == NULL) {
01334         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
01335         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01336     }
01337
01338     if (PrtLvl > PRINT_MIN && Restart < restart) {
01339         printf("### WARNING: GMRES restart number set to %d!\n", Restart);
01340     }
01341
01342     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01343     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01344     norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01345
01346     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
01347
01348     for (i = 0; i < Restart1; i++) p[i] = s + Restart + i*n;
01349
01350     for (i = 0; i < Restart1; i++) hh[i] = p[Restart] + n + i*Restart;
01351
01352     /* initialization */
01353     mf->fct(mf->data, x->val, p[0]);
01354     fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, p[0]);
01355
01356     b_norm = fasp_blas_darray_norm2(n, b->val);
01357     r_norm = fasp_blas_darray_norm2(n, p[0]);
01358
01359     if (PrtLvl > PRINT_NONE) {
01360         norms[0] = r_norm;
01361         if (PrtLvl >= PRINT_SOME) {
01362             ITS_PUTNORM("right-hand side", b_norm);
01363             ITS_PUTNORM("residual", r_norm);

```

```

01364     }
01365 }
01366
01367 if (b_norm > 0.0) den_norm = b_norm;
01368 else den_norm = r_norm;
01369
01370 epsilon = tol*den_norm;
01371
01372 /* outer iteration cycle */
01373 while (iter < MaxIt) {
01374
01375     rs[0] = r_norm;
01376     if (r_norm == 0.0) {
01377         fasp_mem_free(work); work = NULL;
01378         fasp_mem_free(p); p = NULL;
01379         fasp_mem_free(hh); hh = NULL;
01380         fasp_mem_free(norms); norms = NULL;
01381         return iter;
01382     }
01383
01384     if (r_norm <= epsilon && iter >= min_iter) {
01385         mf->fct(mf->data, x->val, r);
01386         fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
01387         r_norm = fasp_blas_darray_norm2(n, r);
01388
01389         if (r_norm <= epsilon) {
01390             break;
01391         }
01392         else {
01393             if (PrtLvl >= PRINT_SOME) ITS_FACONV;
01394         }
01395     }
01396
01397     t = 1.0 / r_norm;
01398     //for (j = 0; j < n; j++) p[0][j] *= t;
01399     fasp_blas_darray_ax(n, t, p[0]);
01400
01401     /* RESTART CYCLE (right-preconditioning) */
01402     i = 0;
01403     while (i < Restart && iter < MaxIt) {
01404
01405         i++; iter++;
01406
01407         /* apply preconditioner */
01408         if (pc == NULL)
01409             fasp_darray_cp(n, p[i-1], r);
01410         else
01411             pc->fct(p[i-1], r, pc->data);
01412
01413         mf->fct(mf->data, r, p[i]);
01414
01415         /* modified Gram_Schmidt */
01416         for (j = 0; j < i; j++) {
01417             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01418             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01419         }
01420         t = fasp_blas_darray_norm2(n, p[i]);
01421         hh[i][i-1] = t;
01422         if (t != 0.0) {
01423             t = 1.0/t;
01424             //for (j = 0; j < n; j++) p[i][j] *= t;
01425             fasp_blas_darray_ax(n, t, p[i]);
01426         }
01427
01428         for (j = 1; j < i; ++j) {
01429             t = hh[j-1][i-1];
01430             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01431             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01432         }
01433         t = hh[i][i-1]*hh[i][i-1];
01434         t += hh[i-1][i-1]*hh[i-1][i-1];
01435         gamma = sqrt(t);
01436         if (gamma == 0.0) gamma = epsmac;
01437         c[i-1] = hh[i-1][i-1] / gamma;
01438         s[i-1] = hh[i][i-1] / gamma;
01439         rs[i] = -s[i-1]*rs[i-1];
01440         rs[i-1] = c[i-1]*rs[i-1];
01441         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01442         r_norm = fabs(rs[i]);
01443
01444         norms[iter] = r_norm;

```

```

01445
01446         if (b_norm > 0 ) {
01447             fasp_itinfo(PrtLvl,StopType,iter,norms[iter]/b_norm,
01448                 norms[iter],norms[iter]/norms[iter-1]);
01449         }
01450         else {
01451             fasp_itinfo(PrtLvl,StopType,iter,norms[iter],norms[iter],
01452                 norms[iter]/norms[iter-1]);
01453         }
01454
01455         /* should we exit restart cycle? */
01456         if (r_norm <= epsilon && iter >= min_iter) {
01457             break;
01458         }
01459     } /* end of restart cycle */
01460
01461     /* now compute solution, first solve upper triangular system */
01462     rs[i-1] = rs[i-1] / hh[i-1][i-1];
01463     for (k = i-2; k >= 0; k --) {
01464         t = 0.0;
01465         for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];
01466
01467         t += rs[k];
01468         rs[k] = t / hh[k][k];
01469     }
01470     fasp_darray_cp(n, p[i-1], w);
01471     //for (j = 0; j < n; j ++) w[j] *= rs[i-1];
01472     fasp_blas_darray_ax(n, rs[i-1], w);
01473     for (j = i-2; j >= 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01474
01475     /* apply preconditioner */
01476     if (pc == NULL)
01477         fasp_darray_cp(n, w, r);
01478     else
01479         pc->fct(w, r, pc->data);
01480
01481     fasp_blas_darray_axpy(n, 1.0, r, x->val);
01482
01483     if (r_norm <= epsilon && iter >= min_iter) {
01484         mf->fct(mf->data, x->val, r);
01485         fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
01486         r_norm = fasp_blas_darray_norm2(n, r);
01487
01488         if (r_norm <= epsilon) {
01489             break;
01490         }
01491         else {
01492             if (PrtLvl >= PRINT_SOME) ITS_FACONV;
01493             fasp_darray_cp(n, r, p[0]); i = 0;
01494         }
01495     } /* end of convergence check */
01496
01497     /* compute residual vector and continue loop */
01498     for (j = i; j > 0; j--) {
01499         rs[j-1] = -s[j-1]*rs[j];
01500         rs[j] = c[j-1]*rs[j];
01501     }
01502
01503     if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01504
01505     for (j = i-1; j > 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01506
01507     if (i) {
01508         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01509         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01510     }
01511 } /* end of iteration while loop */
01512
01513 if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);
01514
01515 /*-----
01516 * Clean up workspace
01517 *-----*/
01518 fasp_mem_free(work);    work = NULL;
01519 fasp_mem_free(p);       p = NULL;
01520 fasp_mem_free(hh);      hh = NULL;
01521 fasp_mem_free(norms);   norms = NULL;
01522
01523 #if DEBUG_MODE > 0
01524 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01525 #endif

```

```

01526
01527     if (iter>=MaxIt)
01528         return ERROR_SOLVER_MAXIT;
01529     else
01530         return iter;
01531 }
01532
01533 #if 0
01534 static double estimate_spectral_radius (const double **A, int n, size_t k = 20)
01535 {
01536     double *x = (double *)malloc(n* sizeof(double));
01537     double *y = (double *)malloc(n* sizeof(double));
01538     double *z = (double *)malloc(n* sizeof(double));
01539     double t;
01540     int il,jl;
01541
01542     // initialize x to random values in [0,1)
01543     // cusp::copy(cusp::detail::random_reals<ValueType>(N), x);
01544     dvector px;
01545     px.row = n;
01546     px.val = x;
01547
01548     fasp_dvec_rand(n, &px);
01549
01550     for(size_t i = 0; i < k; i++)
01551     {
01552         //cusp::blas::scal(x, ValueType(1.0) / cusp::blas::nrmmx(x));
01553         t= 1.0/ fasp_blas_darray_norminf(n, px);
01554         for(il= 0; il <n; il++) x[il] *= t;
01555
01556         //cusp::multiply(A, x, y);
01557
01558         for(il= 0; il <n; il++) {
01559             t= 0.0
01560             for(jl= 0; jl <n; jl++) t += A[il][jl] * x[jl];
01561             y[il] = t;
01562             // x.swap(y);
01563             for(il= 0; il <n; il++) z[il] = x[il];
01564             for(il= 0; il <n; il++) x[il] = y[il];
01565             for(il= 0; il <n; il++) y[il] = z[il];
01566         }
01567
01568         free(x);
01569         free(y);
01570         free(z);
01571
01572         if (k == 0)
01573             return 0;
01574         else
01575             //return cusp::blas::nrm2(x) / cusp::blas::nrm2(y);
01576             return fasp_blas_darray_norm2(n,x) / fasp_blas_darray_norm2(n,y) ;
01577     }
01578
01579 static double spectral_radius (dCSRmat *A,
01580                               const SHORT restart)
01581 {
01582     const INT n          = A->row;
01583     const INT MIN_ITER   = 0;
01584
01585     // local variables
01586     INT      iter = 0;
01587     INT      Restart1 = restart + 1;
01588     INT      i, j, k;
01589
01590     REAL      r_norm, den_norm;
01591     REAL      epsilon, gamma, t;
01592
01593     REAL      *c = NULL, *s = NULL, *rs = NULL;
01594     REAL      *norms = NULL, *r = NULL, *w = NULL;
01595     REAL      **p = NULL, **hh = NULL;
01596     REAL      *work = NULL;
01597
01598     /* allocate memory */
01599     work = (REAL *)fasp_mem_calloc((restart+4)*(restart+n)+1-n, sizeof(REAL));
01600     p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01601     hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01602
01603     norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01604
01605     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + restart;
01606

```

```

01607     for (i = 0; i < Restart1; i++) p[i] = s + restart + i*n;
01608     for (i = 0; i < Restart1; i++) hh[i] = p[restart] + n + i*restart;
01609
01610     /* initialization */
01611     dvector p0;
01612     p0.row = n;
01613     p0.val = p[0];
01614     fasp_dvec_rand(n, &p0);
01615
01616     r_norm = fasp_blas_darray_norm2(n, p[0]);
01617     t = 1.0 / r_norm;
01618     for (j = 0; j < n; j++) p[0][j] *= t;
01619
01620     int maxiter = MIN(n, restart);
01621     for (j = 0; j < maxiter; j++) {
01622         fasp_blas_bdsr_mxv(A, p[j], p[j+1]);
01623
01624         for( i = 0; i <= j; i++ ) {
01625             hh[i][j] = fasp_blas_darray_dotprod(n, p[i], p[j+1]);
01626             fasp_blas_darray_axpy(n, -hh[i][j], p[i], p[ j+1 ]);
01627         }
01628
01629         hh[j+1][j] = fasp_blas_darray_norm2 (n, p[j+1]);
01630         if ( hh[j+1][j] < 1e-10) break;
01631         t = 1.0/hh[j+1][j];
01632         for (k = 0; k < n; k++) p[j+1][k] *= t;
01633     }
01634
01635     H = (REAL **)fasp_mem_calloc(j, sizeof(REAL *));
01636     H[0] = (REAL *)fasp_mem_calloc(j*j, sizeof(REAL));
01637     for (i = 1; i < j; i++) H[i] = H[i-1] + j;
01638
01639
01640     for( size_t row = 0; row < j; row++ )
01641         for( size_t col = 0; col < j; col++ )
01642             H[row][col] = hh[row][col];
01643
01644     double spectral_radius = estimate_spectral_radius( H, j, 20);
01645
01646     /*-----
01647 * Clean up workspace
01648 *-----*/
01649     fasp_mem_free(work); work = NULL;
01650     fasp_mem_free(p); p = NULL;
01651     fasp_mem_free(hh); hh = NULL;
01652     fasp_mem_free(norms); norms = NULL;
01653     fasp_mem_free(H[0]); H[0] = NULL;
01654     fasp_mem_free(H); H = NULL;
01655
01656     return spectral_radius;
01657 }
01658 #endif
01659
01660 /*-----*/
01661 /*--      End of File      --*/
01662 /*-----*/

```

9.119 KryPminres.c File Reference

Krylov subspace methods – Preconditioned minimal residual.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"
#include "KryUtil.inl"

```

Functions

- `INT fasp_solver_dcsr_pminres (dCSRmat *A, dvector *b, dvector *u, precondition *pc, const REAL tol, const INT MaxIt, const SHORT StopType, const SHORT PrtLvl)`

A preconditioned minimal residual (Minres) method for solving $Au=b$.

- `INT fasp_solver_dblc_pminres` (`dBLCmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
A preconditioned minimal residual (Minres) method for solving $Au=b$.
- `INT fasp_solver_dstr_pminres` (`dSTRmat *A`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
A preconditioned minimal residual (Minres) method for solving $Au=b$.
- `INT fasp_solver_pminres` (`mxv_matfree *mf`, `dvector *b`, `dvector *u`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT StopType`, `const SHORT PrtLvl`)
A preconditioned minimal residual (Minres) method for solving $Au=b$.

9.119.1 Detailed Description

Krylov subspace methods – Preconditioned minimal residual.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c.o](#)

See [KrySPminres.c](#) for a safer version

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2012–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPminres.c](#).

9.119.2 Function Documentation

9.119.2.1 fasp_solver_dblc_pminres()

```
INT fasp_solver_dblc_pminres (
    dBLCmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBLCmat : coefficient matrix
<i>b</i>	Pointer to dvector : right hand side
<i>u</i>	Pointer to dvector : unknowns
<i>pc</i>	Pointer to precond : structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

05/01/2012

Rewritten based on the original version by Xiaozhe Hu 05/24/2010 Modified by Chensong Zhang on 04/09/2013
Definition at line 475 of file [KryPminres.c](#).

9.119.2.2 fasp_solver_dcsr_pminres()

```
INT fasp_solver_dcsr_pminres (
    dCSRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

05/01/2012

Rewritten based on the original version by Shiquan Zhang 05/10/2010 Modified by Chensong Zhang on 04/09/2013
Definition at line 62 of file [KryPminres.c](#).

9.119.2.3 fasp_solver_dstr_pminres()

```

INT fasp_solver_dstr_pminres (
    dSTRmat * A,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

A preconditioned minimal residual (Minres) method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/09/2013

Definition at line 885 of file [KryPminres.c](#).

9.119.2.4 fasp_solver_pminres()

```

INT fasp_solver_pminres (
    mxv_matfree * mf,
    dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

A preconditioned minimal residual (Minres) method for solving $Au=b$.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Shiquan Zhang

Date

10/24/2010

Rewritten by Chensong Zhang on 05/01/2012

Definition at line [1296](#) of file [KryPminres.c](#).

9.120 KryPminres.c

[Go to the documentation of this file.](#)

```

00001
00023 #include <math.h>
00024
00025 #include "fasp.h"
00026 #include "fasp_funcs.h"
00027
00028 /*-----*/
00029 /*--  Declare Private Functions  --*/
00030 /*-----*/
00031
00032 #include "KryUtil.inl"
00033
00034 /*-----*/
00035 /*--          Public Functions          --*/
00036 /*-----*/
00037
00062 INT fasp_solver_dcsr_pminres (dCSRmat      *A,
00063                             dvector      *b,
00064                             dvector      *u,
00065                             precondition *pc,
00066                             const REAL   tol,
00067                             const INT     MaxIt,
00068                             const SHORT   StopType,
00069                             const SHORT   PrtLvl)
00070 {
00071     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00072     const INT    m = b->row;
00073     const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
00074     const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
00075
00076     // local variables
00077     INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
00078     REAL          absres0 = BIGREAL, absres = BIGREAL;
00079     REAL          normr0  = BIGREAL, relres = BIGREAL;
00080     REAL          normu2, normuu, normp, infnormu, factor;

```

```

00081     REAL      alpha, alpha0, alpha1, temp2;
00082
00083     // allocate temp memory (need 11*m REAL)
00084     REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
00085     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
00086     REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;
00087
00088     // Output some info for debugging
00089     if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (CSR) ...\n");
00090
00091     #if DEBUG_MODE > 0
00092     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00093     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00094     #endif
00095
00096     // p0 = 0
00097     fasp_darray_set(m,p0,0.0);
00098
00099     // r = b-A*u
00100     fasp_darray_cp(m,b->val,r);
00101     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00102
00103     // p1 = B(r)
00104     if ( pc != NULL )
00105         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00106     else
00107         fasp_darray_cp(m,r,p1); /* No preconditioner */
00108
00109     // compute initial residuals
00110     switch ( StopType ) {
00111     case STOP_REL_RES:
00112         absres0 = fasp_blas_darray_norm2(m,r);
00113         normr0  = MAX(SMALLREAL,absres0);
00114         relres  = absres0/normr0;
00115         break;
00116     case STOP_REL_PRECRES:
00117         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
00118         normr0  = MAX(SMALLREAL,absres0);
00119         relres  = absres0/normr0;
00120         break;
00121     case STOP_MOD_REL_RES:
00122         absres0 = fasp_blas_darray_norm2(m,r);
00123         normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00124         relres  = absres0/normu2;
00125         break;
00126     default:
00127         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00128         goto FINISHED;
00129     }
00130
00131     // if initial residual is small, no need to iterate!
00132     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00133
00134     // output iteration information if needed
00135     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00136
00137     // tp = A*p1
00138     fasp_blas_dcsr_mxv(A,p1,tp);
00139
00140     // tz = B(tp)
00141     if ( pc != NULL )
00142         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00143     else
00144         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00145
00146     // p1 = p1/normp
00147     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00148     normp = sqrt(normp);
00149     fasp_darray_cp(m,p1,t);
00150     fasp_darray_set(m,p1,0.0);
00151     fasp_blas_darray_axpy(m,1/normp,t,p1);
00152
00153     // t0 = A*p0 = 0
00154     fasp_darray_set(m,t0,0.0);
00155     fasp_darray_cp(m,t0,z0);
00156     fasp_darray_cp(m,t0,t1);
00157     fasp_darray_cp(m,t0,z1);
00158
00159     // t1 = tp/normp, z1 = tz/normp
00160     fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
00161     fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

```

```

00162
00163 // main MinRes loop
00164 while ( iter++ < MaxIt ) {
00165
00166     // alpha = <r,z1>
00167     alpha=fasp_blas_darray_dotprod(m,r,z1);
00168
00169     // u = u+alpha*p1
00170     fasp_blas_darray_axpy(m,alpha,p1,u->val);
00171
00172     // r = r-alpha*Ap1
00173     fasp_blas_darray_axpy(m,-alpha,t1,r);
00174
00175     // compute t = A*z1 alpha1 = <z1,t>
00176     fasp_blas_dcsr_mxv(A,z1,t);
00177     alpha1=fasp_blas_darray_dotprod(m,z1,t);
00178
00179     // compute t = A*z0 alpha0 = <z1,t>
00180     fasp_blas_dcsr_mxv(A,z0,t);
00181     alpha0=fasp_blas_darray_dotprod(m,z1,t);
00182
00183     // p2 = z1-alpha1*p1-alpha0*p0
00184     fasp_darray_cp(m,z1,p2);
00185     fasp_blas_darray_axpy(m,-alpha1,p1,p2);
00186     fasp_blas_darray_axpy(m,-alpha0,p0,p2);
00187
00188     // tp = A*p2
00189     fasp_blas_dcsr_mxv(A,p2,tp);
00190
00191     // tz = B(tp)
00192     if ( pc != NULL )
00193         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00194     else
00195         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00196
00197     // p2 = p2/normp
00198     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00199     normp = sqrt(normp);
00200     fasp_darray_cp(m,p2,t);
00201     fasp_darray_set(m,p2,0.0);
00202     fasp_blas_darray_axpy(m,1/normp,t,p2);
00203
00204     // prepare for next iteration
00205     fasp_darray_cp(m,p1,p0);
00206     fasp_darray_cp(m,p2,p1);
00207     fasp_darray_cp(m,t1,t0);
00208     fasp_darray_cp(m,z1,z0);
00209
00210     // t1=tp/normp,z1=tz/normp
00211     fasp_darray_set(m,t1,0.0);
00212     fasp_darray_cp(m,t1,z1);
00213     fasp_blas_darray_axpy(m,1/normp,tp,t1);
00214     fasp_blas_darray_axpy(m,1/normp,tz,z1);
00215
00216     normu2 = fasp_blas_darray_norm2(m,u->val);
00217
00218     // compute residuals
00219     switch ( StopType ) {
00220     case STOP_REL_RES:
00221         temp2 = fasp_blas_darray_dotprod(m,r,r);
00222         absres = sqrt(temp2);
00223         relres = absres/normr0;
00224         break;
00225     case STOP_REL_PRECRES:
00226         if (pc == NULL)
00227             fasp_darray_cp(m,r,t);
00228         else
00229             pc->fct(r,t,pc->data);
00230         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00231         absres = sqrt(temp2);
00232         relres = absres/normr0;
00233         break;
00234     case STOP_MOD_REL_RES:
00235         temp2 = fasp_blas_darray_dotprod(m,r,r);
00236         absres = sqrt(temp2);
00237         relres = absres/normu2;
00238         break;
00239     }
00240
00241     // compute reduction factor of residual ||r||
00242     factor = absres/absres0;

```

```

00243
00244 // output iteration information if needed
00245 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00246
00247 if ( factor > 0.9 ) { // Only check when converge slowly
00248
00249     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00250     infnormu = fasp_blas_darray_norminf(m, u->val);
00251     if (infnormu <= sol_inf_tol) {
00252         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00253         iter = ERROR_SOLVER_SOLSTAG;
00254         break;
00255     }
00256
00257     // Check II: if staggenated, try to restart
00258     normuu = fasp_blas_darray_norm2(m,p1);
00259     normuu = ABS(alpha)*(normuu/normu2);
00260
00261     if ( normuu < maxdiff ) {
00262
00263         if ( stag < MaxStag ) {
00264             if ( PrtLvl >= PRINT_MORE ) {
00265                 ITS_DIFFRES(normuu,relres);
00266                 ITS_RESTART;
00267             }
00268         }
00269
00270         fasp_darray_cp(m,b->val,r);
00271         fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00272
00273         // compute residuals
00274         switch (StopType) {
00275             case STOP_REL_RES:
00276                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00277                 absres = sqrt(temp2);
00278                 relres = absres/normr0;
00279                 break;
00280             case STOP_REL_PRECRES:
00281                 if (pc == NULL)
00282                     fasp_darray_cp(m,r,t);
00283                 else
00284                     pc->fct(r,t,pc->data);
00285                 temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00286                 absres = sqrt(temp2);
00287                 relres = absres/normr0;
00288                 break;
00289             case STOP_MOD_REL_RES:
00290                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00291                 absres = sqrt(temp2);
00292                 relres = absres/normu2;
00293                 break;
00294         }
00295
00296         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00297
00298         if ( relres < tol )
00299             break;
00300         else {
00301             if ( stag >= MaxStag ) {
00302                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00303                 iter = ERROR_SOLVER_STAG;
00304                 break;
00305             }
00306             fasp_darray_set(m,p0,0.0);
00307             ++stag;
00308             ++restart_step;
00309
00310             // p1 = B(r)
00311             if ( pc != NULL )
00312                 pc->fct(r,p1,pc->data); /* Apply preconditioner */
00313             else
00314                 fasp_darray_cp(m,r,p1); /* No preconditioner */
00315
00316             // tp = A*p1
00317             fasp_blas_dcsr_mxx(A,p1,tp);
00318
00319             // tz = B(tp)
00320             if ( pc != NULL )
00321                 pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00322             else
00323                 fasp_darray_cp(m,tp,tz); /* No preconditioner */

```

```

00324
00325         // p1 = p1/normp
00326         normp = fasp_blas_darray_dotprod(m,tz,tp);
00327         normp = sqrt(normp);
00328         fasp_darray_cp(m,p1,t);
00329
00330         // t0 = A*p0=0
00331         fasp_darray_set(m,t0,0.0);
00332         fasp_darray_cp(m,t0,z0);
00333         fasp_darray_cp(m,t0,t1);
00334         fasp_darray_cp(m,t0,z1);
00335         fasp_darray_cp(m,t0,p1);
00336
00337         fasp_blas_darray_axpy(m,1/normp,t,p1);
00338
00339         // t1 = tp/normp, z1 = tz/normp
00340         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00341         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00342     }
00343 }
00344
00345 } // end of check I and II
00346
00347 // Check III: prevent false convergence
00348 if ( relres < tol ) {
00349
00350     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00351
00352     // compute residual r = b - Ax again
00353     fasp_darray_cp(m,b->val,r);
00354     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00355
00356     // compute residuals
00357     switch (StopType) {
00358     case STOP_REL_RES:
00359         temp2 = fasp_blas_darray_dotprod(m,r,r);
00360         absres = sqrt(temp2);
00361         relres = absres/normr0;
00362         break;
00363     case STOP_REL_PRECRES:
00364         if (pc == NULL)
00365             fasp_darray_cp(m,r,t);
00366         else
00367             pc->fct(r,t,pc->data);
00368         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00369         absres = sqrt(temp2);
00370         relres = absres/normr0;
00371         break;
00372     case STOP_MOD_REL_RES:
00373         temp2 = fasp_blas_darray_dotprod(m,r,r);
00374         absres = sqrt(temp2);
00375         relres = absres/normu2;
00376         break;
00377     }
00378
00379     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00380
00381     // check convergence
00382     if ( relres < tol ) break;
00383
00384     if ( more_step >= MaxRestartStep ) {
00385         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00386         iter = ERROR_SOLVER_TOLSMALL;
00387         break;
00388     }
00389
00390     // prepare for restarting method
00391     fasp_darray_set(m,p0,0.0);
00392     ++more_step;
00393     ++restart_step;
00394
00395     // p1 = B(r)
00396     if ( pc != NULL )
00397         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00398     else
00399         fasp_darray_cp(m,r,p1); /* No preconditioner */
00400
00401     // tp = A*p1
00402     fasp_blas_dcsr_mxv(A,p1,tp);
00403
00404     // tz = B(tp)

```

```

00405         if ( pc != NULL )
00406             pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00407         else
00408             fasp_darray_cp(m,tp,tz); /* No preconditioner */
00409
00410         // p1 = p1/normp
00411         normp = fasp_blas_darray_dotprod(m,tz,tp);
00412         normp = sqrt(normp);
00413         fasp_darray_cp(m,p1,t);
00414
00415         // t0 = A*p0 = 0
00416         fasp_darray_set(m,t0,0.0);
00417         fasp_darray_cp(m,t0,z0);
00418         fasp_darray_cp(m,t0,t1);
00419         fasp_darray_cp(m,t0,z1);
00420         fasp_darray_cp(m,t0,p1);
00421
00422         fasp_blas_darray_axpy(m,1/normp,t,p1);
00423
00424         // t1=tp/normp,z1=tz/normp
00425         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00426         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00427
00428     } // end of convergence check
00429
00430     // update relative residual here
00431     absres0 = absres;
00432
00433 } // end of the main loop
00434
00435 FINISHED: // finish iterative method
00436 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00437
00438 // clean up temp memory
00439 fasp_mem_free(work); work = NULL;
00440
00441 #if DEBUG_MODE > 0
00442     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00443 #endif
00444
00445 if ( iter > MaxIt )
00446     return ERROR_SOLVER_MAXIT;
00447 else
00448     return iter;
00449 }
00450
00475 INT fasp_solver_dblc_pminres (dBLCmat      *A,
00476                             dvector      *b,
00477                             dvector      *u,
00478                             precondition *pc,
00479                             const REAL   tol,
00480                             const INT     MaxIt,
00481                             const SHORT   StopType,
00482                             const SHORT   PrtLvl)
00483 {
00484     const SHORT   MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00485     const INT      m = b->row;
00486     const REAL     maxdiff = tol*STAG_RATIO; // stagnation tolerance
00487     const REAL     sol_inf_tol = SMALLREAL; // infinity norm tolerance
00488
00489     // local variables
00490     INT            iter = 0, stag = 1, more_step = 1, restart_step = 1;
00491     REAL           absres0 = BIGREAL, absres = BIGREAL;
00492     REAL           normr0 = BIGREAL, relres = BIGREAL;
00493     REAL           normu2, normu, normp, infnormu, factor;
00494     REAL           alpha, alpha0, alphas, temp2;
00495
00496     // allocate temp memory (need 11*m REAL)
00497     REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
00498     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
00499     REAL *t0=z1+m, *t1=t0+m, *t2=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;
00500
00501     // Output some info for debugging
00502     if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (BLC) ...\n");
00503
00504 #if DEBUG_MODE > 0
00505     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00506     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00507 #endif
00508
00509     // p0 = 0

```

```

00510     fasp_darray_set(m,p0,0.0);
00511
00512     // r = b-A*u
00513     fasp_darray_cp(m,b->val,r);
00514     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00515
00516     // p1 = B(r)
00517     if ( pc != NULL )
00518         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00519     else
00520         fasp_darray_cp(m,r,p1); /* No preconditioner */
00521
00522     // compute initial residuals
00523     switch ( StopType ) {
00524     case STOP_REL_RES:
00525         absres0 = fasp_blas_darray_norm2(m,r);
00526         normr0 = MAX(SMALLREAL,absres0);
00527         relres = absres0/normr0;
00528         break;
00529     case STOP_REL_PRECRES:
00530         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
00531         normr0 = MAX(SMALLREAL,absres0);
00532         relres = absres0/normr0;
00533         break;
00534     case STOP_MOD_REL_RES:
00535         absres0 = fasp_blas_darray_norm2(m,r);
00536         normu2 = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00537         relres = absres0/normu2;
00538         break;
00539     default:
00540         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00541         goto FINISHED;
00542     }
00543
00544     // if initial residual is small, no need to iterate!
00545     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00546
00547     // output iteration information if needed
00548     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00549
00550     // tp = A*p1
00551     fasp_blas_dblc_mxv(A,p1,tp);
00552
00553     // tz = B(tp)
00554     if ( pc != NULL )
00555         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00556     else
00557         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00558
00559     // p1 = p1/normp
00560     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00561     normp = sqrt(normp);
00562     fasp_darray_cp(m,p1,t);
00563     fasp_darray_set(m,p1,0.0);
00564     fasp_blas_darray_axpy(m,1/normp,t,p1);
00565
00566     // t0 = A*p0 = 0
00567     fasp_darray_set(m,t0,0.0);
00568     fasp_darray_cp(m,t0,z0);
00569     fasp_darray_cp(m,t0,t1);
00570     fasp_darray_cp(m,t0,z1);
00571
00572     // t1 = tp/normp, z1 = tz/normp
00573     fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
00574     fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
00575
00576     // main MinRes loop
00577     while ( iter++ < MaxIt ) {
00578
00579         // alpha = <r,z1>
00580         alpha=fasp_blas_darray_dotprod(m,r,z1);
00581
00582         // u = u+alpha*p1
00583         fasp_blas_darray_axpy(m,alpha,p1,u->val);
00584
00585         // r = r-alpha*Apl
00586         fasp_blas_darray_axpy(m,-alpha,t1,r);
00587
00588         // compute t = A*z1 alpha1 = <z1,t>
00589         fasp_blas_dblc_mxv(A,z1,t);
00590         alpha1=fasp_blas_darray_dotprod(m,z1,t);

```

```

00591
00592 // compute t = A*z0 alpha0 = <z1,t>
00593 fasp_blas_dblc_mxv(A,z0,t);
00594 alpha0=fasp_blas_darray_dotprod(m,z1,t);
00595
00596 // p2 = z1-alpha1*p1-alpha0*p0
00597 fasp_darray_cp(m,z1,p2);
00598 fasp_blas_darray_axpy(m,-alpha1,p1,p2);
00599 fasp_blas_darray_axpy(m,-alpha0,p0,p2);
00600
00601 // tp = A*p2
00602 fasp_blas_dblc_mxv(A,p2,tp);
00603
00604 // tz = B(tp)
00605 if ( pc != NULL )
00606     pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00607 else
00608     fasp_darray_cp(m,tp,tz); /* No preconditioner */
00609
00610 // p2 = p2/normp
00611 normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00612 normp = sqrt(normp);
00613 fasp_darray_cp(m,p2,t);
00614 fasp_darray_set(m,p2,0.0);
00615 fasp_blas_darray_axpy(m,1/normp,t,p2);
00616
00617 // prepare for next iteration
00618 fasp_darray_cp(m,p1,p0);
00619 fasp_darray_cp(m,p2,p1);
00620 fasp_darray_cp(m,t1,t0);
00621 fasp_darray_cp(m,z1,z0);
00622
00623 // t1=tp/normp,z1=tz/normp
00624 fasp_darray_set(m,t1,0.0);
00625 fasp_darray_cp(m,t1,z1);
00626 fasp_blas_darray_axpy(m,1/normp,tp,t1);
00627 fasp_blas_darray_axpy(m,1/normp,tz,z1);
00628
00629 normu2 = fasp_blas_darray_norm2(m,u->val);
00630
00631 // compute residuals
00632 switch ( StopType ) {
00633     case STOP_REL_RES:
00634         temp2 = fasp_blas_darray_dotprod(m,r,r);
00635         absres = sqrt(temp2);
00636         relres = absres/normr0;
00637         break;
00638     case STOP_REL_PRECRES:
00639         if (pc == NULL)
00640             fasp_darray_cp(m,r,t);
00641         else
00642             pc->fct(r,t,pc->data);
00643         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00644         absres = sqrt(temp2);
00645         relres = absres/normr0;
00646         break;
00647     case STOP_MOD_REL_RES:
00648         temp2 = fasp_blas_darray_dotprod(m,r,r);
00649         absres = sqrt(temp2);
00650         relres = absres/normu2;
00651         break;
00652 }
00653
00654 // compute reduction factor of residual ||r||
00655 factor = absres/absres0;
00656
00657 // output iteration information if needed
00658 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00659
00660 if ( factor > 0.9 ) { // Only check when converge slowly
00661
00662     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00663     infnormu = fasp_blas_darray_norminf(m, u->val);
00664     if (infnormu <= sol_inf_tol) {
00665         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00666         iter = ERROR_SOLVER_SOLSTAG;
00667         break;
00668     }
00669
00670     // Check II: if staggenated, try to restart
00671     normuu = fasp_blas_darray_norm2(m,p1);

```



```

00672     normuu = ABS(alpha)*(normuu/normu2);
00673
00674     if ( normuu < maxdiff ) {
00675
00676         if ( stag < MaxStag ) {
00677             if ( PrtLvl >= PRINT_MORE ) {
00678                 ITS_DIFFRES(normuu,relres);
00679                 ITS_RESTART;
00680             }
00681         }
00682
00683         fasp_darray_cp(m,b->val,r);
00684         fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00685
00686         // compute residuals
00687         switch (StopType) {
00688             case STOP_REL_RES:
00689                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00690                 absres = sqrt(temp2);
00691                 relres = absres/normr0;
00692                 break;
00693             case STOP_REL_PRECRES:
00694                 if (pc == NULL)
00695                     fasp_darray_cp(m,r,t);
00696                 else
00697                     pc->fct(r,t,pc->data);
00698                 temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00699                 absres = sqrt(temp2);
00700                 relres = absres/normr0;
00701                 break;
00702             case STOP_MOD_REL_RES:
00703                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00704                 absres = sqrt(temp2);
00705                 relres = absres/normu2;
00706                 break;
00707         }
00708
00709         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00710
00711         if ( relres < tol )
00712             break;
00713         else {
00714             if ( stag >= MaxStag ) {
00715                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00716                 iter = ERROR_SOLVER_STAG;
00717                 break;
00718             }
00719             fasp_darray_set(m,p0,0.0);
00720             ++stag;
00721             ++restart_step;
00722
00723             // p1 = B(r)
00724             if ( pc != NULL )
00725                 pc->fct(r,p1,pc->data); /* Apply preconditioner */
00726             else
00727                 fasp_darray_cp(m,r,p1); /* No preconditioner */
00728
00729             // tp = A*p1
00730             fasp_blas_dblc_mxv(A,p1,tp);
00731
00732             // tz = B(tp)
00733             if ( pc != NULL )
00734                 pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
00735             else
00736                 fasp_darray_cp(m,tp,tz); /* No preconditioner */
00737
00738             // p1 = p1/normp
00739             normp = fasp_blas_darray_dotprod(m,tz,tp);
00740             normp = sqrt(normp);
00741             fasp_darray_cp(m,p1,t);
00742
00743             // t0 = A*p0=0
00744             fasp_darray_set(m,t0,0.0);
00745             fasp_darray_cp(m,t0,z0);
00746             fasp_darray_cp(m,t0,t1);
00747             fasp_darray_cp(m,t0,z1);
00748             fasp_darray_cp(m,t0,p1);
00749
00750             fasp_blas_darray_axpy(m,1/normp,t,p1);
00751
00752             // t1 = tp/normp, z1 = tz/normp

```

```

00753         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00754         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00755     }
00756 }
00757
00758 } // end of check I and II
00759
00760 // Check III: prevent false convergence
00761 if ( relres < tol ) {
00762
00763     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00764
00765     // compute residual r = b - Ax again
00766     fasp_darray_cp(m,b->val,r);
00767     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00768
00769     // compute residuals
00770     switch (StopType) {
00771     case STOP_REL_RES:
00772         temp2 = fasp_blas_darray_dotprod(m,r,r);
00773         absres = sqrt(temp2);
00774         relres = absres/normr0;
00775         break;
00776     case STOP_REL_PRECRES:
00777         if (pc == NULL)
00778             fasp_darray_cp(m,r,t);
00779         else
00780             pc->fct(r,t,pc->data);
00781         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00782         absres = sqrt(temp2);
00783         relres = absres/normr0;
00784         break;
00785     case STOP_MOD_REL_RES:
00786         temp2 = fasp_blas_darray_dotprod(m,r,r);
00787         absres = sqrt(temp2);
00788         relres = absres/normu2;
00789         break;
00790     }
00791
00792     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00793
00794     // check convergence
00795     if ( relres < tol ) break;
00796
00797     if ( more_step >= MaxRestartStep ) {
00798         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00799         iter = ERROR_SOLVER_TOLSMALL;
00800         break;
00801     }
00802
00803     // prepare for restarting method
00804     fasp_darray_set(m,p0,0.0);
00805     ++more_step;
00806     ++restart_step;
00807
00808     // p1 = B(r)
00809     if ( pc != NULL )
00810         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00811     else
00812         fasp_darray_cp(m,r,p1); /* No preconditioner */
00813
00814     // tp = A*p1
00815     fasp_blas_dblc_mxv(A,p1,tp);
00816
00817     // tz = B(tp)
00818     if ( pc != NULL )
00819         pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
00820     else
00821         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00822
00823     // p1 = p1/normp
00824     normp = fasp_blas_darray_dotprod(m,tz,tp);
00825     normp = sqrt(normp);
00826     fasp_darray_cp(m,p1,t);
00827
00828     // t0 = A*p0 = 0
00829     fasp_darray_set(m,t0,0.0);
00830     fasp_darray_cp(m,t0,z0);
00831     fasp_darray_cp(m,t0,t1);
00832     fasp_darray_cp(m,t0,z1);
00833     fasp_darray_cp(m,t0,p1);

```

```

00834
00835         fasp_blas_darray_axpy(m,1/normp,t,p1);
00836
00837         // t1=tp/normp,z1=tz/normp
00838         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00839         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00840
00841     } // end of convergence check
00842
00843     // update relative residual here
00844     absres0 = absres;
00845
00846 } // end of the main loop
00847
00848 FINISHED: // finish iterative method
00849 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00850
00851 // clean up temp memory
00852 fasp_mem_free(work); work = NULL;
00853
00854 #if DEBUG_MODE > 0
00855 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00856 #endif
00857
00858 if ( iter > MaxIt )
00859     return ERROR_SOLVER_MAXIT;
00860 else
00861     return iter;
00862 }
00863
00885 INT fasp_solver_dstr_pminres (dSTRmat      *A,
00886                             dvector      *b,
00887                             dvector      *u,
00888                             precondition *pc,
00889                             const REAL   tol,
00890                             const INT    MaxIt,
00891                             const SHORT  StopType,
00892                             const SHORT  PrtLvl)
00893 {
00894     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00895     const INT   m = b->row;
00896     const REAL  maxdiff = tol*STAG_RATIO; // stagantation tolerance
00897     const REAL  sol_inf_tol = SMALLREAL; // infinity norm tolerance
00898
00899     // local variables
00900     INT         iter = 0, stag = 1, more_step = 1, restart_step = 1;
00901     REAL        absres0 = BIGREAL, absres = BIGREAL;
00902     REAL        normr0 = BIGREAL, relres = BIGREAL;
00903     REAL        normu2, normuu, normp, infnormu, factor;
00904     REAL        alpha, alpha0, alphas, temp2;
00905
00906     // allocate temp memory (need 11*m REAL)
00907     REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
00908     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
00909     REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;
00910
00911     // Output some info for debugging
00912     if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (STR) ...\n");
00913
00914     #if DEBUG_MODE > 0
00915     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00916     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00917     #endif
00918
00919     // p0 = 0
00920     fasp_darray_set(m,p0,0.0);
00921
00922     // r = b-A*u
00923     fasp_darray_cp(m,b->val,r);
00924     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
00925
00926     // p1 = B(r)
00927     if ( pc != NULL )
00928         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00929     else
00930         fasp_darray_cp(m,r,p1); /* No preconditioner */
00931
00932     // compute initial residuals
00933     switch ( StopType ) {
00934     case STOP_REL_RES:
00935         absres0 = fasp_blas_darray_norm2(m,r);

```

```

00936         normr0 = MAX(SMALLREAL,absres0);
00937         relres = absres0/normr0;
00938         break;
00939     case STOP_REL_PRECRES:
00940         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
00941         normr0 = MAX(SMALLREAL,absres0);
00942         relres = absres0/normr0;
00943         break;
00944     case STOP_MOD_REL_RES:
00945         absres0 = fasp_blas_darray_norm2(m,r);
00946         normu2 = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00947         relres = absres0/normu2;
00948         break;
00949     default:
00950         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00951         goto FINISHED;
00952 }
00953
00954 // if initial residual is small, no need to iterate!
00955 if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00956
00957 // output iteration information if needed
00958 fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00959
00960 // tp = A*p1
00961 fasp_blas_dstr_m xv(A,p1,tp);
00962
00963 // tz = B(tp)
00964 if ( pc != NULL )
00965     pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00966 else
00967     fasp_darray_cp(m,tp,tz); /* No preconditioner */
00968
00969 // p1 = p1/normp
00970 normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00971 normp = sqrt(normp);
00972 fasp_darray_cp(m,p1,t);
00973 fasp_darray_set(m,p1,0.0);
00974 fasp_blas_darray_axpy(m,1/normp,t,p1);
00975
00976 // t0 = A*p0 = 0
00977 fasp_darray_set(m,t0,0.0);
00978 fasp_darray_cp(m,t0,z0);
00979 fasp_darray_cp(m,t0,t1);
00980 fasp_darray_cp(m,t0,z1);
00981
00982 // t1 = tp/normp, z1 = tz/normp
00983 fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
00984 fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
00985
00986 // main MinRes loop
00987 while ( iter++ < MaxIt ) {
00988
00989     // alpha = <r,z1>
00990     alpha=fasp_blas_darray_dotprod(m,r,z1);
00991
00992     // u = u+alpha*p1
00993     fasp_blas_darray_axpy(m,alpha,p1,u->val);
00994
00995     // r = r-alpha*A*p1
00996     fasp_blas_darray_axpy(m,-alpha,t1,r);
00997
00998     // compute t = A*z1 alpha1 = <z1,t>
00999     fasp_blas_dstr_m xv(A,z1,t);
01000     alpha1=fasp_blas_darray_dotprod(m,z1,t);
01001
01002     // compute t = A*z0 alpha0 = <z1,t>
01003     fasp_blas_dstr_m xv(A,z0,t);
01004     alpha0=fasp_blas_darray_dotprod(m,z1,t);
01005
01006     // p2 = z1-alpha1*p1-alpha0*p0
01007     fasp_darray_cp(m,z1,p2);
01008     fasp_blas_darray_axpy(m,-alpha1,p1,p2);
01009     fasp_blas_darray_axpy(m,-alpha0,p0,p2);
01010
01011     // tp = A*p2
01012     fasp_blas_dstr_m xv(A,p2,tp);
01013
01014     // tz = B(tp)
01015     if ( pc != NULL )
01016         pc->fct(tp,tz,pc->data); /* Apply preconditioner */

```

```

01017         else
01018             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01019
01020         // p2 = p2/normp
01021         normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
01022         normp = sqrt(normp);
01023         fasp_darray_cp(m,p2,t);
01024         fasp_darray_set(m,p2,0.0);
01025         fasp_blas_darray_axpy(m,1/normp,t,p2);
01026
01027         // prepare for next iteration
01028         fasp_darray_cp(m,p1,p0);
01029         fasp_darray_cp(m,p2,p1);
01030         fasp_darray_cp(m,t1,t0);
01031         fasp_darray_cp(m,z1,z0);
01032
01033         // t1=tp/normp, z1=tz/normp
01034         fasp_darray_set(m,t1,0.0);
01035         fasp_darray_cp(m,t1,z1);
01036         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01037         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01038
01039         normu2 = fasp_blas_darray_norm2(m,u->val);
01040
01041         // compute residuals
01042         switch ( StopType ) {
01043             case STOP_REL_RES:
01044                 temp2 = fasp_blas_darray_dotprod(m,r,r);
01045                 absres = sqrt(temp2);
01046                 relres = absres/normr0;
01047                 break;
01048             case STOP_REL_PRECRES:
01049                 if (pc == NULL)
01050                     fasp_darray_cp(m,r,t);
01051                 else
01052                     pc->fct(r,t,pc->data);
01053                 temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
01054                 absres = sqrt(temp2);
01055                 relres = absres/normr0;
01056                 break;
01057             case STOP_MOD_REL_RES:
01058                 temp2 = fasp_blas_darray_dotprod(m,r,r);
01059                 absres = sqrt(temp2);
01060                 relres = absres/normu2;
01061                 break;
01062         }
01063
01064         // compute reduction factor of residual ||r||
01065         factor = absres/absres0;
01066
01067         // output iteration information if needed
01068         fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01069
01070         if ( factor > 0.9 ) { // Only check when converge slowly
01071
01072             // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
01073             infnormu = fasp_blas_darray_norminf(m, u->val);
01074             if ( infnormu <= sol_inf_tol ) {
01075                 if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01076                 iter = ERROR_SOLVER_SOLSTAG;
01077                 break;
01078             }
01079
01080             // Check II: if staggenated, try to restart
01081             normuu = fasp_blas_darray_norm2(m,p1);
01082             normuu = ABS(alpha)*(normuu/normu2);
01083
01084             if ( normuu < maxdiff ) {
01085
01086                 if ( stag < MaxStag ) {
01087                     if ( PrtLvl >= PRINT_MORE ) {
01088                         ITS_DIFFRES(normuu,relres);
01089                         ITS_RESTART;
01090                     }
01091                 }
01092
01093                 fasp_darray_cp(m,b->val,r);
01094                 fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01095
01096                 // compute residuals
01097                 switch (StopType) {

```

```

01098         case STOP_REL_RES:
01099             temp2 = fasp_blas_darray_dotprod(m,r,r);
01100             absres = sqrt(temp2);
01101             relres = absres/normr0;
01102             break;
01103         case STOP_REL_PRECRES:
01104             if (pc == NULL)
01105                 fasp_darray_cp(m,r,t);
01106             else
01107                 pc->fct(r,t,pc->data);
01108             temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
01109             absres = sqrt(temp2);
01110             relres = absres/normr0;
01111             break;
01112         case STOP_MOD_REL_RES:
01113             temp2 = fasp_blas_darray_dotprod(m,r,r);
01114             absres = sqrt(temp2);
01115             relres = absres/normu2;
01116             break;
01117     }
01118
01119     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01120
01121     if ( relres < tol )
01122         break;
01123     else {
01124         if ( stag >= MaxStag ) {
01125             if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01126             iter = ERROR_SOLVER_STAG;
01127             break;
01128         }
01129         fasp_darray_set(m,p0,0.0);
01130         ++stag;
01131         ++restart_step;
01132
01133         // p1 = B(r)
01134         if ( pc != NULL )
01135             pc->fct(r,p1,pc->data); /* Apply preconditioner */
01136         else
01137             fasp_darray_cp(m,r,p1); /* No preconditioner */
01138
01139         // tp = A*p1
01140         fasp_blas_dstr_mxv(A,p1,tp);
01141
01142         // tz = B(tp)
01143         if ( pc != NULL )
01144             pc->fct(tp,tz,pc->data); /* Apply preconditioner */
01145         else
01146             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01147
01148         // p1 = p1/normp
01149         normp = fasp_blas_darray_dotprod(m,tz,tp);
01150         normp = sqrt(normp);
01151         fasp_darray_cp(m,p1,t);
01152
01153         // t0 = A*p0=0
01154         fasp_darray_set(m,t0,0.0);
01155         fasp_darray_cp(m,t0,z0);
01156         fasp_darray_cp(m,t0,t1);
01157         fasp_darray_cp(m,t0,z1);
01158         fasp_darray_cp(m,t0,p1);
01159
01160         fasp_blas_darray_axpy(m,1/normp,t,p1);
01161
01162         // t1 = tp/normp, z1 = tz/normp
01163         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01164         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01165     }
01166 }
01167 } // end of check I and II
01168
01169 // Check III: prevent false convergence
01170 if ( relres < tol ) {
01171
01172     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01173
01174     // compute residual r = b - Ax again
01175     fasp_darray_cp(m,b->val,r);
01176     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01177
01178     // compute residuals

```

```

01179         switch (StopType) {
01180             case STOP_REL_RES:
01181                 temp2 = fasp_blas_darray_dotprod(m, r, r);
01182                 absres = sqrt(temp2);
01183                 relres = absres/normr0;
01184                 break;
01185             case STOP_REL_PRECRES:
01186                 if (pc == NULL)
01187                     fasp_darray_cp(m, r, t);
01188                 else
01189                     pc->fct(r, t, pc->data);
01190                 temp2 = ABS(fasp_blas_darray_dotprod(m, r, t));
01191                 absres = sqrt(temp2);
01192                 relres = absres/normr0;
01193                 break;
01194             case STOP_MOD_REL_RES:
01195                 temp2 = fasp_blas_darray_dotprod(m, r, r);
01196                 absres = sqrt(temp2);
01197                 relres = absres/normu2;
01198                 break;
01199         }
01200
01201         if (PrtLvl >= PRINT_MORE) ITS_REALRES(relres);
01202
01203         // check convergence
01204         if (relres < tol) break;
01205
01206         if (more_step >= MaxRestartStep) {
01207             if (PrtLvl > PRINT_MIN) ITS_ZEROTOL;
01208             iter = ERROR_SOLVER_TOLSMALL;
01209             break;
01210         }
01211
01212         // prepare for restarting method
01213         fasp_darray_set(m, p0, 0.0);
01214         ++more_step;
01215         ++restart_step;
01216
01217         // p1 = B(r)
01218         if (pc != NULL)
01219             pc->fct(r, p1, pc->data); /* Apply preconditioner */
01220         else
01221             fasp_darray_cp(m, r, p1); /* No preconditioner */
01222
01223         // tp = A*p1
01224         fasp_blas_dstr_mmv(A, p1, tp);
01225
01226         // tz = B(tp)
01227         if (pc != NULL)
01228             pc->fct(tp, tz, pc->data); /* Apply rreconditioner */
01229         else
01230             fasp_darray_cp(m, tp, tz); /* No preconditioner */
01231
01232         // p1 = p1/normp
01233         normp = fasp_blas_darray_dotprod(m, tz, tp);
01234         normp = sqrt(normp);
01235         fasp_darray_cp(m, p1, t);
01236
01237         // t0 = A*p0 = 0
01238         fasp_darray_set(m, t0, 0.0);
01239         fasp_darray_cp(m, t0, z0);
01240         fasp_darray_cp(m, t0, t1);
01241         fasp_darray_cp(m, t0, z1);
01242         fasp_darray_cp(m, t0, p1);
01243
01244         fasp_blas_darray_axpy(m, 1/normp, t, p1);
01245
01246         // t1=tp/normp, z1=tz/normp
01247         fasp_blas_darray_axpy(m, 1/normp, tp, t1);
01248         fasp_blas_darray_axpy(m, 1/normp, tz, z1);
01249
01250     } // end of convergence check
01251
01252     // update relative residual here
01253     absres0 = absres;
01254
01255 } // end of the main loop
01256
01257 FINISHED: // finish iterative method
01258     if (PrtLvl > PRINT_NONE) ITS_FINAL(iter, MaxIt, relres);
01259

```

```

01260 // clean up temp memory
01261 fasp_mem_free(work); work = NULL;
01262
01263 #if DEBUG_MODE > 0
01264 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01265 #endif
01266
01267 if ( iter > MaxIt )
01268     return ERROR_SOLVER_MAXIT;
01269 else
01270     return iter;
01271 }
01272
01296 INT fasp_solver_pminres (mxv_matfree *mf,
01297                          dvector *b,
01298                          dvector *u,
01299                          precondition *pc,
01300                          const REAL tol,
01301                          const INT MaxIt,
01302                          const SHORT StopType,
01303                          const SHORT PrtLvl)
01304 {
01305     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
01306     const INT m=b->row;
01307     const REAL maxdiff = tol*STAG_RATIO; // staganation tolerance
01308     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
01309
01310     // local variables
01311     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
01312     REAL absres0 = BIGREAL, absres = BIGREAL;
01313     REAL normr0 = BIGREAL, relres = BIGREAL;
01314     REAL normu2, normuu, normp, infnormu, factor;
01315     REAL alpha, alpha0, alpha1, temp2;
01316
01317     // allocate temp memory (need 11*m REAL)
01318     REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
01319     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
01320     REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;
01321
01322     // Output some info for debugging
01323     if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (MatFree) ...\n");
01324
01325 #if DEBUG_MODE > 0
01326 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01327 printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01328 #endif
01329
01330     // initialization counters
01331     stag=1; more_step=1; restart_step=1;
01332
01333     // p0=0
01334     fasp_darray_set (m,p0,0.0);
01335
01336     // r = b-A*u
01337     mf->fct(mf->data, u->val, r);
01338     fasp_blas_darray_axpy(m, 1.0, b->val, -1.0, r);
01339
01340     // p1 = B(r)
01341     if (pc != NULL)
01342         pc->fct(r,p1,pc->data); /* Apply preconditioner */
01343     else
01344         fasp_darray_cp(m,r,p1); /* No preconditioner */
01345
01346     // compute initial relative residual
01347     switch (StopType) {
01348     case STOP_REL_PRECRES:
01349         absres0=sqrt (ABS (fasp_blas_darray_dotprod(m,r,p1)));
01350         normr0=MAX (SMALLREAL,absres0);
01351         relres=absres0/normr0;
01352         break;
01353     case STOP_MOD_REL_RES:
01354         absres0=fasp_blas_darray_norm2 (m,r);
01355         normu2=MAX (SMALLREAL,fasp_blas_darray_norm2 (m,u->val));
01356         relres=absres0/normu2;
01357         break;
01358     default: // STOP_REL_RES
01359         absres0=fasp_blas_darray_norm2 (m,r);
01360         normr0=MAX (SMALLREAL,absres0);
01361         relres=absres0/normr0;
01362         break;
01363     }

```



```

01364
01365 // if initial residual is small, no need to iterate!
01366 if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
01367
01368 // tp=A*p1
01369 mf->fct(mf->data, p1, tp);
01370
01371 // tz = B(tp)
01372 if (pc != NULL)
01373     pc->fct(tp,tz,pc->data); /* Apply preconditioner */
01374 else
01375     fasp_darray_cp(m,tp,tz); /* No preconditioner */
01376
01377 // p1=p1/normp
01378 normp=ABS(fasp_blas_darray_dotprod(m,tz,tp));
01379 normp=sqrt(normp);
01380 fasp_darray_cp(m,p1,t);
01381 fasp_darray_set(m,p1,0.0);
01382 fasp_blas_darray_axpy(m,1/normp,t,p1);
01383
01384 // t0=A*p0=0
01385 fasp_darray_set(m,t0,0.0);
01386 fasp_darray_cp(m,t0,z0);
01387 fasp_darray_cp(m,t0,t1);
01388 fasp_darray_cp(m,t0,z1);
01389
01390 // t1=tp/normp,z1=tz/normp
01391 fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
01392 fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
01393
01394 while( iter++ < MaxIt) {
01395
01396     // alpha=<r,z1>
01397     alpha=fasp_blas_darray_dotprod(m,r,z1);
01398
01399     // u=u+alpha*p1
01400     fasp_blas_darray_axpy(m,alpha,p1,u->val);
01401
01402     // r=r-alpha*A*p1
01403     fasp_blas_darray_axpy(m,-alpha,t1,r);
01404
01405     // compute t=A*z1 alpha1=<z1,t>
01406     mf->fct(mf->data, z1, t);
01407     alpha1=fasp_blas_darray_dotprod(m,z1,t);
01408
01409     // compute t=A*z0 alpha0=<z1,t>
01410     mf->fct(mf->data, z0, t);
01411     alpha0=fasp_blas_darray_dotprod(m,z1,t);
01412
01413     // p2=z1-alpha1*p1-alpha0*p0
01414     fasp_darray_cp(m,z1,p2);
01415     fasp_blas_darray_axpy(m,-alpha1,p1,p2);
01416     fasp_blas_darray_axpy(m,-alpha0,p0,p2);
01417
01418     // tp=A*p2
01419     mf->fct(mf->data, p2, tp);
01420
01421     // tz = B(tp)
01422     if (pc != NULL)
01423         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
01424     else
01425         fasp_darray_cp(m,tp,tz); /* No preconditioner */
01426
01427     // p2=p2/normp
01428     normp=ABS(fasp_blas_darray_dotprod(m,tz,tp));
01429     normp=sqrt(normp);
01430     fasp_darray_cp(m,p2,t);
01431     fasp_darray_set(m,p2,0.0);
01432     fasp_blas_darray_axpy(m,1/normp,t,p2);
01433
01434     // prepare for next iteration
01435     fasp_darray_cp(m,p1,p0);
01436     fasp_darray_cp(m,p2,p1);
01437     fasp_darray_cp(m,t1,t0);
01438     fasp_darray_cp(m,z1,z0);
01439
01440     // t1=tp/normp,z1=tz/normp
01441     fasp_darray_set(m,t1,0.0);
01442     fasp_darray_cp(m,t1,z1);
01443     fasp_blas_darray_axpy(m,1/normp,tp,t1);
01444     fasp_blas_darray_axpy(m,1/normp,tz,z1);

```

```

01445
01446 // relative residual = ||r||/||r0||
01447 temp2=fasp_blas_darray_dotprod(m,r,r);
01448 absres=sqrt(temp2);
01449
01450 normu2=fasp_blas_darray_norm2(m,u->val);
01451
01452 switch (StopType) {
01453     case STOP_REL_PRECRES:
01454         if (pc == NULL)
01455             fasp_darray_cp(m,r,t);
01456         else
01457             pc->fct(r,t,pc->data);
01458         temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
01459         relres=sqrt(temp2)/normr0;
01460         break;
01461     case STOP_MOD_REL_RES:
01462         relres=sqrt(temp2)/normu2;
01463         break;
01464     default: // STOP_REL_RES
01465         relres=sqrt(temp2)/normr0;
01466         break;
01467 }
01468
01469 // compute reduction factor of residual ||r||
01470 factor=absres/absres0;
01471
01472 // output iteration information if needed
01473 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01474
01475 // solution check, if solution is too small, return ERROR_SOLVER_SOLSTAG.
01476 infnormu = fasp_blas_darray_norminf(m, u->val);
01477 if ( infnormu <= sol_inf_tol ) {
01478     if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01479     iter = ERROR_SOLVER_SOLSTAG;
01480     break;
01481 }
01482
01483 normuu=fasp_blas_darray_norm2(m,p1);
01484 normuu=ABS(alpha)*(normuu/normu2);
01485
01486 // check convergence
01487 if (normuu<maxdiff) {
01488     if ( stag < MaxStag ) {
01489         if ( PrtLvl >= PRINT_MORE ) {
01490             ITS_DIFFRES(normuu,relres);
01491             ITS_RESTART;
01492         }
01493     }
01494 }
01495 mf->fct(mf->data, u->val, r);
01496 fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
01497
01498 temp2=fasp_blas_darray_dotprod(m,r,r);
01499 absres=sqrt(temp2);
01500 switch (StopType) {
01501     case STOP_REL_RES:
01502         relres=sqrt(temp2)/normr0;
01503         break;
01504     case STOP_REL_PRECRES:
01505         if (pc == NULL)
01506             fasp_darray_cp(m,r,t);
01507         else
01508             pc->fct(r,t,pc->data);
01509         temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
01510         relres=sqrt(temp2)/normr0;
01511         break;
01512     case STOP_MOD_REL_RES:
01513         relres=sqrt(temp2)/normu2;
01514         break;
01515 }
01516
01517 if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01518
01519 if ( relres < tol )
01520     break;
01521 else {
01522     if ( stag >= MaxStag ) {
01523         if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01524         iter = ERROR_SOLVER_STAG;
01525         break;

```

```

01526         }
01527         ++stag;
01528         ++restart_step;
01529
01530         fasp_darray_set(m,p0,0.0);
01531
01532         // p1 = B(r)
01533         if (pc != NULL)
01534             pc->fct(r,p1,pc->data); /* Apply preconditioner */
01535         else
01536             fasp_darray_cp(m,r,p1); /* No preconditioner */
01537
01538         // tp=A*p1
01539         mf->fct(mf->data, p1, tp);
01540
01541         // tz = B(tp)
01542         if (pc == NULL)
01543             pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
01544         else
01545             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01546
01547         // p1=p1/normp
01548         normp=fasp_blas_darray_dotprod(m,tz,tp);
01549         normp=sqrt(normp);
01550         fasp_darray_cp(m,p1,t);
01551
01552         // t0=A*p0=0
01553         fasp_darray_set(m,t0,0.0);
01554         fasp_darray_cp(m,t0,z0);
01555         fasp_darray_cp(m,t0,t1);
01556         fasp_darray_cp(m,t0,z1);
01557         fasp_darray_cp(m,t0,p1);
01558
01559         fasp_blas_darray_axpy(m,1/normp,t,p1);
01560
01561         // t1=tp/normp,z1=tz/normp
01562         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01563         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01564     }
01565 }
01566
01567 // safe guard
01568 if ( relres < tol ) {
01569     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01570
01571     mf->fct(mf->data, u->val, r);
01572     fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
01573
01574     temp2=fasp_blas_darray_dotprod(m,r,r);
01575     absres=sqrt(temp2);
01576     switch (StopType) {
01577         case STOP_REL_RES:
01578             relres=sqrt(temp2)/normr0;
01579             break;
01580         case STOP_REL_PRECRES:
01581             if (pc == NULL)
01582                 fasp_darray_cp(m,r,t);
01583             else
01584                 pc->fct(r,t,pc->data);
01585             temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
01586             relres=sqrt(temp2)/normr0;
01587             break;
01588         case STOP_MOD_REL_RES:
01589             relres=sqrt(temp2)/normu2;
01590             break;
01591     }
01592
01593     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01594
01595     // check convergence
01596     if ( relres < tol ) break;
01597
01598     if ( more_step >= MaxRestartStep ) {
01599         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01600         iter = ERROR_SOLVER_TOLSMALL;
01601         break;
01602     }
01603
01604     if ( more_step < MaxRestartStep ) {
01605         if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
01606     }

```

```

01607
01608         ++more_step;
01609         ++restart_step;
01610
01611         fasp_darray_set(m,p0,0.0);
01612
01613         // p1 = B(r)
01614         if (pc != NULL)
01615             pc->fct(r,p1,pc->data); /* Apply preconditioner */
01616         else
01617             fasp_darray_cp(m,r,p1); /* No preconditioner */
01618
01619         // tp = A*p1
01620         mf->fct(mf->data, p1, tp);
01621
01622         // tz = B(tp)
01623         if (pc == NULL)
01624             pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
01625         else
01626             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01627
01628         // p1 = p1/normp
01629         normp=fasp_blas_darray_dotprod(m,tz,tp);
01630         normp=sqrt(normp);
01631         fasp_darray_cp(m,p1,t);
01632
01633         // t0=A*p0=0
01634         fasp_darray_set(m,t0,0.0);
01635         fasp_darray_cp(m,t0,z0);
01636         fasp_darray_cp(m,t0,t1);
01637         fasp_darray_cp(m,t0,z1);
01638         fasp_darray_cp(m,t0,p1);
01639
01640         fasp_blas_darray_axpy(m,1/normp,t,p1);
01641
01642         // t1=tp/normp,z1=tz/normp
01643         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01644         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01645
01646     }
01647
01648     // update relative residual here
01649     absres0 = absres;
01650 }
01651
01652 FINISHED: // finish iterative method
01653 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01654
01655 // clean up temp memory
01656 fasp_mem_free(work); work = NULL;
01657
01658 #if DEBUG_MODE > 0
01659 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01660 #endif
01661
01662 if (iter>MaxIt)
01663     return ERROR_SOLVER_MAXIT;
01664 else
01665     return iter;
01666 }
01667
01668 /*-----*/
01669 /*--          End of File          --*/
01670 /*-----*/

```

9.121 KryPvfgmres.c File Reference

Krylov subspace methods – Preconditioned variable-restarting FGMRes.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dcsr_pvfgmres](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.
- [INT fasp_solver_dbsr_pvfgmres](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.
- [INT fasp_solver_dblc_pvfgmres](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PFGMRES (right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.
- [INT fasp_solver_pvfgmres](#) ([mxv_matfree](#) *mf, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.

9.121.1 Detailed Description

Krylov subspace methods – Preconditioned variable-restarting FGMRes.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), and [BlaSpmvCSR.c](#)

This file is modified from [KryPvfgmres.c](#)

Reference: A.H. Baker, E.R. Jessup, and Tz.V. Kolev A Simple Strategy for Varying the Restart Parameter in GMRES(m) Journal of Computational and Applied Mathematics, 230 (2009) pp. 751-761. UCRL-JRNL-235266.
Copyright (C) 2012–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPvfgmres.c](#).

9.121.2 Function Documentation

9.121.2.1 fasp_solver_dblc_pvfgmres()

```
INT fasp_solver_dblc_pvfgmres (
    dBLCmat * A,
    dvector * b,
    dvector * x,
    precond * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Solve "Ax=b" using PFGMRES (right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.

Parameters

<i>A</i>	Pointer to coefficient matrix
<i>b</i>	Pointer to right hand side vector
<i>x</i>	Pointer to solution vector
<i>MaxIt</i>	Maximal iteration number allowed
<i>tol</i>	Tolerance
<i>pc</i>	Pointer to preconditioner data
<i>PrtLvl</i>	How much information to print out
<i>StopType</i>	Stopping criterion, i.e. $\ r_k\ /\ r_0\ < tol$
<i>restart</i>	Number of restart for GMRES

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/04/2012

Note

Based on Zhiyang Zhou's pvgmres.c

Modified by Chunsheng Feng on 07/22/2013: Add adaptive memory allocate Modified by Chensong Zhang on 05/09/2015: Clean up for stopping types
Definition at line 714 of file [KryPvfgmres.c](#).

9.121.2.2 fasp_solver_dbsr_pvfgmres()

```

INT fasp_solver_dbsr_pvfgmres (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition

Parameters

<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type – DO not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

02/05/2012

Modified by Chunsheng Feng on 07/22/2013: Add adaptive memory allocate Modified by Chensong Zhang on 05/09/2015: Clean up for stopping types
Definition at line 389 of file [KryPvfgmres.c](#).

9.121.2.3 fasp_solver_dcsr_pvfgmres()

```
INT fasp_solver_dcsr_pvfgmres (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Solve " $Ax=b$ " using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type – DO not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/04/2012

Modified by Chunsheng Feng on 07/22/2013: Add adaptive memory allocate Modified by Chensong Zhang on 05/09/2015: Clean up for stopping types

Definition at line 67 of file [KryPvfgmres.c](#).

9.121.2.4 fasp_solver_pvfgmres()

```

INT fasp_solver_pvfgmres (
    mxv_matfree * mf,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PFGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration and flexible preconditioner can be used.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type – DO not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/04/2012

Modified by Feiteng Huang on 09/26/2012: matrix free Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate

Definition at line 1036 of file KryPvfgmres.c.

9.122 KryPvfgmres.c

[Go to the documentation of this file.](#)

```

00001
00025 #include <math.h>
00026
00027 #include "fasp.h"
00028 #include "fasp_functs.h"
00029
00030 /*-----*/
00031 /*--  Declare Private Functions  --*/
00032 /*-----*/
00033
00034 #include "KryUtil.inl"
00035
00036 /*-----*/
00037 /*--      Public Functions      --*/
00038 /*-----*/
00039
00067 INT fasp_solver_dcsr_pvfgmres (dCSRmat      *A,
00068                               dvector      *b,
00069                               dvector      *x,
00070                               precondition *pc,
00071                               const REAL   tol,
00072                               const INT    MaxIt,
00073                               const SHORT  restart,
00074                               const SHORT  StopType,
00075                               const SHORT  PrtLvl)
00076 {
00077     const INT n          = b->row;
00078     const INT min_iter   = 0;
00079
00080     //-----//
00081     // Newly added parameters to monitor when //
00082     // to change the restart parameter //
00083     //-----//
00084     const REAL cr_max      = 0.99;    // = cos(8^o) (experimental)
00085     const REAL cr_min      = 0.174;    // = cos(80^o) (experimental)
00086
00087     // local variables
00088     INT iter               = 0;
00089     int i, j, k; // must be signed! -zcs
00090
00091     REAL epsmac            = SMALLREAL;
00092     REAL r_norm, b_norm, den_norm;
00093     REAL epsilon, gamma, t;
00094     REAL relres, normu, r_normb;
00095
00096     REAL *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
00097     REAL **p = NULL, **hh = NULL, **z=NULL;
00098
00099     REAL cr          = 1.0;    // convergence rate
00100     REAL r_norm_old  = 0.0;    // save residual norm of previous restart cycle
00101     INT d            = 3;    // reduction for restart parameter
00102     INT restart_max   = restart; // upper bound for restart in each restart cycle
00103     INT restart_min   = 3;    // lower bound for restart in each restart cycle
00104
00105     INT Restart      = restart; // real restart in some fixed restarted cycle
00106     INT Restart1     = Restart + 1;
00107     LONG worksize    = (Restart+4)*(Restart+n)+1-n+Restart*n;
00108
00109     // Output some info for debugging
00110     if (PrtLvl > PRINT_NONE) printf("\nCalling VFGMRes solver (CSR) ...\n");
00111
00112     #if DEBUG_MODE > 0
00113         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00114         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00115     #endif
00116

```

```

00117      /* allocate memory and setup temp work space */
00118      REAL *work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00119
00120      /* check whether memory is enough for GMRES */
00121      while ( (work == NULL) && (Restart > 5) ) {
00122          Restart = Restart - 5;
00123          worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
00124          work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00125          Restart1 = Restart + 1;
00126      }
00127
00128      if ( work == NULL ) {
00129          printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00130          fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00131      }
00132
00133      if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00134          printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
00135      }
00136
00137      p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00138      hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00139      z = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00140      norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00141
00142      r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
00143      for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00144      for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00145      for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;
00146
00147      /* initialization */
00148      fasp_darray_cp(n, b->val, p[0]);
00149      fasp_blas_dcsr_aAxy(-1.0, A, x->val, p[0]);
00150
00151      b_norm = fasp_blas_darray_norm2(n, b->val);
00152      r_norm = fasp_blas_darray_norm2(n, p[0]);
00153      norms[0] = r_norm;
00154
00155      if ( PrtLvl >= PRINT_SOME ) {
00156          ITS_PUTNORM("right-hand side", b_norm);
00157          ITS_PUTNORM("residual", r_norm);
00158      }
00159
00160      if ( b_norm > 0.0 ) den_norm = b_norm;
00161      else den_norm = r_norm;
00162
00163      epsilon = tol*den_norm;
00164
00165      // if initial residual is small, no need to iterate!
00166      if ( r_norm < epsilon || r_norm < 1e-12*tol ) goto FINISHED;
00167
00168      if ( b_norm > 0.0 ) {
00169          fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
00170      }
00171      else {
00172          fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
00173      }
00174
00175      /* outer iteration cycle */
00176      while ( iter < MaxIt ) {
00177
00178          rs[0] = r_norm;
00179          r_norm_old = r_norm;
00180          if ( r_norm == 0.0 ) {
00181              fasp_mem_free(work); work = NULL;
00182              fasp_mem_free(p); p = NULL;
00183              fasp_mem_free(hh); hh = NULL;
00184              fasp_mem_free(norms); norms = NULL;
00185              fasp_mem_free(z); z = NULL;
00186              return iter;
00187          }
00188
00189          //-----//
00190          // adjust the restart parameter //
00191          //-----//
00192
00193          if ( cr > cr_max || iter == 0 ) {
00194              Restart = restart_max;
00195          }
00196          else if ( cr < cr_min ) {
00197              // Restart = Restart;

```

```

00198     }
00199     else {
00200         if ( Restart - d > restart_min ) Restart -= d;
00201         else Restart = restart_max;
00202     }
00203
00204     // Enter the cycle at the first iteration for at least one iteration
00205     t = 1.0 / r_norm;
00206     fasp_blas_darray_ax(n, t, p[0]);
00207     i = 0;
00208
00209     // RESTART CYCLE (right-preconditioning)
00210     while ( i < Restart && iter < MaxIt ) {
00211
00212         i ++;  iter ++;
00213
00214         /* apply preconditioner */
00215         if ( pc == NULL )
00216             fasp_darray_cp(n, p[i-1], z[i-1]);
00217         else
00218             pc->fct(p[i-1], z[i-1], pc->data);
00219
00220         fasp_blas_dcsr_mxv(A, z[i-1], p[i]);
00221
00222         /* modified Gram_Schmidt */
00223         for ( j = 0; j < i; j++ ) {
00224             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00225             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00226         }
00227         t = fasp_blas_darray_norm2(n, p[i]);
00228         hh[i][i-1] = t;
00229         if ( t != 0.0 ) {
00230             t = 1.0 / t;
00231             fasp_blas_darray_ax(n, t, p[i]);
00232         }
00233
00234         for ( j = 1; j < i; ++j ) {
00235             t = hh[j-1][i-1];
00236             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00237             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00238         }
00239         t = hh[i][i-1] * hh[i][i-1];
00240         t += hh[i-1][i-1] * hh[i-1][i-1];
00241         gamma = sqrt(t);
00242         if (gamma == 0.0) gamma = epsmac;
00243         c[i-1] = hh[i-1][i-1] / gamma;
00244         s[i-1] = hh[i][i-1] / gamma;
00245         rs[i] = -s[i-1] * rs[i-1];
00246         rs[i-1] = c[i-1] * rs[i-1];
00247         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00248
00249         r_norm = fabs(rs[i]);
00250         norms[iter] = r_norm;
00251
00252         if ( b_norm > 0 ) {
00253             fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
00254                 norms[iter], norms[iter]/norms[iter-1]);
00255         }
00256         else {
00257             fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
00258                 norms[iter]/norms[iter-1]);
00259         }
00260
00261         /* Check: Exit the restart cycle? */
00262         if (r_norm <= epsilon && iter >= min_iter) break;
00263
00264     } /* end of restart cycle */
00265
00266     /* now compute solution, first solve upper triangular system */
00267
00268     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00269     for ( k = i-2; k >= 0; k-- ) {
00270         t = 0.0;
00271         for ( j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
00272
00273         t += rs[k];
00274         rs[k] = t / hh[k][k];
00275     }
00276
00277     fasp_darray_cp(n, z[i-1], r);
00278     fasp_blas_darray_ax(n, rs[i-1], r);

```

```

00279
00280     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);
00281
00282     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00283
00284     if ( r_norm <= epsilon && iter >= min_iter ) {
00285         fasp_darray_cp(n, b->val, r);
00286         fasp_blas_dcsr_aAxy(-1.0, A, x->val, r);
00287         r_norm = fasp_blas_darray_norm2(n, r);
00288
00289         switch (StopType) {
00290             case STOP_REL_RES:
00291                 relres = r_norm/den_norm;
00292                 break;
00293             case STOP_REL_PRECRES:
00294                 if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
00295                 else pc->fct(r, p[0], pc->data);
00296                 r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00297                 relres = r_normb/den_norm;
00298                 break;
00299             case STOP_MOD_REL_RES:
00300                 normu = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
00301                 relres = r_norm/normu;
00302                 break;
00303             default:
00304                 printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00305                 goto FINISHED;
00306         }
00307
00308         if ( relres <= tol ) {
00309             break;
00310         }
00311         else {
00312             if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
00313             fasp_darray_cp(n, r, p[0]); i = 0;
00314         }
00315     } /* end of convergence check */
00316
00317     /* compute residual vector and continue loop */
00318     for ( j = i; j > 0; j-- ) {
00319         rs[j-1] = -s[j-1]*rs[j];
00320         rs[j] = c[j-1]*rs[j];
00321     }
00322
00323     if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00324
00325     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00326
00327     if (i) {
00328         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00329         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00330     }
00331
00332     //-----//
00333     // compute the convergence rate //
00334     //-----//
00335     cr = r_norm / r_norm_old;
00336
00337 } /* end of iteration while loop */
00338
00339 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);
00340
00341 FINISHED:
00342 //-----
00343 * Free some stuff
00344 *-----*/
00345 fasp_mem_free(work); work = NULL;
00346 fasp_mem_free(p); p = NULL;
00347 fasp_mem_free(hh); hh = NULL;
00348 fasp_mem_free(norms); norms = NULL;
00349 fasp_mem_free(z); z = NULL;
00350
00351 #if DEBUG_MODE > 0
00352     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00353 #endif
00354
00355 if ( iter >= MaxIt )
00356     return ERROR_SOLVER_MAXIT;
00357 else
00358     return iter;
00359

```

```

00360 }
00361
00389 INT fasp_solver_dbsr_pvfgmres (dBSRmat      *A,
00390                                dvector      *b,
00391                                dvector      *x,
00392                                precondition *pc,
00393                                const REAL    tol,
00394                                const INT     MaxIt,
00395                                const SHORT   restart,
00396                                const SHORT   StopType,
00397                                const SHORT   PrtLvl)
00398 {
00399     const INT n          = b->row;
00400     const INT min_iter   = 0;
00401
00402     //-----//
00403     // Newly added parameters to monitor when //
00404     // to change the restart parameter //
00405     //-----//
00406     const REAL cr_max    = 0.99;    // = cos(8^o) (experimental)
00407     const REAL cr_min    = 0.174;   // = cos(80^o) (experimental)
00408
00409     // local variables
00410     INT iter              = 0;
00411     int i, j, k; // must be signed! -zcs
00412
00413     REAL epsmac           = SMALLREAL;
00414     REAL r_norm, b_norm, den_norm;
00415     REAL epsilon, gamma, t;
00416     REAL relres, normu, r_normb;
00417
00418     REAL *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
00419     REAL **p = NULL, **hh = NULL, **z=NULL;
00420
00421     REAL cr          = 1.0;    // convergence rate
00422     REAL r_norm_old  = 0.0;    // save residual norm of previous restart cycle
00423     INT d            = 3;      // reduction for restart parameter
00424     INT restart_max   = restart; // upper bound for restart in each restart cycle
00425     INT restart_min   = 3;      // lower bound for restart in each restart cycle
00426
00427     INT Restart       = restart; // real restart in some fixed restarted cycle
00428     INT Restart1      = Restart + 1;
00429     LONG worksize     = (Restart+4)*(Restart+n)+1-n+Restart*n;
00430
00431     // Output some info for debugging
00432     if (PrtLvl > PRINT_NONE) printf("\nCalling VFGMRes solver (BSR) ...\n");
00433
00434     #if DEBUG_MODE > 0
00435         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00436         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00437     #endif
00438
00439     /* allocate memory and setup temp work space */
00440     REAL *work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
00441
00442     /* check whether memory is enough for GMRES */
00443     while ( (work == NULL) && (Restart > 5) ) {
00444         Restart = Restart - 5;
00445         worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
00446         work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
00447         Restart1 = Restart + 1;
00448     }
00449
00450     if (work == NULL) {
00451         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00452         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00453     }
00454
00455     if (PrtLvl > PRINT_MIN && Restart < restart) {
00456         printf("### WARNING: vFGMRRES restart number set to %d!\n", Restart);
00457     }
00458
00459     p = (REAL **)fasp_mem_malloc(Restart1, sizeof(REAL *));
00460     hh = (REAL **)fasp_mem_malloc(Restart1, sizeof(REAL *));
00461     z = (REAL **)fasp_mem_malloc(Restart1, sizeof(REAL *));
00462     norms = (REAL *)fasp_mem_malloc(MaxIt+1, sizeof(REAL));
00463
00464     r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
00465     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00466     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00467     for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;

```

```

00468
00469 /* initialization */
00470 fasp_darray_cp(n, b->val, p[0]);
00471 fasp_blas_dbsr_aAxy(-1.0, A, x->val, p[0]);
00472
00473 b_norm = fasp_blas_darray_norm2(n, b->val);
00474 r_norm = fasp_blas_darray_norm2(n, p[0]);
00475 norms[0] = r_norm;
00476
00477 if ( PrtLvl >= PRINT_SOME ) {
00478     ITS_PUTNORM("right-hand side", b_norm);
00479     ITS_PUTNORM("residual", r_norm);
00480 }
00481
00482 if ( b_norm > 0.0 ) den_norm = b_norm;
00483 else                den_norm = r_norm;
00484
00485 epsilon = tol*den_norm;
00486
00487 // if initial residual is small, no need to iterate!
00488 if ( r_norm < epsilon || r_norm < 1e-12*tol ) goto FINISHED;
00489
00490 if ( b_norm > 0.0 ) {
00491     fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
00492 }
00493 else {
00494     fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
00495 }
00496
00497 /* outer iteration cycle */
00498 while ( iter < MaxIt ) {
00499
00500     rs[0] = r_norm;
00501     r_norm_old = r_norm;
00502     if ( r_norm == 0.0 ) {
00503         fasp_mem_free(work); work = NULL;
00504         fasp_mem_free(p); p = NULL;
00505         fasp_mem_free(hh); hh = NULL;
00506         fasp_mem_free(norms); norms = NULL;
00507         fasp_mem_free(z); z = NULL;
00508         return iter;
00509     }
00510
00511     //-----//
00512     // adjust the restart parameter //
00513     //-----//
00514
00515     if ( cr > cr_max || iter == 0 ) {
00516         Restart = restart_max;
00517     }
00518     else if ( cr < cr_min ) {
00519         // Restart = Restart;
00520     }
00521     else {
00522         if ( Restart - d > restart_min ) Restart -= d;
00523         else Restart = restart_max;
00524     }
00525
00526     // Enter the cycle at the first iteration for at least one iteration
00527     t = 1.0 / r_norm;
00528     fasp_blas_darray_ax(n, t, p[0]);
00529     i = 0;
00530
00531     // RESTART CYCLE (right-preconditioning)
00532     while ( i < Restart && iter < MaxIt ) {
00533
00534         i ++; iter ++;
00535
00536         /* apply preconditioner */
00537         if ( pc == NULL )
00538             fasp_darray_cp(n, p[i-1], z[i-1]);
00539         else
00540             pc->fct(p[i-1], z[i-1], pc->data);
00541
00542         fasp_blas_dbsr_mxv(A, z[i-1], p[i]);
00543
00544         /* modified Gram_Schmidt */
00545         for ( j = 0; j < i; j++ ) {
00546             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00547             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00548         }

```

```

00549     t = fasp_blas_darray_norm2(n, p[i]);
00550     hh[i][i-1] = t;
00551     if ( t != 0.0 ) {
00552         t = 1.0 / t;
00553         fasp_blas_darray_ax(n, t, p[i]);
00554     }
00555
00556     for ( j = 1; j < i; ++j ) {
00557         t = hh[j-1][i-1];
00558         hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00559         hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00560     }
00561     t = hh[i][i-1] * hh[i][i-1];
00562     t += hh[i-1][i-1] * hh[i-1][i-1];
00563     gamma = sqrt(t);
00564     if (gamma == 0.0) gamma = epsmac;
00565     c[i-1] = hh[i-1][i-1] / gamma;
00566     s[i-1] = hh[i][i-1] / gamma;
00567     rs[i] = -s[i-1] * rs[i-1];
00568     rs[i-1] = c[i-1] * rs[i-1];
00569     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00570
00571     r_norm = fabs(rs[i]);
00572     norms[iter] = r_norm;
00573
00574     if ( b_norm > 0 ) {
00575         fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
00576                     norms[iter], norms[iter]/norms[iter-1]);
00577     }
00578     else {
00579         fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
00580                     norms[iter]/norms[iter-1]);
00581     }
00582
00583     /* Check: Exit the restart cycle? */
00584     if (r_norm <= epsilon && iter >= min_iter) break;
00585
00586 } /* end of restart cycle */
00587
00588 /* now compute solution, first solve upper triangular system */
00589
00590 rs[i-1] = rs[i-1] / hh[i-1][i-1];
00591 for ( k = i-2; k >= 0; k-- ) {
00592     t = 0.0;
00593     for ( j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
00594
00595     t += rs[k];
00596     rs[k] = t / hh[k][k];
00597 }
00598
00599 fasp_darray_cp(n, z[i-1], r);
00600 fasp_blas_darray_ax(n, rs[i-1], r);
00601
00602 for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);
00603
00604 fasp_blas_darray_axpy(n, 1.0, r, x->val);
00605
00606 if ( r_norm <= epsilon && iter >= min_iter ) {
00607     fasp_darray_cp(n, b->val, r);
00608     fasp_blas_dbsr_aApy(-1.0, A, x->val, r);
00609     r_norm = fasp_blas_darray_norm2(n, r);
00610
00611     switch (StopType) {
00612         case STOP_REL_RES:
00613             relres = r_norm/den_norm;
00614             break;
00615         case STOP_REL_PRECRES:
00616             if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
00617             else pc->fct(r, p[0], pc->data);
00618             r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00619             relres = r_normb/den_norm;
00620             break;
00621         case STOP_MOD_REL_RES:
00622             normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00623             relres = r_norm/normu;
00624             break;
00625         default:
00626             printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00627             goto FINISHED;
00628     }
00629

```

```

00630         if ( relres <= tol ) {
00631             break;
00632         }
00633         else {
00634             if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
00635             fasp_darray_cp(n, r, p[0]); i = 0;
00636         }
00637     } /* end of convergence check */
00638
00639     /* compute residual vector and continue loop */
00640     for ( j = i; j > 0; j-- ) {
00641         rs[j-1] = -s[j-1]*rs[j];
00642         rs[j] = c[j-1]*rs[j];
00643     }
00644
00645     if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00646
00647     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00648
00649     if (i) {
00650         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00651         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00652     }
00653
00654     //-----//
00655     // compute the convergence rate //
00656     //-----//
00657     cr = r_norm / r_norm_old;
00658
00659 } /* end of iteration while loop */
00660
00661 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);
00662
00663 FINISHED:
00664 /*-----*/
00665 * Free some stuff
00666 *-----*/
00667 fasp_mem_free(work); work = NULL;
00668 fasp_mem_free(p); p = NULL;
00669 fasp_mem_free(hh); hh = NULL;
00670 fasp_mem_free(norms); norms = NULL;
00671 fasp_mem_free(z); z = NULL;
00672
00673 #if DEBUG_MODE > 0
00674 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00675 #endif
00676
00677 if ( iter >= MaxIt )
00678     return ERROR_SOLVER_MAXIT;
00679 else
00680     return iter;
00681 }
00682
00683
00714 INT fasp_solver_dblc_pvfgrms (dBLMat *A,
00715                             dvector *b,
00716                             dvector *x,
00717                             precondition *pc,
00718                             const REAL tol,
00719                             const INT MaxIt,
00720                             const SHORT restart,
00721                             const SHORT StopType,
00722                             const SHORT PrtLvl)
00723 {
00724     const INT n = b->row;
00725     const INT min_iter = 0;
00726
00727     //-----//
00728     // Newly added parameters to monitor when //
00729     // to change the restart parameter //
00730     //-----//
00731     const REAL cr_max = 0.99; // = cos(8^o) (experimental)
00732     const REAL cr_min = 0.174; // = cos(80^o) (experimental)
00733
00734     // local variables
00735     INT iter = 0;
00736     int i, j, k; // must be signed! -zcs
00737
00738     REAL epsmac = SMALLREAL;
00739     REAL r_norm, b_norm, den_norm;
00740     REAL epsilon, gamma, t;

```



```

00741     REAL    relres, normu, r_normb;
00742
00743     REAL    *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
00744     REAL    **p = NULL, **hh = NULL, **z=NULL;
00745
00746     REAL    cr          = 1.0;      // convergence rate
00747     REAL    r_norm_old  = 0.0;      // save residual norm of previous restart cycle
00748     INT     d           = 3;        // reduction for restart parameter
00749     INT     restart_max = restart;   // upper bound for restart in each restart cycle
00750     INT     restart_min = 3;        // lower bound for restart in each restart cycle
00751
00752     INT     Restart     = restart;   // real restart in some fixed restarted cycle
00753     INT     Restart1    = Restart + 1;
00754     LONG    worksize    = (Restart+4)*(Restart+n)+1-n+Restart*n;
00755
00756     // Output some info for debugging
00757     if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (BLC) ...\n");
00758
00759     #if DEBUG_MODE > 0
00760         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00761         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00762     #endif
00763
00764     /* allocate memory and setup temp work space */
00765     REAL *work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00766
00767     /* check whether memory is enough for GMRES */
00768     while ( (work == NULL) && (Restart > 5) ) {
00769         Restart = Restart - 5;
00770         worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
00771         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00772         Restart1 = Restart + 1;
00773     }
00774
00775     if ( work == NULL ) {
00776         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00777         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00778     }
00779
00780     if ( PrtLvl > PRINT_MIN && Restart < restart ) {
00781         printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
00782     }
00783
00784     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00785     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00786     z = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00787     norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00788
00789     r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
00790     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00791     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00792     for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;
00793
00794     /* initialization */
00795     fasp_darray_cp(n, b->val, p[0]);
00796     fasp_blas_dblc_aAxy(-1.0, A, x->val, p[0]);
00797
00798     b_norm = fasp_blas_darray_norm2(n, b->val);
00799     r_norm = fasp_blas_darray_norm2(n, p[0]);
00800     norms[0] = r_norm;
00801
00802     if ( PrtLvl >= PRINT_SOME ) {
00803         ITS_PUTNORM("right-hand side", b_norm);
00804         ITS_PUTNORM("residual", r_norm);
00805     }
00806
00807     if ( b_norm > 0.0 ) den_norm = b_norm;
00808     else den_norm = r_norm;
00809
00810     epsilon = tol*den_norm;
00811
00812     // if initial residual is small, no need to iterate!
00813     if ( r_norm < epsilon || r_norm < 1e-12 * tol ) goto FINISHED;
00814
00815     if ( b_norm > 0.0 ) {
00816         fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
00817     }
00818     else {
00819         fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
00820     }
00821

```

```

00822     /* outer iteration cycle */
00823     while ( iter < MaxIt ) {
00824
00825         rs[0] = r_norm;
00826         r_norm_old = r_norm;
00827         if ( r_norm == 0.0 ) {
00828             fasp_mem_free(work); work = NULL;
00829             fasp_mem_free(p); p = NULL;
00830             fasp_mem_free(hh); hh = NULL;
00831             fasp_mem_free(norms); norms = NULL;
00832             fasp_mem_free(z); z = NULL;
00833             return iter;
00834         }
00835
00836         //-----//
00837         // adjust the restart parameter //
00838         //-----//
00839
00840         if ( cr > cr_max || iter == 0 ) {
00841             Restart = restart_max;
00842         }
00843         else if ( cr < cr_min ) {
00844             // Restart = Restart;
00845         }
00846         else {
00847             if ( Restart - d > restart_min ) Restart -= d;
00848             else Restart = restart_max;
00849         }
00850
00851         // Enter the cycle at the first iteration for at least one iteration
00852         t = 1.0 / r_norm;
00853         fasp_blas_darray_ax(n, t, p[0]);
00854         i = 0;
00855
00856         // RESTART CYCLE (right-preconditioning)
00857         while ( i < Restart && iter < MaxIt ) {
00858
00859             i ++; iter ++;
00860
00861             /* apply preconditioner */
00862             if ( pc == NULL )
00863                 fasp_darray_cp(n, p[i-1], z[i-1]);
00864             else
00865                 pc->fct(p[i-1], z[i-1], pc->data);
00866
00867             fasp_blas_dblc_mxv(A, z[i-1], p[i]);
00868
00869             /* modified Gram_Schmidt */
00870             for ( j = 0; j < i; j++ ) {
00871                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00872                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00873             }
00874             t = fasp_blas_darray_norm2(n, p[i]);
00875             hh[i][i-1] = t;
00876             if ( t != 0.0 ) {
00877                 t = 1.0 / t;
00878                 fasp_blas_darray_ax(n, t, p[i]);
00879             }
00880
00881             for ( j = 1; j < i; ++j ) {
00882                 t = hh[j-1][i-1];
00883                 hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00884                 hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00885             }
00886             t = hh[i][i-1] * hh[i][i-1];
00887             t += hh[i-1][i-1] * hh[i-1][i-1];
00888             gamma = sqrt(t);
00889             if (gamma == 0.0) gamma = epsmac;
00890             c[i-1] = hh[i-1][i-1] / gamma;
00891             s[i-1] = hh[i][i-1] / gamma;
00892             rs[i] = -s[i-1] * rs[i-1];
00893             rs[i-1] = c[i-1] * rs[i-1];
00894             hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00895
00896             r_norm = fabs(rs[i]);
00897             norms[iter] = r_norm;
00898
00899             if ( b_norm > 0 ) {
00900                 fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
00901                             norms[iter], norms[iter]/norms[iter-1]);
00902             }

```

```

00903         else {
00904             fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
00905                         norms[iter]/norms[iter-1]);
00906         }
00907
00908         /* Check: Exit the restart cycle? */
00909         if (r_norm <= epsilon && iter >= min_iter) break;
00910
00911     } /* end of restart cycle */
00912
00913     /* now compute solution, first solve upper triangular system */
00914
00915     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00916     for ( k = i-2; k >= 0; k-- ) {
00917         t = 0.0;
00918         for ( j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
00919
00920         t += rs[k];
00921         rs[k] = t / hh[k][k];
00922     }
00923
00924     fasp_darray_cp(n, z[i-1], r);
00925     fasp_blas_darray_ax(n, rs[i-1], r);
00926
00927     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);
00928
00929     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00930
00931     if ( r_norm <= epsilon && iter >= min_iter ) {
00932         fasp_darray_cp(n, b->val, r);
00933         fasp_blas_dblc_aAxy(-1.0, A, x->val, r);
00934         r_norm = fasp_blas_darray_norm2(n, r);
00935
00936         switch (StopType) {
00937             case STOP_REL_RES:
00938                 relres = r_norm/den_norm;
00939                 break;
00940             case STOP_REL_PRECRES:
00941                 if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
00942                 else pc->fct(r, p[0], pc->data);
00943                 r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00944                 relres = r_normb/den_norm;
00945                 break;
00946             case STOP_MOD_REL_RES:
00947                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00948                 relres = r_norm/normu;
00949                 break;
00950             default:
00951                 printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00952                 goto FINISHED;
00953         }
00954
00955         if ( relres <= tol ) {
00956             break;
00957         }
00958         else {
00959             if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
00960             fasp_darray_cp(n, r, p[0]); i = 0;
00961         }
00962     }
00963     /* end of convergence check */
00964
00965     /* compute residual vector and continue loop */
00966     for ( j = i; j > 0; j-- ) {
00967         rs[j-1] = -s[j-1]*rs[j];
00968         rs[j] = c[j-1]*rs[j];
00969     }
00970
00971     if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00972
00973     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00974
00975     if (i) {
00976         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00977         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00978     }
00979
00980     //-----//
00981     // compute the convergence rate //
00982     //-----//
00983     cr = r_norm / r_norm_old;

```

```

00984
00985     } /* end of iteration while loop */
00986
00987     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);
00988
00989 FINISHED:
00990     /*-----
00991     * Free some stuff
00992     *-----*/
00993     fasp_mem_free(work);    work = NULL;
00994     fasp_mem_free(p);       p = NULL;
00995     fasp_mem_free(hh);      hh = NULL;
00996     fasp_mem_free(norms);   norms = NULL;
00997     fasp_mem_free(z);       z = NULL;
00998
00999 #if DEBUG_MODE > 0
01000     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01001 #endif
01002
01003     if ( iter >= MaxIt )
01004         return ERROR_SOLVER_MAXIT;
01005     else
01006         return iter;
01007 }
01008
01036 INT fasp_solver_pvfgmres (mxv_matfree *mf,
01037                          dvector *b,
01038                          dvector *x,
01039                          precondition *pc,
01040                          const REAL tol,
01041                          const INT MaxIt,
01042                          const SHORT restart,
01043                          const SHORT StopType,
01044                          const SHORT PrtLvl)
01045 {
01046     const INT n = b->row;
01047     const INT min_iter = 0;
01048
01049     /*-----
01050     // Newly added parameters to monitor when
01051     // to change the restart parameter
01052     *-----*/
01053     const REAL cr_max = 0.99; // = cos(8^o) (experimental)
01054     const REAL cr_min = 0.174; // = cos(80^o) (experimental)
01055
01056     // local variables
01057     INT iter = 0;
01058     int i, j, k; // must be signed! -zcs
01059
01060     REAL epsmac = SMALLREAL;
01061     REAL r_norm, b_norm, den_norm;
01062     REAL epsilon, gamma, t;
01063
01064     REAL *c = NULL, *s = NULL, *rs = NULL;
01065     REAL *norms = NULL, *r = NULL;
01066     REAL **p = NULL, **hh = NULL, **z=NULL;
01067     REAL *work = NULL;
01068
01069     REAL cr = 1.0; // convergence rate
01070     REAL r_norm_old = 0.0; // save residual norm of previous restart cycle
01071     INT d = 3; // reduction for restart parameter
01072     INT restart_max = restart; // upper bound for restart in each restart cycle
01073     INT restart_min = 3; // lower bound for restart in each restart cycle
01074
01075     INT Restart = restart; // real restart in some fixed restarted cycle
01076     INT Restart1 = Restart + 1;
01077     LONG worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
01078
01079     // Output some info for debugging
01080     if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (MatFree) ...\n");
01081
01082 #if DEBUG_MODE > 0
01083     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01084     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01085 #endif
01086
01087     /* allocate memory and setup temp work space */
01088     work = (REAL *) fasp_mem_malloc(worksize, sizeof(REAL));
01089
01090     /* check whether memory is enough for GMRES */
01091     while ( (work == NULL) && (Restart > 5) ) {

```

```

01092     Restart = Restart - 5;
01093     worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
01094     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01095     Restart1 = Restart + 1;
01096 }
01097
01098 if ( work == NULL ) {
01099     printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
01100     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01101 }
01102
01103 if ( PrtLvl > PRINT_MIN && Restart < restart ) {
01104     printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
01105 }
01106
01107 p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01108 hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01109 z = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01110 norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01111
01112 r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
01113 for (i = 0; i < Restart1; i++) p[i] = s + Restart + i*n;
01114 for (i = 0; i < Restart1; i++) hh[i] = p[Restart] + n + i*Restart;
01115 for (i = 0; i < Restart1; i++) z[i] = hh[Restart] + Restart + i*n;
01116
01117 /* initialization */
01118 mf->fct(mf->data, x->val, p[0]);
01119 fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, p[0]);
01120
01121 b_norm = fasp_blas_darray_norm2(n, b->val);
01122 r_norm = fasp_blas_darray_norm2(n, p[0]);
01123 norms[0] = r_norm;
01124
01125 if ( PrtLvl >= PRINT_SOME ) {
01126     ITS_PUTNORM("right-hand side", b_norm);
01127     ITS_PUTNORM("residual", r_norm);
01128 }
01129
01130 if (b_norm > 0.0) den_norm = b_norm;
01131 else den_norm = r_norm;
01132
01133 epsilon = tol*den_norm;
01134
01135 /* outer iteration cycle */
01136 while (iter < MaxIt) {
01137     rs[0] = r_norm;
01138     r_norm_old = r_norm;
01139     if (r_norm == 0.0) {
01140         fasp_mem_free(work); work = NULL;
01141         fasp_mem_free(p); p = NULL;
01142         fasp_mem_free(hh); hh = NULL;
01143         fasp_mem_free(norms); norms = NULL;
01144         fasp_mem_free(z); z = NULL;
01145         return iter;
01146     }
01147
01148     //-----//
01149     // adjust the restart parameter //
01150     //-----//
01151
01152     if (cr > cr_max || iter == 0) {
01153         Restart = restart_max;
01154     }
01155     else if (cr < cr_min) {
01156         // Restart = Restart;
01157     }
01158     else {
01159         if ( Restart - d > restart_min ) Restart -= d;
01160         else Restart = restart_max;
01161     }
01162
01163     if (r_norm <= epsilon && iter >= min_iter) {
01164         mf->fct(mf->data, x->val, r);
01165         fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
01166         r_norm = fasp_blas_darray_norm2(n, r);
01167
01168         if (r_norm <= epsilon) {
01169             break;
01170         }
01171         else {
01172             if (PrtLvl >= PRINT_SOME) ITS_FACONV;

```

```

01173     }
01174 }
01175
01176 t = 1.0 / r_norm;
01177 fasp_blas_darray_ax(n, t, p[0]);
01178
01179 /* RESTART CYCLE (right-preconditioning) */
01180 i = 0;
01181 while (i < Restart && iter < MaxIt) {
01182     i ++; iter ++;
01183
01184     /* apply preconditioner */
01185     if (pc == NULL) fasp_darray_cp(n, p[i-1], z[i-1]);
01186     else pc->fct(p[i-1], z[i-1], pc->data);
01187
01188     mf->fct(mf->data, z[i-1], p[i]);
01189
01190     /* modified Gram_Schmidt */
01191     for (j = 0; j < i; j ++) {
01192         hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01193         fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01194     }
01195     t = fasp_blas_darray_norm2(n, p[i]);
01196     hh[i][i-1] = t;
01197     if (t != 0.0) {
01198         t = 1.0/t;
01199         fasp_blas_darray_ax(n, t, p[i]);
01200     }
01201
01202     for (j = 1; j < i; ++j) {
01203         t = hh[j-1][i-1];
01204         hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01205         hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01206     }
01207     t = hh[i][i-1]*hh[i][i-1];
01208     t += hh[i-1][i-1]*hh[i-1][i-1];
01209     gamma = sqrt(t);
01210     if (gamma == 0.0) gamma = epsmac;
01211     c[i-1] = hh[i-1][i-1] / gamma;
01212     s[i-1] = hh[i][i-1] / gamma;
01213     rs[i] = -s[i-1]*rs[i-1];
01214     rs[i-1] = c[i-1]*rs[i-1];
01215     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01216
01217     r_norm = fabs(rs[i]);
01218     norms[iter] = r_norm;
01219
01220     if (b_norm > 0) {
01221         fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
01222             norms[iter], norms[iter]/norms[iter-1]);
01223     }
01224     else {
01225         fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
01226             norms[iter]/norms[iter-1]);
01227     }
01228 }
01229
01230 /* Check: Exit restart cycle? */
01231 if (r_norm <= epsilon && iter >= min_iter) break;
01232
01233 } /* end of restart cycle */
01234
01235 /* now compute solution, first solve upper triangular system */
01236
01237 rs[i-1] = rs[i-1] / hh[i-1][i-1];
01238 for (k = i-2; k >= 0; k --) {
01239     t = 0.0;
01240     for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];
01241     t += rs[k];
01242     rs[k] = t / hh[k][k];
01243 }
01244
01245 fasp_darray_cp(n, z[i-1], r);
01246 fasp_blas_darray_ax(n, rs[i-1], r);
01247 for (j = i-2; j >= 0; j --) fasp_blas_darray_axpy(n, rs[j], z[j], r);
01248
01249 fasp_blas_darray_axpy(n, 1.0, r, x->val);
01250
01251 if (r_norm <= epsilon && iter >= min_iter) {
01252     mf->fct(mf->data, x->val, r);
01253 }

```

```

01254         fasp_blas_darray_axpy(n, 1.0, b->val, -1.0, r);
01255         r_norm = fasp_blas_darray_norm2(n, r);
01256
01257         if (r_norm <= epsilon) {
01258             break;
01259         }
01260         else {
01261             if (PrtLvl >= PRINT_SOME) ITS_FACONV;
01262             fasp_darray_cp(n, r, p[0]); i = 0;
01263         }
01264     } /* end of convergence check */
01265
01266     /* compute residual vector and continue loop */
01267     for (j = i; j > 0; j--) {
01268         rs[j-1] = -s[j-1]*rs[j];
01269         rs[j] = c[j-1]*rs[j];
01270     }
01271
01272     if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01273
01274     for (j = i-1; j > 0; j--) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01275
01276     if (i) {
01277         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01278         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01279     }
01280
01281     //-----//
01282     // compute the convergence rate //
01283     //-----//
01284     cr = r_norm / r_norm_old;
01285
01286 } /* end of iteration while loop */
01287
01288 if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);
01289
01290 /*-----*/
01291 * Free some stuff
01292 *-----*/
01293 fasp_mem_free(work); work = NULL;
01294 fasp_mem_free(p); p = NULL;
01295 fasp_mem_free(hh); hh = NULL;
01296 fasp_mem_free(norms); norms = NULL;
01297 fasp_mem_free(z); z = NULL;
01298
01299 #if DEBUG_MODE > 0
01300 printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01301 #endif
01302
01303 if (iter>=MaxIt)
01304     return ERROR_SOLVER_MAXIT;
01305 else
01306     return iter;
01307 }
01308
01309 /*-----*/
01310 /*-- End of File --*/
01311 /*-----*/

```

9.123 KryPvgmres.c File Reference

Krylov subspace methods – Preconditioned variable-restart GMRes.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- `INT fasp_solver_dcsr_pvgmres (dCSRmat *A, dvector *b, dvector *x, precondition *pc, const REAL tol, const INT MaxIt, const SHORT restart, const SHORT StopType, const SHORT PrtLvl)`

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

- `INT fasp_solver_dbsr_pvgmres` (`dBSRmat *A`, `dvector *b`, `dvector *x`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT restart`, `const SHORT StopType`, `const SHORT PrtLvl`)

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

- `INT fasp_solver_dblc_pvgmres` (`dBLCmat *A`, `dvector *b`, `dvector *x`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT restart`, `const SHORT StopType`, `const SHORT PrtLvl`)

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

- `INT fasp_solver_dstr_pvgmres` (`dSTRmat *A`, `dvector *b`, `dvector *x`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `const SHORT restart`, `const SHORT StopType`, `const SHORT PrtLvl`)

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

- `INT fasp_solver_pvgmres` (`mxv_matfree *mf`, `dvector *b`, `dvector *x`, `precond *pc`, `const REAL tol`, `const INT MaxIt`, `SHORT restart`, `const SHORT StopType`, `const SHORT PrtLvl`)

Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

9.123.1 Detailed Description

Krylov subspace methods – Preconditioned variable-restart GMRes.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

See [KrySPvgmres.c](#) for a safer version

Reference: A.H. Baker, E.R. Jessup, and Tz.V. Kolev A Simple Strategy for Varying the Restart Parameter in GMRES(m) Journal of Computational and Applied Mathematics, 230 (2009) pp. 751-761. UCRL-JRNL-235266.
Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KryPvgmres.c](#).

9.123.2 Function Documentation

9.123.2.1 fasp_solver_dblc_pvgmres()

```
INT fasp_solver_dblc_pvgmres (
    dBLCmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

Parameters

A	Pointer to dCSRmat : coefficient matrix
---	---

Parameters

<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/05/2013

Definition at line 757 of file [KryPvgmres.c](#).

9.123.2.2 fasp_solver_dbsr_pvgmres()

```

INT fasp_solver_dbsr_pvgmres (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

12/21/2011

Modified by Chensong Zhang on 04/06/2013: Add stop type support
Definition at line 413 of file [KryPvgmres.c](#).

9.123.2.3 fasp_solver_dcsr_pvgmres()

```

INT fasp_solver_dcsr_pvgmres (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/12/14

Modified by Chensong Zhang on 04/06/2013: Add stop type support Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate

Definition at line 66 of file [KryPvgmres.c](#).

9.123.2.4 fasp_solver_dstr_pvgmres()

```

INT fasp_solver_dstr_pvgmres (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Right preconditioned GMRES method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/12/14

Modified by Chensong Zhang on 04/06/2013: Add stop type support

Definition at line 1104 of file [KryPvgmres.c](#).

9.123.2.5 fasp_solver_pvgmres()

```

INT fasp_solver_pvgmres (
    mxv_matfree * mf,

```

```

dvector * b,
dvector * x,
precond * pc,
const REAL tol,
const INT MaxIt,
SHORT restart,
const SHORT StopType,
const SHORT PrtLvl )

```

Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>mf</i>	Pointer to mxv_matfree : spmv operation
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to precondition: structure of precondition
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type – DOES not support this parameter
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

2010/12/14

Modified by Feiteng Huang on 09/26/2012: matrix free Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate

Definition at line 1451 of file [KryPvgmres.c](#).

9.124 KryPvgmres.c

[Go to the documentation of this file.](#)

```

00001
00025 #include <math.h>
00026
00027 #include "fasp.h"
00028 #include "fasp_funcs.h"
00029
00030 /*-----*/
00031 /*--  Declare Private Functions  --*/
00032 /*-----*/
00033
00034 #include "KryUtil.inl"
00035
00036 /*-----*/
00037 /*--      Public Functions      --*/
00038 /*-----*/
00039

```

```

00066 INT fasp_solver_dcsr_pvgmres (dCSRmat      *A,
00067                                dvector      *b,
00068                                dvector      *x,
00069                                precondition  *pc,
00070                                const REAL    tol,
00071                                const INT     MaxIt,
00072                                const SHORT   restart,
00073                                const SHORT   StopType,
00074                                const SHORT   PrtLvl)
00075 {
00076     const INT    n      = b->row;
00077     const INT    MIN_ITER = 0;
00078     const REAL   epsmac  = SMALLREAL;
00079
00080     //-----//
00081     // Newly added parameters to monitor when //
00082     // to change the restart parameter //
00083     //-----//
00084     const REAL cr_max      = 0.99;    // = cos(8^o) (experimental)
00085     const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)
00086
00087     // local variables
00088     INT    iter      = 0;
00089     int    i, j, k; // must be signed! -zcs
00090
00091     REAL   r_norm, r_normb, gamma, t;
00092     REAL   absres0 = BIGREAL, absres = BIGREAL;
00093     REAL   relres  = BIGREAL, normu  = BIGREAL;
00094
00095     REAL   cr      = 1.0;    // convergence rate
00096     REAL   r_norm_old = 0.0; // save residual norm of previous restart cycle
00097     INT    d      = 3;      // reduction for restart parameter
00098     INT    restart_max = restart; // upper bound for restart in each restart cycle
00099     INT    restart_min = 3;    // lower bound for restart in each restart cycle
00100
00101     INT    Restart = restart; // real restart in some fixed restarted cycle
00102     INT    Restart1 = Restart + 1;
00103     unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;
00104
00105     // allocate temp memory (need about (restart+4)*n REAL numbers)
00106     REAL *c = NULL, *s = NULL, *rs = NULL;
00107     REAL *norms = NULL, *r = NULL, *w = NULL;
00108     REAL *work = NULL;
00109     REAL **p = NULL, **hh = NULL;
00110
00111     // Output some info for debugging
00112     if (PrtLvl > PRINT_NONE) printf("\nCalling VGMRes solver (CSR) ...\n");
00113
00114     #if DEBUG_MODE > 0
00115     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00116     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00117     #endif
00118
00119     /* allocate memory and setup temp work space */
00120     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00121
00122     /* check whether memory is enough for GMRES */
00123     while ( (work == NULL) && (Restart > 5) ) {
00124         Restart = Restart - 5;
00125         worksize = (Restart+4)*(Restart+n)+1-n;
00126         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00127         Restart1 = Restart + 1;
00128     }
00129
00130     if (work == NULL) {
00131         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00132         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00133     }
00134
00135     if (PrtLvl > PRINT_MIN && Restart < restart) {
00136         printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
00137     }
00138
00139     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00140     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00141     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00142
00143     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00144
00145     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00146

```

```

00147     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00148
00149     // r = b-A*x
00150     fasp_darray_cp(n, b->val, p[0]);
00151     fasp_blas_dcsr_aAxy(-1.0, A, x->val, p[0]);
00152
00153     r_norm = fasp_blas_darray_norm2(n, p[0]);
00154
00155     // compute initial residuals
00156     switch (StopType) {
00157     case STOP_REL_RES:
00158         absres0 = MAX(SMALLREAL, r_norm);
00159         relres = r_norm/absres0;
00160         break;
00161     case STOP_REL_PRECRES:
00162         if ( pc == NULL )
00163             fasp_darray_cp(n, p[0], r);
00164         else
00165             pc->fct(p[0], r, pc->data);
00166         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
00167         absres0 = MAX(SMALLREAL, r_normb);
00168         relres = r_normb/absres0;
00169         break;
00170     case STOP_MOD_REL_RES:
00171         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00172         absres0 = r_norm;
00173         relres = absres0/normu;
00174         break;
00175     default:
00176         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00177         goto FINISHED;
00178     }
00179
00180     // if initial residual is small, no need to iterate!
00181     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00182
00183     // output iteration information if needed
00184     fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0);
00185
00186     // store initial residual
00187     norms[0] = relres;
00188
00189     /* outer iteration cycle */
00190     while ( iter < MaxIt ) {
00191
00192         rs[0] = r_norm_old = r_norm;
00193
00194         t = 1.0 / r_norm;
00195
00196         fasp_blas_darray_ax(n, t, p[0]);
00197
00198         //-----//
00199         //   adjust the restart parameter   //
00200         //-----//
00201         if ( cr > cr_max || iter == 0 ) {
00202             Restart = restart_max;
00203         }
00204         else if ( cr < cr_min ) {
00205             // Restart = Restart;
00206         }
00207         else {
00208             if ( Restart - d > restart_min ) {
00209                 Restart -= d;
00210             }
00211             else {
00212                 Restart = restart_max;
00213             }
00214         }
00215
00216         /* RESTART CYCLE (right-preconditioning) */
00217         i = 0;
00218         while ( i < Restart && iter < MaxIt ) {
00219
00220             i++; iter++;
00221
00222             /* apply preconditioner */
00223             if (pc == NULL)
00224                 fasp_darray_cp(n, p[i-1], r);
00225             else
00226                 pc->fct(p[i-1], r, pc->data);
00227

```

```

00228     fasp_blas_dcsr_mxv(A, r, p[i]);
00229
00230     /* modified Gram_Schmidt */
00231     for (j = 0; j < i; j++) {
00232         hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00233         fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00234     }
00235     t = fasp_blas_darray_norm2(n, p[i]);
00236     hh[i][i-1] = t;
00237     if (t != 0.0) {
00238         t = 1.0/t;
00239         fasp_blas_darray_ax(n, t, p[i]);
00240     }
00241
00242     for (j = 1; j < i; ++j) {
00243         t = hh[j-1][i-1];
00244         hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00245         hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00246     }
00247     t = hh[i][i-1]*hh[i][i-1];
00248     t += hh[i-1][i-1]*hh[i-1][i-1];
00249
00250     gamma = sqrt(t);
00251     if (gamma == 0.0) gamma = epsmac;
00252     c[i-1] = hh[i-1][i-1] / gamma;
00253     s[i-1] = hh[i][i-1] / gamma;
00254     rs[i] = -s[i-1]*rs[i-1];
00255     rs[i-1] = c[i-1]*rs[i-1];
00256     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00257
00258     absres = r_norm = fabs(rs[i]);
00259
00260     relres = absres/absres0;
00261
00262     norms[iter] = relres;
00263
00264     // output iteration information if needed
00265     fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00266               norms[iter]/norms[iter-1]);
00267
00268     // should we exit restart cycle
00269     if ( relres < tol && iter >= MIN_ITER ) break;
00270
00271 } /* end of restart cycle */
00272
00273 /* now compute solution, first solve upper triangular system */
00274 rs[i-1] = rs[i-1] / hh[i-1][i-1];
00275 for (k = i-2; k >= 0; k--) {
00276     t = 0.0;
00277     for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
00278
00279     t += rs[k];
00280     rs[k] = t / hh[k][k];
00281 }
00282
00283 fasp_darray_cp(n, p[i-1], w);
00284
00285 fasp_blas_darray_ax(n, rs[i-1], w);
00286
00287 for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00288
00289 /* apply preconditioner */
00290 if ( pc == NULL )
00291     fasp_darray_cp(n, w, r);
00292 else
00293     pc->fct(w, r, pc->data);
00294
00295 fasp_blas_darray_axpy(n, 1.0, r, x->val);
00296
00297 // Check: prevent false convergence
00298 if ( relres < tol && iter >= MIN_ITER ) {
00299
00300     REAL computed_relres = relres;
00301
00302     // compute current residual
00303     fasp_darray_cp(n, b->val, r);
00304     fasp_blas_dcsr_aAxy(-1.0, A, x->val, r);
00305
00306     r_norm = fasp_blas_darray_norm2(n, r);
00307
00308     switch ( StopType ) {

```

```

00309         case STOP_REL_RES:
00310             absres = r_norm;
00311             relres = absres/absres0;
00312             break;
00313         case STOP_REL_PRECRES:
00314             if ( pc == NULL )
00315                 fasp_darray_cp(n, r, w);
00316             else
00317                 pc->fct(r, w, pc->data);
00318             absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00319             relres = absres/absres0;
00320             break;
00321         case STOP_MOD_REL_RES:
00322             absres = r_norm;
00323             normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00324             relres = absres/normu;
00325             break;
00326     }
00327
00328     norms[iter] = relres;
00329
00330     if ( relres < tol ) {
00331         break;
00332     }
00333     else {
00334         // Need to restart
00335         fasp_darray_cp(n, r, p[0]); i = 0;
00336     }
00337
00338     if ( PrtLvl >= PRINT_MORE ) {
00339         ITS_COMPRES(computed_relres); ITS_REALRES(relres);
00340     }
00341
00342     } /* end of convergence check */
00343
00344     /* compute residual vector and continue loop */
00345     for ( j = i; j > 0; j-- ) {
00346         rs[j-1] = -s[j-1]*rs[j];
00347         rs[j] = c[j-1]*rs[j];
00348     }
00349
00350     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00351
00352     for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00353
00354     if ( i ) {
00355         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00356         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00357     }
00358
00359     //-----//
00360     // compute the convergence rate //
00361     //-----//
00362     cr = r_norm / r_norm_old;
00363
00364     } /* end of iteration while loop */
00365
00366     FINISHED:
00367     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00368
00369     /*-----*/
00370     * Free some stuff
00371     *-----*/
00372     fasp_mem_free(work); work = NULL;
00373     fasp_mem_free(p); p = NULL;
00374     fasp_mem_free(hh); hh = NULL;
00375     fasp_mem_free(norms); norms = NULL;
00376
00377     #if DEBUG_MODE > 0
00378     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00379     #endif
00380
00381     if (iter>=MaxIt)
00382         return ERROR_SOLVER_MAXIT;
00383     else
00384         return iter;
00385 }
00386
00413 INT fasp_solver_dbsr_pvgmres (dBSRmat *A,
00414                               dvector *b,
00415                               dvector *x,

```



```

00416                                     precondition      *pc,
00417                                     const REAL      tol,
00418                                     const INT      MaxIt,
00419                                     const SHORT     restart,
00420                                     const SHORT     StopType,
00421                                     const SHORT     PrtLvl)
00422 {
00423     const INT      n          = b->row;
00424     const INT      MIN_ITER   = 0;
00425     const REAL     epsmac     = SMALLREAL;
00426
00427     //-----//
00428     // Newly added parameters to monitor when //
00429     // to change the restart parameter //
00430     //-----//
00431     const REAL cr_max      = 0.99;    // = cos(8^o) (experimental)
00432     const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)
00433
00434     // local variables
00435     INT      iter          = 0;
00436     int      i, j, k; // must be signed! -zcs
00437
00438     REAL     r_norm, r_normb, gamma, t;
00439     REAL     absres0 = BIGREAL, absres = BIGREAL;
00440     REAL     relres = BIGREAL, normu = BIGREAL;
00441
00442     REAL     cr           = 1.0;    // convergence rate
00443     REAL     r_norm_old   = 0.0;    // save residual norm of previous restart cycle
00444     INT      d            = 3;      // reduction for restart parameter
00445     INT      restart_max   = restart; // upper bound for restart in each restart cycle
00446     INT      restart_min  = 3;      // lower bound for restart in each restart cycle (should be small)
00447
00448     INT      Restart      = restart; // real restart in some fixed restarted cycle
00449     INT      Restart1     = Restart + 1;
00450     unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;
00451
00452     // allocate temp memory (need about (restart+4)*n REAL numbers)
00453     REAL *c = NULL, *s = NULL, *rs = NULL;
00454     REAL *norms = NULL, *r = NULL, *w = NULL;
00455     REAL *work = NULL;
00456     REAL **p = NULL, **hh = NULL;
00457
00458     // Output some info for debugging
00459     if (PrtLvl > PRINT_NONE) printf("\nCalling VGMRes solver (BSR) ...\n");
00460
00461 #if DEBUG_MODE > 0
00462     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00463     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00464 #endif
00465
00466     /* allocate memory and setup temp work space */
00467     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00468
00469     /* check whether memory is enough for GMRES */
00470     while ( (work == NULL) && (Restart > 5) ) {
00471         Restart = Restart - 5;
00472         worksize = (Restart+4)*(Restart+n)+1-n;
00473         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00474         Restart1 = Restart + 1;
00475     }
00476
00477     if (work == NULL) {
00478         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00479         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00480     }
00481
00482     if (PrtLvl > PRINT_MIN && Restart < restart) {
00483         printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
00484     }
00485
00486     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00487     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00488     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00489
00490     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00491
00492     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00493
00494     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00495
00496     // r = b-A*x

```

```

00497     fasp_darray_cp(n, b->val, p[0]);
00498     fasp_blas_dbsr_aAxy(-1.0, A, x->val, p[0]);
00499
00500     r_norm = fasp_blas_darray_norm2(n, p[0]);
00501
00502     // compute initial residuals
00503     switch (StopType) {
00504     case STOP_REL_RES:
00505         absres0 = MAX(SMALLREAL, r_norm);
00506         relres = r_norm/absres0;
00507         break;
00508     case STOP_REL_PRECRES:
00509         if ( pc == NULL )
00510             fasp_darray_cp(n, p[0], r);
00511         else
00512             pc->fct(p[0], r, pc->data);
00513         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00514         absres0 = MAX(SMALLREAL, r_normb);
00515         relres = r_normb/absres0;
00516         break;
00517     case STOP_MOD_REL_RES:
00518         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00519         absres0 = r_norm;
00520         relres = absres0/normu;
00521         break;
00522     default:
00523         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00524         goto FINISHED;
00525     }
00526
00527     // if initial residual is small, no need to iterate!
00528     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00529
00530     // output iteration information if needed
00531     fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0);
00532
00533     // store initial residual
00534     norms[0] = relres;
00535
00536     /* outer iteration cycle */
00537     while ( iter < MaxIt ) {
00538
00539         rs[0] = r_norm_old = r_norm;
00540
00541         t = 1.0 / r_norm;
00542
00543         fasp_blas_darray_ax(n, t, p[0]);
00544
00545         //-----//
00546         // adjust the restart parameter //
00547         //-----//
00548         if ( cr > cr_max || iter == 0 ) {
00549             Restart = restart_max;
00550         }
00551         else if ( cr < cr_min ) {
00552             // Restart = Restart;
00553         }
00554         else {
00555             if ( Restart - d > restart_min ) {
00556                 Restart -= d;
00557             }
00558             else {
00559                 Restart = restart_max;
00560             }
00561         }
00562
00563         /* RESTART CYCLE (right-preconditioning) */
00564         i = 0;
00565         while ( i < Restart && iter < MaxIt ) {
00566
00567             i++; iter++;
00568
00569             /* apply preconditioner */
00570             if (pc == NULL)
00571                 fasp_darray_cp(n, p[i-1], r);
00572             else
00573                 pc->fct(p[i-1], r, pc->data);
00574
00575             fasp_blas_dbsr_m xv(A, r, p[i]);
00576
00577             /* modified Gram_Schmidt */

```

```

00578         for (j = 0; j < i; j++) {
00579             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00580             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00581         }
00582         t = fasp_blas_darray_norm2(n, p[i]);
00583         hh[i][i-1] = t;
00584         if (t != 0.0) {
00585             t = 1.0/t;
00586             fasp_blas_darray_ax(n, t, p[i]);
00587         }
00588
00589         for (j = 1; j < i; ++j) {
00590             t = hh[j-1][i-1];
00591             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00592             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00593         }
00594         t = hh[i][i-1]*hh[i][i-1];
00595         t += hh[i-1][i-1]*hh[i-1][i-1];
00596
00597         gamma = sqrt(t);
00598         if (gamma == 0.0) gamma = epsmac;
00599         c[i-1] = hh[i-1][i-1] / gamma;
00600         s[i-1] = hh[i][i-1] / gamma;
00601         rs[i] = -s[i-1]*rs[i-1];
00602         rs[i-1] = c[i-1]*rs[i-1];
00603         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00604
00605         absres = r_norm = fabs(rs[i]);
00606
00607         relres = absres/absres0;
00608
00609         norms[iter] = relres;
00610
00611         // output iteration information if needed
00612         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00613                   norms[iter]/norms[iter-1]);
00614
00615         // should we exit restart cycle
00616         if ( relres < tol && iter >= MIN_ITER ) break;
00617     } /* end of restart cycle */
00618
00619     /* now compute solution, first solve upper triangular system */
00620     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00621     for (k = i-2; k >= 0; k--) {
00622         t = 0.0;
00623         for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
00624
00625         t += rs[k];
00626         rs[k] = t / hh[k][k];
00627     }
00628
00629     fasp_darray_cp(n, p[i-1], w);
00630
00631     fasp_blas_darray_ax(n, rs[i-1], w);
00632
00633     for (j = i-2; j >= 0; j--) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00634
00635     /* apply preconditioner */
00636     if ( pc == NULL )
00637         fasp_darray_cp(n, w, r);
00638     else
00639         pc->fct(w, r, pc->data);
00640
00641     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00642
00643     // Check: prevent false convergence
00644     if ( relres < tol && iter >= MIN_ITER ) {
00645         REAL computed_relres = relres;
00646
00647         // compute current residual
00648         fasp_darray_cp(n, b->val, r);
00649         fasp_blas_dbsr_aAxy(-1.0, A, x->val, r);
00650
00651         r_norm = fasp_blas_darray_norm2(n, r);
00652
00653         switch ( StopType ) {
00654             case STOP_REL_RES:
00655                 absres = r_norm;
00656                 relres = absres/absres0;
00657             }
00658     }

```

```

00659         break;
00660     case STOP_REL_PRECRES:
00661         if ( pc == NULL )
00662             fasp_darray_cp(n, r, w);
00663         else
00664             pc->fct(r, w, pc->data);
00665         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00666         relres = absres/absres0;
00667         break;
00668     case STOP_MOD_REL_RES:
00669         absres = r_norm;
00670         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00671         relres = absres/normu;
00672         break;
00673     }
00674
00675     norms[iter] = relres;
00676
00677     if ( relres < tol ) {
00678         break;
00679     }
00680     else {
00681         // Need to restart
00682         fasp_darray_cp(n, r, p[0]); i = 0;
00683     }
00684
00685     if ( PrtLvl >= PRINT_MORE ) {
00686         ITS_COMPRES(computed_relres); ITS_REALRES(relres);
00687     }
00688
00689     } /* end of convergence check */
00690
00691     /* compute residual vector and continue loop */
00692     for ( j = i; j > 0; j-- ) {
00693         rs[j-1] = -s[j-1]*rs[j];
00694         rs[j] = c[j-1]*rs[j];
00695     }
00696
00697     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00698
00699     for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00700
00701     if ( i ) {
00702         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00703         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00704     }
00705
00706     //-----//
00707     // compute the convergence rate //
00708     //-----//
00709     cr = r_norm / r_norm_old;
00710
00711     } /* end of iteration while loop */
00712
00713     FINISHED:
00714     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00715
00716     /*-----*/
00717     * Free some stuff
00718     *-----*/
00719     fasp_mem_free(work); work = NULL;
00720     fasp_mem_free(p); p = NULL;
00721     fasp_mem_free(hh); hh = NULL;
00722     fasp_mem_free(norms); norms = NULL;
00723
00724     #if DEBUG_MODE > 0
00725     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00726     #endif
00727
00728     if (iter>=MaxIt)
00729         return ERROR_SOLVER_MAXIT;
00730     else
00731         return iter;
00732 }
00733
00757 INT fasp_solver_dbic_pvgmres (dBLCmat *A,
00758                               dvector *b,
00759                               dvector *x,
00760                               precondition *pc,
00761                               const REAL tol,
00762                               const INT MaxIt,

```

```

00763             const SHORT restart,
00764             const SHORT StopType,
00765             const SHORT PrtLvl)
00766 {
00767     const INT    n        = b->row;
00768     const INT    MIN_ITER = 0;
00769     const REAL   epsmac   = SMALLREAL;
00770
00771     //-----//
00772     // Newly added parameters to monitor when //
00773     // to change the restart parameter //
00774     //-----//
00775     const REAL cr_max   = 0.99;    // = cos(8^o) (experimental)
00776     const REAL cr_min   = 0.174;   // = cos(80^o) (experimental)
00777
00778     // local variables
00779     INT    iter        = 0;
00780     int    i, j, k;    // must be signed! -zcs
00781
00782     REAL   r_norm, r_normb, gamma, t;
00783     REAL   absres0 = BIGREAL, absres = BIGREAL;
00784     REAL   relres  = BIGREAL, normu  = BIGREAL;
00785
00786     REAL   cr          = 1.0;      // convergence rate
00787     REAL   r_norm_old  = 0.0;      // save residual norm of previous restart cycle
00788     INT    d           = 3;        // reduction for restart parameter
00789     INT    restart_max  = restart;  // upper bound for restart in each restart cycle
00790     INT    restart_min  = 3;        // lower bound for restart in each restart cycle (should be small)
00791
00792     INT    Restart      = restart;  // real restart in some fixed restarted cycle
00793     INT    Restart1     = Restart + 1;
00794     unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;
00795
00796     // allocate temp memory (need about (restart+4)*n REAL numbers)
00797     REAL *c = NULL, *s = NULL, *rs = NULL;
00798     REAL *norms = NULL, *r = NULL, *w = NULL;
00799     REAL *work = NULL;
00800     REAL **p = NULL, **hh = NULL;
00801
00802     // Output some info for debugging
00803     if (PrtLvl > PRINT_NONE) printf("\nCalling VGMRes solver (BLC) ...\n");
00804
00805     #if DEBUG_MODE > 0
00806     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00807     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00808     #endif
00809
00810     /* allocate memory and setup temp work space */
00811     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00812
00813     /* check whether memory is enough for GMRES */
00814     while ( (work == NULL) && (Restart > 5) ) {
00815         Restart = Restart - 5;
00816         worksize = (Restart+4)*(Restart+n)+1-n;
00817         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
00818         Restart1 = Restart + 1;
00819     }
00820
00821     if (work == NULL) {
00822         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
00823         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00824     }
00825
00826     if (PrtLvl > PRINT_MIN && Restart < restart) {
00827         printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
00828     }
00829
00830     p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00831     hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
00832     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00833
00834     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
00835
00836     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
00837
00838     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
00839
00840     // r = b-A*x
00841     fasp_darray_cp(n, b->val, p[0]);
00842     fasp_blas_dblc_aAxy(-1.0, A, x->val, p[0]);
00843

```

```

00844     r_norm = fasp_blas_darray_norm2(n, p[0]);
00845
00846     // compute initial residuals
00847     switch (StopType) {
00848     case STOP_REL_RES:
00849         absres0 = MAX(SMALLREAL, r_norm);
00850         relres = r_norm/absres0;
00851         break;
00852     case STOP_REL_PRECRES:
00853         if ( pc == NULL )
00854             fasp_darray_cp(n, p[0], r);
00855         else
00856             pc->fct(p[0], r, pc->data);
00857         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00858         absres0 = MAX(SMALLREAL, r_normb);
00859         relres = r_normb/absres0;
00860         break;
00861     case STOP_MOD_REL_RES:
00862         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
00863         absres0 = r_norm;
00864         relres = absres0/normu;
00865         break;
00866     default:
00867         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00868         goto FINISHED;
00869     }
00870
00871     // if initial residual is small, no need to iterate!
00872     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
00873
00874     // output iteration information if needed
00875     fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0);
00876
00877     // store initial residual
00878     norms[0] = relres;
00879
00880     /* outer iteration cycle */
00881     while ( iter < MaxIt ) {
00882
00883         rs[0] = r_norm_old = r_norm;
00884
00885         t = 1.0 / r_norm;
00886
00887         fasp_blas_darray_ax(n, t, p[0]);
00888
00889         //-----//
00890         // adjust the restart parameter //
00891         //-----//
00892         if ( cr > cr_max || iter == 0 ) {
00893             Restart = restart_max;
00894         }
00895         else if ( cr < cr_min ) {
00896             // Restart = Restart;
00897         }
00898         else {
00899             if ( Restart - d > restart_min ) {
00900                 Restart -= d;
00901             }
00902             else {
00903                 Restart = restart_max;
00904             }
00905         }
00906
00907         /* RESTART CYCLE (right-preconditioning) */
00908         i = 0;
00909         while ( i < Restart && iter < MaxIt ) {
00910
00911             i++; iter++;
00912
00913             /* apply preconditioner */
00914             if (pc == NULL)
00915                 fasp_darray_cp(n, p[i-1], r);
00916             else
00917                 pc->fct(p[i-1], r, pc->data);
00918
00919             fasp_blas_dblc_mxv(A, r, p[i]);
00920
00921             /* modified Gram_Schmidt */
00922             for (j = 0; j < i; j++) {
00923                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00924                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);

```

```

00925     }
00926     t = fasp_blas_darray_norm2(n, p[i]);
00927     hh[i][i-1] = t;
00928     if (t != 0.0) {
00929         t = 1.0/t;
00930         fasp_blas_darray_ax(n, t, p[i]);
00931     }
00932
00933     for (j = 1; j < i; ++j) {
00934         t = hh[j-1][i-1];
00935         hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00936         hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00937     }
00938     t = hh[i][i-1]*hh[i][i-1];
00939     t += hh[i-1][i-1]*hh[i-1][i-1];
00940
00941     gamma = sqrt(t);
00942     if (gamma == 0.0) gamma = epsmac;
00943     c[i-1] = hh[i-1][i-1] / gamma;
00944     s[i-1] = hh[i][i-1] / gamma;
00945     rs[i] = -s[i-1]*rs[i-1];
00946     rs[i-1] = c[i-1]*rs[i-1];
00947     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00948
00949     absres = r_norm = fabs(rs[i]);
00950
00951     relres = absres/absres0;
00952
00953     norms[iter] = relres;
00954
00955     // output iteration information if needed
00956     fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00957               norms[iter]/norms[iter-1]);
00958
00959     // should we exit restart cycle
00960     if ( relres < tol && iter >= MIN_ITER ) break;
00961
00962 } /* end of restart cycle */
00963
00964 /* now compute solution, first solve upper triangular system */
00965 rs[i-1] = rs[i-1] / hh[i-1][i-1];
00966 for (k = i-2; k >= 0; k --) {
00967     t = 0.0;
00968     for (j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
00969
00970     t += rs[k];
00971     rs[k] = t / hh[k][k];
00972 }
00973
00974 fasp_darray_cp(n, p[i-1], w);
00975
00976 fasp_blas_darray_ax(n, rs[i-1], w);
00977
00978 for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00979
00980 /* apply preconditioner */
00981 if ( pc == NULL )
00982     fasp_darray_cp(n, w, r);
00983 else
00984     pc->fct(w, r, pc->data);
00985
00986 fasp_blas_darray_axpy(n, 1.0, r, x->val);
00987
00988 // Check: prevent false convergence
00989 if ( relres < tol && iter >= MIN_ITER ) {
00990
00991     REAL computed_relres = relres;
00992
00993     // compute current residual
00994     fasp_darray_cp(n, b->val, r);
00995     fasp_blas_dbic_aAxy(-1.0, A, x->val, r);
00996
00997     r_norm = fasp_blas_darray_norm2(n, r);
00998
00999     switch ( StopType ) {
01000         case STOP_REL_RES:
01001             absres = r_norm;
01002             relres = absres/absres0;
01003             break;
01004         case STOP_REL_PRECRES:
01005             if ( pc == NULL )

```

```

01006         fasp_darray_cp(n, r, w);
01007     else
01008         pc->fct(r, w, pc->data);
01009         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
01010         relres = absres/absres0;
01011         break;
01012     case STOP_MOD_REL_RES:
01013         absres = r_norm;
01014         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
01015         relres = absres/normu;
01016         break;
01017     }
01018     norms[iter] = relres;
01019
01020     if ( relres < tol ) {
01021         break;
01022     }
01023     else {
01024         // Need to restart
01025         fasp_darray_cp(n, r, p[0]); i = 0;
01026     }
01027
01028     if ( PrtLvl >= PRINT_MORE ) {
01029         ITS_COMPRES(computed_relres); ITS_REALRES(relres);
01030     }
01031 }
01032
01033 } /* end of convergence check */
01034
01035 /* compute residual vector and continue loop */
01036 for ( j = i; j > 0; j-- ) {
01037     rs[j-1] = -s[j-1]*rs[j];
01038     rs[j] = c[j-1]*rs[j];
01039 }
01040
01041 if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01042
01043 for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01044
01045 if ( i ) {
01046     fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01047     fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01048 }
01049
01050 //-----//
01051 // compute the convergence rate //
01052 //-----//
01053 cr = r_norm / r_norm_old;
01054
01055 } /* end of iteration while loop */
01056
01057 FINISHED:
01058     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01059
01060     /*-----*/
01061     * Free some stuff
01062     *-----*/
01063     fasp_mem_free(work); work = NULL;
01064     fasp_mem_free(p); p = NULL;
01065     fasp_mem_free(hh); hh = NULL;
01066     fasp_mem_free(norms); norms = NULL;
01067
01068     #if DEBUG_MODE > 0
01069     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01070     #endif
01071
01072     if (iter>=MaxIt)
01073         return ERROR_SOLVER_MAXIT;
01074     else
01075         return iter;
01076 }
01077
01104 INT fasp_solver_dstr_pvgmres (dSTRmat *A,
01105                             dvector *b,
01106                             dvector *x,
01107                             precondition *pc,
01108                             const REAL tol,
01109                             const INT MaxIt,
01110                             const SHORT restart,
01111                             const SHORT StopType,
01112                             const SHORT PrtLvl)

```



```

01113 {
01114     const INT    n          = b->row;
01115     const INT    MIN_ITER   = 0;
01116     const REAL   epsmac     = SMALLREAL;
01117
01118     //-----//
01119     // Newly added parameters to monitor when //
01120     // to change the restart parameter //
01121     //-----//
01122     const REAL cr_max      = 0.99;    // = cos(8^o) (experimental)
01123     const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)
01124
01125     // local variables
01126     INT    iter            = 0;
01127     int    i, j, k; // must be signed! -zcs
01128
01129     REAL   r_norm, r_normb, gamma, t;
01130     REAL   absres0 = BIGREAL, absres = BIGREAL;
01131     REAL   relres  = BIGREAL, normu  = BIGREAL;
01132
01133     REAL   cr          = 1.0;    // convergence rate
01134     REAL   r_norm_old  = 0.0;    // save residual norm of previous restart cycle
01135     INT    d           = 3;      // reduction for restart parameter
01136     INT    restart_max  = restart; // upper bound for restart in each restart cycle
01137     INT    restart_min  = 3;      // lower bound for restart in each restart cycle (should be small)
01138
01139     INT    Restart      = restart; // real restart in some fixed restarted cycle
01140     INT    Restart1     = Restart + 1;
01141     unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;
01142
01143     // allocate temp memory (need about (restart+4)*n REAL numbers)
01144     REAL *c = NULL, *s = NULL, *rs = NULL;
01145     REAL *norms = NULL, *r = NULL, *w = NULL;
01146     REAL *work = NULL;
01147     REAL **p = NULL, **hh = NULL;
01148
01149     // Output some info for debugging
01150     if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (STR) ...\n");
01151
01152     #if DEBUG_MODE > 0
01153         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01154         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01155     #endif
01156
01157     /* allocate memory and setup temp work space */
01158     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01159
01160     /* check whether memory is enough for GMRES */
01161     while ( (work == NULL) && (Restart > 5) ) {
01162         Restart = Restart - 5;
01163         worksize = (Restart+4)*(Restart+n)+1-n;
01164         work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01165         Restart1 = Restart + 1;
01166     }
01167
01168     if ( work == NULL ) {
01169         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
01170         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01171     }
01172
01173     if ( PrtLvl > PRINT_MIN && Restart < restart ) {
01174         printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
01175     }
01176
01177     p      = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01178     hh     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01179     norms  = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01180
01181     r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
01182
01183     for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
01184
01185     for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
01186
01187     // r = b-A*x
01188     fasp_darray_cp(n, b->val, p[0]);
01189     fasp_blas_dstr_aAxy(-1.0, A, x->val, p[0]);
01190
01191     r_norm = fasp_blas_darray_norm2(n, p[0]);
01192
01193     // compute initial residuals

```

```

01194     switch (StopType) {
01195     case STOP_REL_RES:
01196         absres0 = MAX(SMALLREAL, r_norm);
01197         relres = r_norm/absres0;
01198         break;
01199     case STOP_REL_PRECRES:
01200         if ( pc == NULL )
01201             fasp_darray_cp(n, p[0], r);
01202         else
01203             pc->fct(p[0], r, pc->data);
01204         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
01205         absres0 = MAX(SMALLREAL, r_normb);
01206         relres = r_normb/absres0;
01207         break;
01208     case STOP_MOD_REL_RES:
01209         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
01210         absres0 = r_norm;
01211         relres = absres0/normu;
01212         break;
01213     default:
01214         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01215         goto FINISHED;
01216     }
01217
01218     // if initial residual is small, no need to iterate!
01219     if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;
01220
01221     // output iteration information if needed
01222     fasp_itinfo(PrtLvl, StopType, 0, relres, absres0, 0);
01223
01224     // store initial residual
01225     norms[0] = relres;
01226
01227     /* outer iteration cycle */
01228     while ( iter < MaxIt ) {
01229
01230         rs[0] = r_norm_old = r_norm;
01231
01232         t = 1.0 / r_norm;
01233
01234         fasp_blas_darray_ax(n, t, p[0]);
01235
01236         //-----//
01237         // adjust the restart parameter //
01238         //-----//
01239         if ( cr > cr_max || iter == 0 ) {
01240             Restart = restart_max;
01241         }
01242         else if ( cr < cr_min ) {
01243             // Restart = Restart;
01244         }
01245         else {
01246             if ( Restart - d > restart_min ) {
01247                 Restart -= d;
01248             }
01249             else {
01250                 Restart = restart_max;
01251             }
01252         }
01253
01254         /* RESTART CYCLE (right-preconditioning) */
01255         i = 0;
01256         while ( i < Restart && iter < MaxIt ) {
01257
01258             i++; iter++;
01259
01260             /* apply preconditioner */
01261             if (pc == NULL)
01262                 fasp_darray_cp(n, p[i-1], r);
01263             else
01264                 pc->fct(p[i-1], r, pc->data);
01265
01266             fasp_blas_dstr_mxv(A, r, p[i]);
01267
01268             /* modified Gram_Schmidt */
01269             for (j = 0; j < i; j++) {
01270                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01271                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01272             }
01273             t = fasp_blas_darray_norm2(n, p[i]);
01274             hh[i][i-1] = t;

```

```

01275         if (t != 0.0) {
01276             t = 1.0/t;
01277             fasp_blas_darray_ax(n, t, p[i]);
01278         }
01279
01280         for (j = 1; j < i; ++j) {
01281             t = hh[j-1][i-1];
01282             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01283             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01284         }
01285         t = hh[i][i-1]*hh[i][i-1];
01286         t += hh[i-1][i-1]*hh[i-1][i-1];
01287
01288         gamma = sqrt(t);
01289         if (gamma == 0.0) gamma = epsmac;
01290         c[i-1] = hh[i-1][i-1] / gamma;
01291         s[i-1] = hh[i][i-1] / gamma;
01292         rs[i] = -s[i-1]*rs[i-1];
01293         rs[i-1] = c[i-1]*rs[i-1];
01294         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01295
01296         absres = r_norm = fabs(rs[i]);
01297
01298         relres = absres/absres0;
01299
01300         norms[iter] = relres;
01301
01302         // output iteration information if needed
01303         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
01304                 norms[iter]/norms[iter-1]);
01305
01306         // should we exit restart cycle
01307         if ( relres < tol && iter >= MIN_ITER ) break;
01308
01309     } /* end of restart cycle */
01310
01311     /* now compute solution, first solve upper triangular system */
01312     rs[i-1] = rs[i-1] / hh[i-1][i-1];
01313     for (k = i-2; k >= 0; k--) {
01314         t = 0.0;
01315         for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
01316
01317         t += rs[k];
01318         rs[k] = t / hh[k][k];
01319     }
01320
01321     fasp_darray_cp(n, p[i-1], w);
01322
01323     fasp_blas_darray_ax(n, rs[i-1], w);
01324
01325     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01326
01327     /* apply preconditioner */
01328     if ( pc == NULL )
01329         fasp_darray_cp(n, w, r);
01330     else
01331         pc->fct(w, r, pc->data);
01332
01333     fasp_blas_darray_axpy(n, 1.0, r, x->val);
01334
01335     // Check: prevent false convergence
01336     if ( relres < tol && iter >= MIN_ITER ) {
01337
01338         REAL computed_relres = relres;
01339
01340         // compute current residual
01341         fasp_darray_cp(n, b->val, r);
01342         fasp_blas_dstr_aAxy(-1.0, A, x->val, r);
01343
01344         r_norm = fasp_blas_darray_norm2(n, r);
01345
01346         switch ( StopType ) {
01347             case STOP_REL_RES:
01348                 absres = r_norm;
01349                 relres = absres/absres0;
01350                 break;
01351             case STOP_REL_PRECRES:
01352                 if ( pc == NULL )
01353                     fasp_darray_cp(n, r, w);
01354                 else
01355                     pc->fct(r, w, pc->data);

```

```

01356         absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
01357         relres = absres/absres0;
01358         break;
01359     case STOP_MOD_REL_RES:
01360         absres = r_norm;
01361         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
01362         relres = absres/normu;
01363         break;
01364     }
01365     norms[iter] = relres;
01366
01367     if ( relres < tol ) {
01368         break;
01369     }
01370     else {
01371         // Need to restart
01372         fasp_darray_cp(n, r, p[0]); i = 0;
01373     }
01374
01375     if ( PrtLvl >= PRINT_MORE ) {
01376         ITS_COMPRES(computed_relres); ITS_REALRES(relres);
01377     }
01378 }
01379
01380 /* end of convergence check */
01381
01382 /* compute residual vector and continue loop */
01383 for ( j = i; j > 0; j-- ) {
01384     rs[j-1] = -s[j-1]*rs[j];
01385     rs[j] = c[j-1]*rs[j];
01386 }
01387
01388 if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01389
01390 for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01391
01392 if ( i ) {
01393     fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01394     fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01395 }
01396
01397 //-----//
01398 // compute the convergence rate //
01399 //-----//
01400 cr = r_norm / r_norm_old;
01401
01402 } /* end of iteration while loop */
01403
01404 FINISHED:
01405 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01406
01407 /*-----*/
01408 * Free some stuff
01409 *-----*/
01410 fasp_mem_free(work); work = NULL;
01411 fasp_mem_free(p); p = NULL;
01412 fasp_mem_free(hh); hh = NULL;
01413 fasp_mem_free(norms); norms = NULL;
01414
01415 #if DEBUG_MODE > 0
01416 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01417 #endif
01418
01419 if (iter>=MaxIt)
01420     return ERROR_SOLVER_MAXIT;
01421 else
01422     return iter;
01423 }
01424
01451 INT fasp_solver_pvgmres (mxv_matfree *mf,
01452                         dvector *b,
01453                         dvector *x,
01454                         precondition *pc,
01455                         const REAL tol,
01456                         const INT MaxIt,
01457                         SHORT restart,
01458                         const SHORT StopType,
01459                         const SHORT PrtLvl)
01460 {
01461     const INT n = b->row;
01462     const INT min_iter = 0;

```

```

01463
01464 //-----//
01465 // Newly added parameters to monitor when //
01466 // to change the restart parameter //
01467 //-----//
01468 const REAL cr_max = 0.99; // = cos(8^o) (experimental)
01469 const REAL cr_min = 0.174; // = cos(80^o) (experimental)
01470
01471 // local variables
01472 INT iter = 0;
01473 int i, j, k; // must be signed! -zcs
01474
01475 REAL epsmac = SMALLREAL;
01476 REAL r_norm, b_norm, den_norm;
01477 REAL epsilon, gamma, t;
01478
01479 REAL *c = NULL, *s = NULL, *rs = NULL;
01480 REAL *norms = NULL, *r = NULL, *w = NULL;
01481 REAL **p = NULL, **hh = NULL;
01482 REAL *work = NULL;
01483
01484 REAL cr = 1.0; // convergence rate
01485 REAL r_norm_old = 0.0; // save residual norm of previous restart cycle
01486 INT d = 3; // reduction for restart parameter
01487 INT restart_max = restart; // upper bound for restart in each restart cycle
01488 INT restart_min = 3; // lower bound for restart in each restart cycle
01489
01490 INT Restart = restart; // real restart in some fixed restarted cycle
01491 INT Restart1 = Restart + 1;
01492 unsigned LONG worksize = (restart+4)*(restart+n)+1-n;
01493
01494 // Output some info for debugging
01495 if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (MatFree) ...\n");
01496
01497 #if DEBUG_MODE > 0
01498 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01499 printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01500 #endif
01501
01502 /* allocate memory and setup temp work space */
01503 work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01504
01505 /* check whether memory is enough for GMRES */
01506 while ( (work == NULL) && (Restart > 5) ) {
01507     Restart = Restart - 5;
01508     worksize = (Restart+4)*(Restart+n)+1-n;
01509     work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
01510     Restart1 = Restart + 1;
01511 }
01512
01513 if ( work == NULL ) {
01514     printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__ );
01515     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01516 }
01517
01518 if ( PrtLvl > PRINT_MIN && Restart < restart ) {
01519     printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
01520 }
01521
01522 p = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01523 hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
01524 norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01525
01526 r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
01527 for (i = 0; i < Restart1; i++) p[i] = s + Restart + i*n;
01528 for (i = 0; i < Restart1; i++) hh[i] = p[Restart] + n + i*Restart;
01529
01530 /* initialization */
01531 mf->fct(mf->data, x->val, p[0]);
01532 fasp_blas_darray_axpy(n, 1.0, b->val, -1.0, p[0]);
01533
01534 b_norm = fasp_blas_darray_norm2(n, b->val);
01535 r_norm = fasp_blas_darray_norm2(n, p[0]);
01536 norms[0] = r_norm;
01537
01538 if ( PrtLvl >= PRINT_SOME ) {
01539     ITS_PUTNORM("right-hand side", b_norm);
01540     ITS_PUTNORM("residual", r_norm);
01541 }
01542
01543 if (b_norm > 0.0) den_norm = b_norm;

```

```

01544     else                den_norm = r_norm;
01545
01546     epsilon = tol*den_norm;
01547
01548     /* outer iteration cycle */
01549     while (iter < MaxIt) {
01550         rs[0] = r_norm;
01551         r_norm_old = r_norm;
01552         if (r_norm == 0.0) {
01553             fasp_mem_free(work); work = NULL;
01554             fasp_mem_free(p); p = NULL;
01555             fasp_mem_free(hh); hh = NULL;
01556             fasp_mem_free(norms); norms = NULL;
01557             return iter;
01558         }
01559
01560         //-----//
01561         //  adjust the restart parameter  //
01562         //-----//
01563
01564         if (cr > cr_max || iter == 0) {
01565             Restart = restart_max;
01566         }
01567         else if (cr < cr_min) {
01568             // Restart = Restart;
01569         }
01570         else {
01571             if (Restart - d > restart_min) {
01572                 Restart -= d;
01573             }
01574             else {
01575                 Restart = restart_max;
01576             }
01577         }
01578
01579         if (r_norm <= epsilon && iter >= min_iter) {
01580             mf->fct(mf->data, x->val, r);
01581             fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
01582             r_norm = fasp_blas_darray_norm2(n, r);
01583
01584             if (r_norm <= epsilon) {
01585                 break;
01586             }
01587             else {
01588                 if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
01589             }
01590         }
01591
01592         t = 1.0 / r_norm;
01593
01594         //for (j = 0; j < n; j++) p[0][j] *= t;
01595         fasp_blas_darray_ax(n, t, p[0]);
01596
01597         /* RESTART CYCLE (right-preconditioning) */
01598         i = 0;
01599         while (i < Restart && iter < MaxIt) {
01600
01601             i++; iter++;
01602
01603             /* apply preconditioner */
01604             if (pc == NULL)
01605                 fasp_darray_cp(n, p[i-1], r);
01606             else
01607                 pc->fct(p[i-1], r, pc->data);
01608
01609             mf->fct(mf->data, r, p[i]);
01610
01611             /* modified Gram_Schmidt */
01612             for (j = 0; j < i; j++) {
01613                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01614                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01615             }
01616             t = fasp_blas_darray_norm2(n, p[i]);
01617             hh[i][i-1] = t;
01618             if (t != 0.0) {
01619                 t = 1.0/t;
01620                 //for (j = 0; j < n; j++) p[i][j] *= t;
01621                 fasp_blas_darray_ax(n, t, p[i]);
01622             }
01623
01624             for (j = 1; j < i; ++j) {

```

```

01625         t = hh[j-1][i-1];
01626         hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01627         hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01628     }
01629     t= hh[i][i-1]*hh[i][i-1];
01630     t+= hh[i-1][i-1]*hh[i-1][i-1];
01631     gamma = sqrt(t);
01632     if (gamma == 0.0) gamma = epsmac;
01633     c[i-1] = hh[i-1][i-1] / gamma;
01634     s[i-1] = hh[i][i-1] / gamma;
01635     rs[i] = -s[i-1]*rs[i-1];
01636     rs[i-1] = c[i-1]*rs[i-1];
01637     hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01638     r_norm = fabs(rs[i]);
01639
01640     norms[iter] = r_norm;
01641
01642     if (b_norm > 0 ) {
01643         fasp_itinfo(PrtLvl,StopType,iter,norms[iter]/b_norm,
01644             norms[iter],norms[iter]/norms[iter-1]);
01645     }
01646     else {
01647         fasp_itinfo(PrtLvl,StopType,iter,norms[iter],norms[iter],
01648             norms[iter]/norms[iter-1]);
01649     }
01650
01651     /* should we exit restart cycle? */
01652     if (r_norm <= epsilon && iter >= min_iter) break;
01653
01654 } /* end of restart cycle */
01655
01656 /* now compute solution, first solve upper triangular system */
01657
01658 rs[i-1] = rs[i-1] / hh[i-1][i-1];
01659 for (k = i-2; k >= 0; k --) {
01660     t = 0.0;
01661     for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];
01662
01663     t += rs[k];
01664     rs[k] = t / hh[k][k];
01665 }
01666 fasp_darray_cp(n, p[i-1], w);
01667 //for (j = 0; j < n; j ++) w[j] *= rs[i-1];
01668 fasp_blas_darray_ax(n, rs[i-1], w);
01669 for (j = i-2; j >= 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01670
01671 /* apply preconditioner */
01672 if (pc == NULL)
01673     fasp_darray_cp(n, w, r);
01674 else
01675     pc->fct(w, r, pc->data);
01676
01677 fasp_blas_darray_axpy(n, 1.0, r, x->val);
01678
01679 if (r_norm <= epsilon && iter >= min_iter) {
01680     mf->fct(mf->data, x->val, r);
01681     fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
01682     r_norm = fasp_blas_darray_norm2(n, r);
01683
01684     if (r_norm <= epsilon) {
01685         break;
01686     }
01687     else {
01688         if (PrtLvl >= PRINT_SOME) ITS_FACONV;
01689         fasp_darray_cp(n, r, p[0]); i = 0;
01690     }
01691 } /* end of convergence check */
01692
01693 /* compute residual vector and continue loop */
01694 for (j = i; j > 0; j --) {
01695     rs[j-1] = -s[j-1]*rs[j];
01696     rs[j] = c[j-1]*rs[j];
01697 }
01698
01699 if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01700
01701 for (j = i-1; j > 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01702
01703 if (i) {
01704     fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01705     fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);

```

```

01706     }
01707
01708     //-----//
01709     //  compute the convergence rate  //
01710     //-----//
01711     cr = r_norm / r_norm_old;
01712
01713     } /* end of iteration while loop */
01714
01715     if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);
01716
01717     /*-----
01718  * Free some stuff
01719  *-----*/
01720     fasp_mem_free(work); work = NULL;
01721     fasp_mem_free(p); p = NULL;
01722     fasp_mem_free(hh); hh = NULL;
01723     fasp_mem_free(norms); norms = NULL;
01724
01725     #if DEBUG_MODE > 0
01726     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01727     #endif
01728
01729     if (iter>=MaxIt)
01730         return ERROR_SOLVER_MAXIT;
01731     else
01732         return iter;
01733 }
01734
01735 /*-----*/
01736 /*--      End of File      --*/
01737 /*-----*/

```

9.125 KrySPbcgs.c File Reference

Krylov subspace methods – Preconditioned BiCGstab with safety net.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- **INT fasp_solver_dcsr_spbcgs** (const **dCSRmat** *A, const **dvector** *b, **dvector** *u, **precond** *pc, const **REAL** tol, const **INT** MaxIt, const **SHORT** StopType, const **SHORT** PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ with safety net.
- **INT fasp_solver_dbsr_spbcgs** (const **dBSRmat** *A, const **dvector** *b, **dvector** *u, **precond** *pc, const **REAL** tol, const **INT** MaxIt, const **SHORT** StopType, const **SHORT** PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ with safety net.
- **INT fasp_solver_dblc_spbcgs** (const **dBLCmat** *A, const **dvector** *b, **dvector** *u, **precond** *pc, const **REAL** tol, const **INT** MaxIt, const **SHORT** StopType, const **SHORT** PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ with safety net.
- **INT fasp_solver_dstr_spbcgs** (const **dSTRmat** *A, const **dvector** *b, **dvector** *u, **precond** *pc, const **REAL** tol, const **INT** MaxIt, const **SHORT** StopType, const **SHORT** PrtLvl)
Preconditioned BiCGstab method for solving $Au=b$ with safety net.

9.125.1 Detailed Description

Krylov subspace methods – Preconditioned BiCGstab with safety net.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

The 'best' iterative solution will be saved and used upon exit; See [KryPbcgs.c](#) for a version without safety net

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Update this version with the new BiCGstab implementation! –Chensong TODO: Use one single function for all!

–Chensong

Definition in file [KrySPbcgs.c](#).

9.125.2 Function Documentation

9.125.2.1 fasp_solver_dblc_spmcgs()

```
INT fasp_solver_dblc_spmcgs (
    const dBLCmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dBLCmat : the coefficient matrix
<i>b</i>	Pointer to dvector : the right hand side
<i>u</i>	Pointer to dvector : the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/31/2013

Definition at line [843](#) of file [KrySPbcgs.c](#).

9.125.2.2 fasp_solver_dbsr_spgbcs()

```

INT fasp_solver_dbsr_spgbcs (
    const dBSRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dBSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector : the right hand side
<i>u</i>	Pointer to dvector : the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/31/2013

Definition at line [452](#) of file [KrySPbcs.c](#).

9.125.2.3 fasp_solver_dcsr_spgbcs()

```

INT fasp_solver_dcsr_spgbcs (
    const dCSRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/31/2013

Definition at line 61 of file [KrySPbcgs.c](#).

9.125.2.4 fasp_solver_dstr_spbcs()

```

INT fasp_solver_dstr_spbcs (
    const dSTRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned BiCGstab method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dSTRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/31/2013

Definition at line 1234 of file KrySPbcgs.c.

9.126 KrySPbcgs.c

[Go to the documentation of this file.](#)

```

00001
00025 #include <math.h>
00026
00027 #include "fasp.h"
00028 #include "fasp_funcs.h"
00029
00030 /*-----*/
00031 /*--  Declare Private Functions  --*/
00032 /*-----*/
00033
00034 #include "KryUtil.inl"
00035
00036 /*-----*/
00037 /*--      Public Functions      --*/
00038 /*-----*/
00039
00061 INT fasp_solver_dcsr_spgbcs (const dCSRmat *A,
00062                             const dvector *b,
00063                             dvector *u,
00064                             precondition *pc,
00065                             const REAL tol,
00066                             const INT MaxIt,
00067                             const SHORT StopType,
00068                             const SHORT PrtLvl)
00069 {
00070     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00071     const INT m = b->row;
00072     const REAL maxdiff = tol*STAG_RATIO; // stagantion tolerance
00073     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00074     const REAL TOL_s = tol*1e-2; // tolerance for norm(p)
00075
00076     // local variables
00077     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00078     REAL alpha, beta, omega, templ, temp2;
00079     REAL absres0 = BIGREAL, absres = BIGREAL;
00080     REAL relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00081     REAL reldiff, factor, normd, tempr, normuinf;
00082     REAL *uval = u->val, *bval = b->val;
00083     INT iter_best = 0; // initial best known iteration
00084     REAL absres_best = BIGREAL; // initial best known residual
00085
00086     // allocate temp memory (need 8*m REAL)
00087     REAL *work = (REAL *)fasp_mem_malloc(9*m, sizeof(REAL));
00088     REAL *p = work, *z = work + m, *r = z + m, *t = r + m;
00089     REAL *rho = t + m, *pp = rho + m, *s = pp + m, *sp = s + m, *u_best = sp + m;
00090
00091     // Output some info for debugging
00092     if (PrtLvl > PRINT_NONE) printf("\nCalling Safe BiCGstab solver (CSR) ...\n");
00093
00094     #if DEBUG_MODE > 0
00095         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00096         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00097     #endif
00098
00099     // r = b-A*u
00100     fasp_darray_cp(m, bval, r);
00101     fasp_blas_dcsr_aAxy(-1.0, A, uval, r);

```

```

00102     absres0 = fasp_blas_darray_norm2(m,r);
00103
00104     // compute initial relative residual
00105     switch (StopType) {
00106     case STOP_REL_RES:
00107         normr0 = MAX(SMALLREAL,absres0);
00108         relres = absres0/normr0;
00109         break;
00110     case STOP_REL_PRECRES:
00111         normr0 = MAX(SMALLREAL,absres0);
00112         relres = absres0/normr0;
00113         break;
00114     case STOP_MOD_REL_RES:
00115         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
00116         relres = absres0/normu;
00117         break;
00118     default:
00119         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00120         goto FINISHED;
00121     }
00122
00123     // if initial residual is small, no need to iterate!
00124     if (relres<tol) goto FINISHED;
00125
00126     // output iteration information if needed
00127     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00128
00129     // rho = r* := r
00130     fasp_darray_cp(m,r,rho);
00131     temp1 = fasp_blas_darray_dotprod(m,r,rho);
00132
00133     // p = r
00134     fasp_darray_cp(m,r,p);
00135
00136     // main BiCGstab loop
00137     while ( iter++ < MaxIt ) {
00138
00139         // pp = precondition(p)
00140         if ( pc != NULL )
00141             pc->fct(p,pp,pc->data); /* Apply preconditioner */
00142         else
00143             fasp_darray_cp(m,p,pp); /* No preconditioner */
00144
00145         // z = A*pp
00146         fasp_blas_dcsr_mxv(A,pp,z);
00147
00148         // alpha = (r,rho)/(A*p,rho)
00149         temp2 = fasp_blas_darray_dotprod(m,z,rho);
00150         if ( ABS(temp2) > SMALLREAL ) {
00151             alpha = temp1/temp2;
00152         }
00153         else {
00154             ITS_DIVZERO; goto FINISHED;
00155         }
00156
00157         // s = r - alpha z
00158         fasp_darray_cp(m,r,s);
00159         fasp_blas_darray_axpy(m,-alpha,z,s);
00160
00161         // sp = precondition(s)
00162         if ( pc != NULL )
00163             pc->fct(s,sp,pc->data); /* Apply preconditioner */
00164         else
00165             fasp_darray_cp(m,s,sp); /* No preconditioner */
00166
00167         // t = A*sp;
00168         fasp_blas_dcsr_mxv(A,sp,t);
00169
00170         // omega = (t,s)/(t,t)
00171         tempr = fasp_blas_darray_dotprod(m,t,t);
00172
00173         if ( ABS(tempr) > SMALLREAL ) {
00174             omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
00175         }
00176         else {
00177             omega = 0.0;
00178             if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
00179         }
00180
00181         // delu = alpha pp + omega sp
00182         fasp_blas_darray_axpby(m,alpha,pp,omega,sp);

```

```

00183
00184 // u = u + delu
00185 fasp_blas_darray_axpy(m,1.0,sp,uval);
00186
00187 // r = s - omega t
00188 fasp_blas_darray_axpy(m,-omega,t,s);
00189 fasp_darray_cp(m,s,r);
00190
00191 // beta = (r,rho)/(rp,rho)
00192 temp2 = temp1;
00193 temp1 = fasp_blas_darray_dotprod(m,r,rho);
00194
00195 if ( ABS(temp2) > SMALLREAL ) {
00196     beta = (temp1*alpha)/(temp2*omega);
00197 }
00198 else {
00199     ITS_DIVZERO; goto RESTORE_BESTSOL;
00200 }
00201
00202 // p = p - omega z
00203 fasp_blas_darray_axpy(m,-omega,z,p);
00204
00205 // p = r + beta p
00206 fasp_blas_darray_axpy(m,1.0,r,beta,p);
00207
00208 // compute difference
00209 normd = fasp_blas_darray_norm2(m,sp);
00210 normu = fasp_blas_darray_norm2(m,uval);
00211 reldiff = normd/normu;
00212
00213 if ( normd < TOL_s ) {
00214     ITS_SMALLSP; goto FINISHED;
00215 }
00216
00217 // compute residuals
00218 switch (StopType) {
00219     case STOP_REL_RES:
00220         absres = fasp_blas_darray_norm2(m,r);
00221         relres = absres/normr0;
00222         break;
00223     case STOP_REL_PRECRES:
00224         if ( pc == NULL )
00225             fasp_darray_cp(m,r,z);
00226         else
00227             pc->fct(r,z,pc->data);
00228         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
00229         relres = absres/normr0;
00230         break;
00231     case STOP_MOD_REL_RES:
00232         absres = fasp_blas_darray_norm2(m,r);
00233         relres = absres/normu;
00234         break;
00235 }
00236
00237 // safety net check: save the best-so-far solution
00238 if ( fasp_dvec_isnan(u) ) {
00239     // If the solution is NAN, restore the best solution
00240     absres = BIGREAL;
00241     goto RESTORE_BESTSOL;
00242 }
00243
00244 if ( absres < absres_best - maxdiff ) {
00245     absres_best = absres;
00246     iter_best = iter;
00247     fasp_darray_cp(m,uval,u_best);
00248 }
00249
00250 // compute reduction factor of residual ||r||
00251 factor = absres/absres0;
00252
00253 // output iteration information if needed
00254 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00255
00256 // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00257 normuinf = fasp_blas_darray_norminf(m,uval);
00258 if ( normuinf <= sol_inf_tol ) {
00259     if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00260     iter = ERROR_SOLVER_SOLSTAG;
00261     goto FINISHED;
00262 }
00263

```

```

00264         // Check II: if staggenated, try to restart
00265         if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {
00266
00267             if ( PrtLvl >= PRINT_MORE ) {
00268                 ITS_DIFFRES(reldiff, relres);
00269                 ITS_RESTART;
00270             }
00271
00272             // re-init iteration param
00273             fasp_darray_cp(m, bval, r);
00274             fasp_blas_dcsr_aAxy(-1.0, A, uval, r);
00275
00276             // pp = precondition(p)
00277             fasp_darray_cp(m, r, p);
00278             if ( pc != NULL )
00279                 pc->fct(p, pp, pc->data); /* Apply preconditioner */
00280             else
00281                 fasp_darray_cp(m, p, pp); /* No preconditioner */
00282
00283             // rho = r* := r
00284             fasp_darray_cp(m, r, rho);
00285             templ = fasp_blas_darray_dotprod(m, r, rho);
00286
00287             // compute residuals
00288             switch (StopType) {
00289                 case STOP_REL_RES:
00290                     absres = fasp_blas_darray_norm2(m, r);
00291                     relres = absres/normr0;
00292                     break;
00293                 case STOP_REL_PRECRES:
00294                     if ( pc != NULL )
00295                         pc->fct(r, z, pc->data);
00296                     else
00297                         fasp_darray_cp(m, r, z);
00298                     absres = sqrt(ABS(fasp_blas_darray_dotprod(m, r, z)));
00299                     relres = absres/normr0;
00300                     break;
00301                 case STOP_MOD_REL_RES:
00302                     absres = fasp_blas_darray_norm2(m, r);
00303                     relres = absres/normu;
00304                     break;
00305             }
00306
00307             if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00308
00309             if ( relres < tol )
00310                 break;
00311             else {
00312                 if ( stag >= MaxStag ) {
00313                     if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00314                     iter = ERROR_SOLVER_STAG;
00315                     goto FINISHED;
00316                 }
00317                 ++stag;
00318                 ++restart_step;
00319             }
00320
00321         } // end of stagnation check!
00322
00323         // Check III: prevent false convergence
00324         if ( relres < tol ) {
00325             if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00326
00327             // re-init iteration param
00328             fasp_darray_cp(m, bval, r);
00329             fasp_blas_dcsr_aAxy(-1.0, A, uval, r);
00330
00331             // pp = precondition(p)
00332             fasp_darray_cp(m, r, p);
00333             if ( pc != NULL )
00334                 pc->fct(p, pp, pc->data); /* Apply preconditioner */
00335             else
00336                 fasp_darray_cp(m, p, pp); /* No preconditioner */
00337
00338             // rho = r* := r
00339             fasp_darray_cp(m, r, rho);
00340             templ = fasp_blas_darray_dotprod(m, r, rho);
00341
00342             // compute residuals
00343             switch (StopType) {
00344                 case STOP_REL_RES:

```

```

00345         absres = fasp_blas_darray_norm2(m,r);
00346         relres = absres/normr0;
00347         break;
00348     case STOP_REL_PRECRES:
00349         if ( pc != NULL )
00350             pc->fct(r,z,pc->data);
00351         else
00352             fasp_darray_cp(m,r,z);
00353         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
00354         relres = tempr/normr0;
00355         break;
00356     case STOP_MOD_REL_RES:
00357         absres = fasp_blas_darray_norm2(m,r);
00358         relres = absres/normu;
00359         break;
00360     }
00361
00362     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00363
00364     // check convergence
00365     if ( relres < tol ) break;
00366
00367     if ( more_step >= MaxRestartStep ) {
00368         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00369         iter = ERROR_SOLVER_TOLSMALL;
00370         goto FINISHED;
00371     }
00372     else {
00373         if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
00374     }
00375
00376     ++more_step;
00377     ++restart_step;
00378 } // end if safe guard
00379
00380 absres0 = absres;
00381
00382 } // end of main BiCGstab loop
00383
00384 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00385 if ( iter != iter_best ) {
00386
00387     // compute best residual
00388     fasp_darray_cp(m,b->val,r);
00389     fasp_blas_dcsr_aAxy(-1.0,A,u_best,r);
00390
00391     switch ( StopType ) {
00392     case STOP_REL_RES:
00393         absres_best = fasp_blas_darray_norm2(m,r);
00394         break;
00395     case STOP_REL_PRECRES:
00396         // z = B(r)
00397         if ( pc != NULL )
00398             pc->fct(r,z,pc->data); /* Apply preconditioner */
00399         else
00400             fasp_darray_cp(m,r,z); /* No preconditioner */
00401         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00402         break;
00403     case STOP_MOD_REL_RES:
00404         absres_best = fasp_blas_darray_norm2(m,r);
00405         break;
00406     }
00407
00408     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00409         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00410         fasp_darray_cp(m,u_best,u->val);
00411         relres = absres_best / normr0;
00412     }
00413 }
00414
00415 FINISHED: // finish the iterative method
00416 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00417
00418 // clean up temp memory
00419 fasp_mem_free(work); work = NULL;
00420
00421 #if DEBUG_MODE > 0
00422     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00423 #endif
00424
00425 if ( iter > MaxIt )

```



```

00426         return ERROR_SOLVER_MAXIT;
00427     else
00428         return iter;
00429 }
00430
00452 INT fasp_solver_dbsr_spbcgs (const dBSRmat *A,
00453                             const dvector *b,
00454                             dvector *u,
00455                             precondition *pc,
00456                             const REAL tol,
00457                             const INT MaxIt,
00458                             const SHORT StopType,
00459                             const SHORT PrtLvl)
00460 {
00461     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00462     const INT m = b->row;
00463     const REAL maxdiff = tol*STAG_RATIO; // stagantation tolerance
00464     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00465     const REAL TOL_s = tol*1e-2; // tolerance for norm(p)
00466
00467     // local variables
00468     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00469     REAL alpha, beta, omega, temp1, temp2;
00470     REAL absres0 = BIGREAL, absres = BIGREAL;
00471     REAL relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00472     REAL reldiff, factor, normd, tempr, normuin;
00473     REAL *uval = u->val, *bval = b->val;
00474     INT iter_best = 0; // initial best known iteration
00475     REAL absres_best = BIGREAL; // initial best known residual
00476
00477     // allocate temp memory (need 8*m REAL)
00478     REAL *work = (REAL *)fasp_mem_malloc(9*m, sizeof(REAL));
00479     REAL *p = work, *z = work + m, *r = z + m, *t = r + m;
00480     REAL *rho = t + m, *pp = rho + m, *s = pp + m, *sp = s + m, *u_best = sp + m;
00481
00482     // Output some info for debugging
00483     if (PrtLvl > PRINT_NONE) printf("\nCalling Safe BiCGstab solver (BSR) ...\n");
00484
00485     #if DEBUG_MODE > 0
00486         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00487         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00488     #endif
00489
00490     // r = b-A*u
00491     fasp_darray_cp(m, bval, r);
00492     fasp_blas_dbsr_aAxy(-1.0, A, uval, r);
00493     absres0 = fasp_blas_darray_norm2(m, r);
00494
00495     // compute initial relative residual
00496     switch (StopType) {
00497     case STOP_REL_RES:
00498         normr0 = MAX(SMALLREAL, absres0);
00499         relres = absres0/normr0;
00500         break;
00501     case STOP_REL_PRECRES:
00502         normr0 = MAX(SMALLREAL, absres0);
00503         relres = absres0/normr0;
00504         break;
00505     case STOP_MOD_REL_RES:
00506         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(m, uval));
00507         relres = absres0/normu;
00508         break;
00509     default:
00510         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00511         goto FINISHED;
00512     }
00513
00514     // if initial residual is small, no need to iterate!
00515     if (relres < tol) goto FINISHED;
00516
00517     // output iteration information if needed
00518     fasp_itinfo(PrtLvl, StopType, iter, relres, absres0, 0.0);
00519
00520     // rho = r* := r
00521     fasp_darray_cp(m, r, rho);
00522     temp1 = fasp_blas_darray_dotprod(m, r, rho);
00523
00524     // p = r
00525     fasp_darray_cp(m, r, p);
00526
00527     // main BiCGstab loop

```

```

00528     while ( iter++ < MaxIt ) {
00529
00530         // pp = precondition(p)
00531         if ( pc != NULL )
00532             pc->fct(p,pp,pc->data); /* Apply preconditioner */
00533         else
00534             fasp_darray_cp(m,p,pp); /* No preconditioner */
00535
00536         // z = A*pp
00537         fasp_blas_dbsr_mnv(A,pp,z);
00538
00539         // alpha = (r,rho)/(A*p,rho)
00540         temp2 = fasp_blas_darray_dotprod(m,z,rho);
00541         if ( ABS(temp2) > SMALLREAL ) {
00542             alpha = temp1/temp2;
00543         }
00544         else {
00545             ITS_DIVZERO; goto FINISHED;
00546         }
00547
00548         // s = r - alpha z
00549         fasp_darray_cp(m,r,s);
00550         fasp_blas_darray_axpy(m,-alpha,z,s);
00551
00552         // sp = precondition(s)
00553         if ( pc != NULL )
00554             pc->fct(s,sp,pc->data); /* Apply preconditioner */
00555         else
00556             fasp_darray_cp(m,s,sp); /* No preconditioner */
00557
00558         // t = A*sp;
00559         fasp_blas_dbsr_mnv(A,sp,t);
00560
00561         // omega = (t,s)/(t,t)
00562         tempr = fasp_blas_darray_dotprod(m,t,t);
00563
00564         if ( ABS(tempr) > SMALLREAL ) {
00565             omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
00566         }
00567         else {
00568             omega = 0.0;
00569             if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
00570         }
00571
00572         // delu = alpha pp + omega sp
00573         fasp_blas_darray_axpby(m,alpha,pp,omega,sp);
00574
00575         // u = u + delu
00576         fasp_blas_darray_axpy(m,1.0,sp,uval);
00577
00578         // r = s - omega t
00579         fasp_blas_darray_axpy(m,-omega,t,s);
00580         fasp_darray_cp(m,s,r);
00581
00582         // beta = (r,rho)/(rp,rho)
00583         temp2 = temp1;
00584         temp1 = fasp_blas_darray_dotprod(m,r,rho);
00585
00586         if ( ABS(temp2) > SMALLREAL ) {
00587             beta = (temp1*alpha)/(temp2*omega);
00588         }
00589         else {
00590             ITS_DIVZERO; goto RESTORE_BESTSOL;
00591         }
00592
00593         // p = p - omega z
00594         fasp_blas_darray_axpy(m,-omega,z,p);
00595
00596         // p = r + beta p
00597         fasp_blas_darray_axpby(m,1.0,r,beta,p);
00598
00599         // compute difference
00600         normd = fasp_blas_darray_norm2(m,sp);
00601         normu = fasp_blas_darray_norm2(m,uval);
00602         relldiff = normd/normu;
00603
00604         if ( normd < TOL_s ) {
00605             ITS_SMALLSP; goto FINISHED;
00606         }
00607
00608         // compute residuals

```

```

00609         switch (StopType) {
00610             case STOP_REL_RES:
00611                 absres = fasp_blas_darray_norm2(m,r);
00612                 relres = absres/normr0;
00613                 break;
00614             case STOP_REL_PRECRES:
00615                 if ( pc == NULL )
00616                     fasp_darray_cp(m,r,z);
00617                 else
00618                     pc->fct(r,z,pc->data);
00619                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
00620                 relres = absres/normr0;
00621                 break;
00622             case STOP_MOD_REL_RES:
00623                 absres = fasp_blas_darray_norm2(m,r);
00624                 relres = absres/normu;
00625                 break;
00626         }
00627
00628         // safety net check: save the best-so-far solution
00629         if ( fasp_dvec_isnan(u) ) {
00630             // If the solution is NAN, restore the best solution
00631             absres = BIGREAL;
00632             goto RESTORE_BESTSOL;
00633         }
00634
00635         if ( absres < absres_best - maxdiff ) {
00636             absres_best = absres;
00637             iter_best = iter;
00638             fasp_darray_cp(m,uval,u_best);
00639         }
00640
00641         // compute reduction factor of residual ||r||
00642         factor = absres/absres0;
00643
00644         // output iteration information if needed
00645         fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00646
00647         // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00648         normuinf = fasp_blas_darray_norminf(m, uval);
00649         if ( normuinf <= sol_inf_tol ) {
00650             if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00651             iter = ERROR_SOLVER_SOLSTAG;
00652             goto FINISHED;
00653         }
00654
00655         // Check II: if staggenated, try to restart
00656         if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {
00657
00658             if ( PrtLvl >= PRINT_MORE ) {
00659                 ITS_DIFFRES(reldiff,relres);
00660                 ITS_RESTART;
00661             }
00662
00663             // re-init iteration param
00664             fasp_darray_cp(m,bval,r);
00665             fasp_blas_dbsr_aApy(-1.0,A,uval,r);
00666
00667             // pp = precondition(p)
00668             fasp_darray_cp(m,r,p);
00669             if ( pc != NULL )
00670                 pc->fct(p,pp,pc->data); /* Apply preconditioner */
00671             else
00672                 fasp_darray_cp(m,p,pp); /* No preconditioner */
00673
00674             // rho = r* := r
00675             fasp_darray_cp(m,r,rho);
00676             temp1 = fasp_blas_darray_dotprod(m,r,rho);
00677
00678             // compute residuals
00679             switch (StopType) {
00680                 case STOP_REL_RES:
00681                     absres = fasp_blas_darray_norm2(m,r);
00682                     relres = absres/normr0;
00683                     break;
00684                 case STOP_REL_PRECRES:
00685                     if ( pc != NULL )
00686                         pc->fct(r,z,pc->data);
00687                     else
00688                         fasp_darray_cp(m,r,z);
00689                     absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));

```

```

00690         relres = absres/normr0;
00691         break;
00692     case STOP_MOD_REL_RES:
00693         absres = fasp_blas_darray_norm2(m,r);
00694         relres = absres/normu;
00695         break;
00696     }
00697
00698     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00699
00700     if ( relres < tol )
00701         break;
00702     else {
00703         if ( stag >= MaxStag ) {
00704             if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00705             iter = ERROR_SOLVER_STAG;
00706             goto FINISHED;
00707         }
00708         ++stag;
00709         ++restart_step;
00710     }
00711 } // end of stagnation check!
00712
00713 // Check III: prevent false convergence
00714 if ( relres < tol ) {
00715     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00716
00717     // re-init iteration param
00718     fasp_darray_cp(m,bval,r);
00719     fasp_blas_dbsr_aAxy(-1.0,A,uval,r);
00720
00721     // pp = precondition(p)
00722     fasp_darray_cp(m,r,p);
00723     if ( pc != NULL )
00724         pc->fct(p,pp,pc->data); /* Apply preconditioner */
00725     else
00726         fasp_darray_cp(m,p,pp); /* No preconditioner */
00727
00728     // rho = r* := r
00729     fasp_darray_cp(m,r,rho);
00730     temp1 = fasp_blas_darray_dotprod(m,r,rho);
00731
00732     // compute residuals
00733     switch (StopType) {
00734     case STOP_REL_RES:
00735         absres = fasp_blas_darray_norm2(m,r);
00736         relres = absres/normr0;
00737         break;
00738     case STOP_REL_PRECRES:
00739         if ( pc != NULL )
00740             pc->fct(r,z,pc->data);
00741         else
00742             fasp_darray_cp(m,r,z);
00743         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
00744         relres = tempr/normr0;
00745         break;
00746     case STOP_MOD_REL_RES:
00747         absres = fasp_blas_darray_norm2(m,r);
00748         relres = absres/normu;
00749         break;
00750     }
00751 }
00752
00753 if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00754
00755 // check convergence
00756 if ( relres < tol ) break;
00757
00758 if ( more_step >= MaxRestartStep ) {
00759     if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00760     iter = ERROR_SOLVER_TOLSMALL;
00761     goto FINISHED;
00762 }
00763 else {
00764     if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
00765 }
00766
00767 ++more_step;
00768 ++restart_step;
00769 } // end if safe guard
00770

```

```

00771         absres0 = absres;
00772     } // end of main BiCGstab loop
00773
00774 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00775     if ( iter != iter_best ) {
00776         // compute best residual
00777         fasp_darray_cp(m,b->val,r);
00778         fasp_blas_dbsr_aAxy(-1.0,A,u_best,r);
00781
00782         switch ( StopType ) {
00783             case STOP_REL_RES:
00784                 absres_best = fasp_blas_darray_norm2(m,r);
00785                 break;
00786             case STOP_REL_PRECRES:
00787                 // z = B(r)
00788                 if ( pc != NULL )
00789                     pc->fct(r,z,pc->data); /* Apply preconditioner */
00790                 else
00791                     fasp_darray_cp(m,r,z); /* No preconditioner */
00792                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00793                 break;
00794             case STOP_MOD_REL_RES:
00795                 absres_best = fasp_blas_darray_norm2(m,r);
00796                 break;
00797         }
00798
00799         if ( absres > absres_best + maxdiff || isnan(absres) ) {
00800             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00801             fasp_darray_cp(m,u_best,u->val);
00802             relres = absres_best / normr0;
00803         }
00804     }
00805
00806 FINISHED: // finish the iterative method
00807     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00808
00809     // clean up temp memory
00810     fasp_mem_free(work); work = NULL;
00811
00812 #if DEBUG_MODE > 0
00813     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00814 #endif
00815
00816     if ( iter > MaxIt )
00817         return ERROR_SOLVER_MAXIT;
00818     else
00819         return iter;
00820 }
00821
00843 INT fasp_solver_dblc_spbcgs (const dBLMat *A,
00844                             const dvector *b,
00845                             dvector *u,
00846                             precondition *pc,
00847                             const REAL tol,
00848                             const INT MaxIt,
00849                             const SHORT StopType,
00850                             const SHORT PrtLvl)
00851 {
00852     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00853     const INT m = b->row;
00854     const REAL maxdiff = tol*STAG_RATIO; // stagantion tolerance
00855     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00856     const REAL TOL_s = tol*1e-2; // tolerance for norm(p)
00857
00858     // local variables
00859     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00860     REAL alpha, beta, omega, templ, temp2;
00861     REAL absres0 = BIGREAL, absres = BIGREAL;
00862     REAL relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00863     REAL reldiff, factor, normd, tempr, normuinf;
00864     REAL *uval = u->val, *bval = b->val;
00865     INT iter_best = 0; // initial best known iteration
00866     REAL absres_best = BIGREAL; // initial best known residual
00867
00868     // allocate temp memory (need 8*m REAL)
00869     REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
00870     REAL *p = work, *z = work + m, *r = z + m, *t = r + m;
00871     REAL *rho = t + m, *pp = rho + m, *s = pp + m, *sp = s + m, *u_best = sp + m;
00872

```

```

00873 // Output some info for debugging
00874 if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (BLC) ...\n");
00875
00876 #if DEBUG_MODE > 0
00877     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00878     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00879 #endif
00880
00881 // r = b-A*u
00882 fasp_darray_cp(m,bval,r);
00883 fasp_blas_dblc_aApy(-1.0,A,uval,r);
00884 absres0 = fasp_blas_darray_norm2(m,r);
00885
00886 // compute initial relative residual
00887 switch (StopType) {
00888     case STOP_REL_RES:
00889         normr0 = MAX(SMALLREAL,absres0);
00890         relres = absres0/normr0;
00891         break;
00892     case STOP_REL_PRECRES:
00893         normr0 = MAX(SMALLREAL,absres0);
00894         relres = absres0/normr0;
00895         break;
00896     case STOP_MOD_REL_RES:
00897         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
00898         relres = absres0/normu;
00899         break;
00900     default:
00901         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00902         goto FINISHED;
00903 }
00904
00905 // if initial residual is small, no need to iterate!
00906 if (relres<tol) goto FINISHED;
00907
00908 // output iteration information if needed
00909 fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00910
00911 // rho = r* := r
00912 fasp_darray_cp(m,r,rho);
00913 temp1 = fasp_blas_darray_dotprod(m,r,rho);
00914
00915 // p = r
00916 fasp_darray_cp(m,r,p);
00917
00918 // main BiCGstab loop
00919 while ( iter++ < MaxIt ) {
00920
00921     // pp = precondition(p)
00922     if ( pc != NULL )
00923         pc->fct(p,pp,pc->data); /* Apply preconditioner */
00924     else
00925         fasp_darray_cp(m,p,pp); /* No preconditioner */
00926
00927     // z = A*pp
00928     fasp_blas_dblc_mxv(A,pp,z);
00929
00930     // alpha = (r,rho)/(A*p,rho)
00931     temp2 = fasp_blas_darray_dotprod(m,z,rho);
00932     if ( ABS(temp2) > SMALLREAL ) {
00933         alpha = temp1/temp2;
00934     }
00935     else {
00936         ITS_DIVZERO; goto FINISHED;
00937     }
00938
00939     // s = r - alpha z
00940     fasp_darray_cp(m,r,s);
00941     fasp_blas_darray_axpy(m,-alpha,z,s);
00942
00943     // sp = precondition(s)
00944     if ( pc != NULL )
00945         pc->fct(s,sp,pc->data); /* Apply preconditioner */
00946     else
00947         fasp_darray_cp(m,s,sp); /* No preconditioner */
00948
00949     // t = A*sp;
00950     fasp_blas_dblc_mxv(A,sp,t);
00951
00952     // omega = (t,s)/(t,t)
00953     tempr = fasp_blas_darray_dotprod(m,t,t);

```

```

00954
00955     if ( ABS(tempr) > SMALLREAL ) {
00956         omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
00957     }
00958     else {
00959         omega = 0.0;
00960         if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
00961     }
00962
00963     // delu = alpha pp + omega sp
00964     fasp_blas_darray_axpy(m,alpha,pp,omega,sp);
00965
00966     // u = u + delu
00967     fasp_blas_darray_axpy(m,1.0,sp,uval);
00968
00969     // r = s - omega t
00970     fasp_blas_darray_axpy(m,-omega,t,s);
00971     fasp_darray_cp(m,s,r);
00972
00973     // beta = (r,rho)/(rp,rho)
00974     temp2 = temp1;
00975     temp1 = fasp_blas_darray_dotprod(m,r,rho);
00976
00977     if ( ABS(temp2) > SMALLREAL ) {
00978         beta = (temp1*alpha)/(temp2*omega);
00979     }
00980     else {
00981         ITS_DIVZERO; goto RESTORE_BESTSOL;
00982     }
00983
00984     // p = p - omega z
00985     fasp_blas_darray_axpy(m,-omega,z,p);
00986
00987     // p = r + beta p
00988     fasp_blas_darray_axpy(m,1.0,r,beta,p);
00989
00990     // compute difference
00991     normd = fasp_blas_darray_norm2(m,sp);
00992     normu = fasp_blas_darray_norm2(m,uval);
00993     reldiff = normd/normu;
00994
00995     if ( normd < TOL_s ) {
00996         ITS_SMALLSP; goto FINISHED;
00997     }
00998
00999     // compute residuals
01000     switch (StopType) {
01001     case STOP_REL_RES:
01002         absres = fasp_blas_darray_norm2(m,r);
01003         relres = absres/normr0;
01004         break;
01005     case STOP_REL_PRECRES:
01006         if ( pc == NULL )
01007             fasp_darray_cp(m,r,z);
01008         else
01009             pc->fct(r,z,pc->data);
01010         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
01011         relres = absres/normr0;
01012         break;
01013     case STOP_MOD_REL_RES:
01014         absres = fasp_blas_darray_norm2(m,r);
01015         relres = absres/normu;
01016         break;
01017     }
01018
01019     // safety net check: save the best-so-far solution
01020     if ( fasp_dvec_isnan(u) ) {
01021         // If the solution is NAN, restore the best solution
01022         absres = BIGREAL;
01023         goto RESTORE_BESTSOL;
01024     }
01025
01026     if ( absres < absres_best - maxdiff ) {
01027         absres_best = absres;
01028         iter_best = iter;
01029         fasp_darray_cp(m,uval,u_best);
01030     }
01031
01032     // compute reduction factor of residual ||r||
01033     factor = absres/absres0;
01034

```

```

01035     // output iteration information if needed
01036     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01037
01038     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
01039     normuinf = fasp_blas_darray_norminf(m, uval);
01040     if ( normuinf <= sol_inf_tol ) {
01041         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01042         iter = ERROR_SOLVER_SOLSTAG;
01043         goto FINISHED;
01044     }
01045
01046     // Check II: if staggenated, try to restart
01047     if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {
01048
01049         if ( PrtLvl >= PRINT_MORE ) {
01050             ITS_DIFFRES(reldiff,relres);
01051             ITS_RESTART;
01052         }
01053
01054         // re-init iteration param
01055         fasp_darray_cp(m,bval,r);
01056         fasp_blas_dbic_aAxy(-1.0,A,uval,r);
01057
01058         // pp = precondition(p)
01059         fasp_darray_cp(m,r,p);
01060         if ( pc != NULL )
01061             pc->fct(p,pp,pc->data); /* Apply preconditioner */
01062         else
01063             fasp_darray_cp(m,p,pp); /* No preconditioner */
01064
01065         // rho = r* := r
01066         fasp_darray_cp(m,r,rho);
01067         temp1 = fasp_blas_darray_dotprod(m,r,rho);
01068
01069         // compute residuals
01070         switch (StopType) {
01071             case STOP_REL_RES:
01072                 absres = fasp_blas_darray_norm2(m,r);
01073                 relres = absres/normr0;
01074                 break;
01075             case STOP_REL_PRECRES:
01076                 if ( pc != NULL )
01077                     pc->fct(r,z,pc->data);
01078                 else
01079                     fasp_darray_cp(m,r,z);
01080                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
01081                 relres = absres/normr0;
01082                 break;
01083             case STOP_MOD_REL_RES:
01084                 absres = fasp_blas_darray_norm2(m,r);
01085                 relres = absres/normu;
01086                 break;
01087         }
01088
01089         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01090
01091         if ( relres < tol )
01092             break;
01093         else {
01094             if ( stag >= MaxStag ) {
01095                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01096                 iter = ERROR_SOLVER_STAG;
01097                 goto FINISHED;
01098             }
01099             ++stag;
01100             ++restart_step;
01101         }
01102     } // end of stagnation check!
01103
01104     // Check III: prevent false convergence
01105     if ( relres < tol ) {
01106         if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01107
01108         // re-init iteration param
01109         fasp_darray_cp(m,bval,r);
01110         fasp_blas_dbic_aAxy(-1.0,A,uval,r);
01111
01112         // pp = precondition(p)
01113         fasp_darray_cp(m,r,p);
01114         if ( pc != NULL )
01115

```



```

01116         pc->fct(p,pp,pc->data); /* Apply preconditioner */
01117     else
01118         fasp_darray_cp(m,p,pp); /* No preconditioner */
01119
01120     // rho = r* := r
01121     fasp_darray_cp(m,r,rho);
01122     temp1 = fasp_blas_darray_dotprod(m,r,rho);
01123
01124     // compute residuals
01125     switch (StopType) {
01126     case STOP_REL_RES:
01127         absres = fasp_blas_darray_norm2(m,r);
01128         relres = absres/normr0;
01129         break;
01130     case STOP_REL_PRECRES:
01131         if ( pc != NULL )
01132             pc->fct(r,z,pc->data);
01133         else
01134             fasp_darray_cp(m,r,z);
01135         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
01136         relres = tempr/normr0;
01137         break;
01138     case STOP_MOD_REL_RES:
01139         absres = fasp_blas_darray_norm2(m,r);
01140         relres = absres/normu;
01141         break;
01142     }
01143
01144     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01145
01146     // check convergence
01147     if ( relres < tol ) break;
01148
01149     if ( more_step >= MaxRestartStep ) {
01150         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01151         iter = ERROR_SOLVER_TOLSMALL;
01152         goto FINISHED;
01153     }
01154     else {
01155         if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
01156     }
01157
01158     ++more_step;
01159     ++restart_step;
01160 } // end if safe guard
01161
01162     absres0 = absres;
01163
01164 } // end of main BiCGstab loop
01165
01166 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01167     if ( iter != iter_best ) {
01168         // compute best residual
01169         fasp_darray_cp(m,b->val,r);
01170         fasp_blas_dblc_aAxy(-1.0,A,u_best,r);
01171
01172         switch ( StopType ) {
01173         case STOP_REL_RES:
01174             absres_best = fasp_blas_darray_norm2(m,r);
01175             break;
01176         case STOP_REL_PRECRES:
01177             // z = B(r)
01178             if ( pc != NULL )
01179                 pc->fct(r,z,pc->data); /* Apply preconditioner */
01180             else
01181                 fasp_darray_cp(m,r,z); /* No preconditioner */
01182             absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01183             break;
01184         case STOP_MOD_REL_RES:
01185             absres_best = fasp_blas_darray_norm2(m,r);
01186             break;
01187         }
01188
01189         if ( absres > absres_best + maxdiff || isnan(absres) ) {
01190             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01191             fasp_darray_cp(m,u_best,u->val);
01192             relres = absres_best / normr0;
01193         }
01194     }
01195 }
01196

```

```

01197 FINISHED: // finish the iterative method
01198     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01199
01200     // clean up temp memory
01201     fasp_mem_free(work); work = NULL;
01202
01203 #if DEBUG_MODE > 0
01204     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01205 #endif
01206
01207     if ( iter > MaxIt )
01208         return ERROR_SOLVER_MAXIT;
01209     else
01210         return iter;
01211 }
01212
01234 INT fasp_solver_dstr_spgcgs (const dSTRmat *A,
01235                             const dvector *b,
01236                             dvector *u,
01237                             precondition *pc,
01238                             const REAL tol,
01239                             const INT MaxIt,
01240                             const SHORT StopType,
01241                             const SHORT PrtLvl)
01242 {
01243     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
01244     const INT m = b->row;
01245     const REAL maxdiff = tol*STAG_RATIO; // stagnation tolerance
01246     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
01247     const REAL TOL_s = tol*1e-2; // tolerance for norm(p)
01248
01249     // local variables
01250     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
01251     REAL alpha, beta, omega, templ, temp2;
01252     REAL absres0 = BIGREAL, absres = BIGREAL;
01253     REAL relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
01254     REAL reldiff, factor, normd, tempr, normuin;
01255     REAL *uval = u->val, *bval = b->val;
01256     INT iter_best = 0; // initial best known iteration
01257     REAL absres_best = BIGREAL; // initial best known residual
01258
01259     // allocate temp memory (need 8*m REAL)
01260     REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
01261     REAL *p = work, *z = work + m, *r = z + m, *t = r + m;
01262     REAL *rho = t + m, *pp = rho + m, *s = pp + m, *sp = s + m, *u_best = sp + m;
01263
01264     // Output some info for debugging
01265     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (STR) ...\n");
01266
01267 #if DEBUG_MODE > 0
01268     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01269     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01270 #endif
01271
01272     // r = b-A*u
01273     fasp_darray_cp(m,bval,r);
01274     fasp_blas_dstr_aAxy(-1.0,A,uval,r);
01275     absres0 = fasp_blas_darray_norm2(m,r);
01276
01277     // compute initial relative residual
01278     switch (StopType) {
01279     case STOP_REL_RES:
01280         normr0 = MAX(SMALLREAL,absres0);
01281         relres = absres0/normr0;
01282         break;
01283     case STOP_REL_PRECRES:
01284         normr0 = MAX(SMALLREAL,absres0);
01285         relres = absres0/normr0;
01286         break;
01287     case STOP_MOD_REL_RES:
01288         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
01289         relres = absres0/normu;
01290         break;
01291     default:
01292         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01293         goto FINISHED;
01294     }
01295
01296     // if initial residual is small, no need to iterate!
01297     if (relres<tol) goto FINISHED;
01298

```

```

01299 // output iteration information if needed
01300 fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
01301
01302 // rho = r* := r
01303 fasp_darray_cp(m,r,rho);
01304 temp1 = fasp_blas_darray_dotprod(m,r,rho);
01305
01306 // p = r
01307 fasp_darray_cp(m,r,p);
01308
01309 // main BiCGstab loop
01310 while ( iter++ < MaxIt ) {
01311
01312     // pp = precondition(p)
01313     if ( pc != NULL )
01314         pc->fct(p,pp,pc->data); /* Apply preconditioner */
01315     else
01316         fasp_darray_cp(m,p,pp); /* No preconditioner */
01317
01318     // z = A*pp
01319     fasp_blas_dstr_mxdv(A,pp,z);
01320
01321     // alpha = (r,rho)/(A*p,rho)
01322     temp2 = fasp_blas_darray_dotprod(m,z,rho);
01323     if ( ABS(temp2) > SMALLREAL ) {
01324         alpha = temp1/temp2;
01325     }
01326     else {
01327         ITS_DIVZERO; goto FINISHED;
01328     }
01329
01330     // s = r - alpha z
01331     fasp_darray_cp(m,r,s);
01332     fasp_blas_darray_axpy(m,-alpha,z,s);
01333
01334     // sp = precondition(s)
01335     if ( pc != NULL )
01336         pc->fct(s,sp,pc->data); /* Apply preconditioner */
01337     else
01338         fasp_darray_cp(m,s,sp); /* No preconditioner */
01339
01340     // t = A*sp;
01341     fasp_blas_dstr_mxdv(A,sp,t);
01342
01343     // omega = (t,s)/(t,t)
01344     tempr = fasp_blas_darray_dotprod(m,t,t);
01345
01346     if ( ABS(tempr) > SMALLREAL ) {
01347         omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
01348     }
01349     else {
01350         omega = 0.0;
01351         if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
01352     }
01353
01354     // delu = alpha pp + omega sp
01355     fasp_blas_darray_axpy(m,alpha,pp,omega,sp);
01356
01357     // u = u + delu
01358     fasp_blas_darray_axpy(m,1.0,sp,uval);
01359
01360     // r = s - omega t
01361     fasp_blas_darray_axpy(m,-omega,t,s);
01362     fasp_darray_cp(m,s,r);
01363
01364     // beta = (r,rho)/(rp,rho)
01365     temp2 = temp1;
01366     temp1 = fasp_blas_darray_dotprod(m,r,rho);
01367
01368     if ( ABS(temp2) > SMALLREAL ) {
01369         beta = (temp1*alpha)/(temp2*omega);
01370     }
01371     else {
01372         ITS_DIVZERO; goto RESTORE_BESTSOL;
01373     }
01374
01375     // p = p - omega z
01376     fasp_blas_darray_axpy(m,-omega,z,p);
01377
01378     // p = r + beta p
01379     fasp_blas_darray_axpy(m,1.0,r,beta,p);

```

```

01380
01381 // compute difference
01382 normd = fasp_blas_darray_norm2(m, sp);
01383 normu = fasp_blas_darray_norm2(m, uval);
01384 reldiff = normd/normu;
01385
01386 if ( normd < TOL_s ) {
01387     ITS_SMALLSP; goto FINISHED;
01388 }
01389
01390 // compute residuals
01391 switch (StopType) {
01392     case STOP_REL_RES:
01393         absres = fasp_blas_darray_norm2(m, r);
01394         relres = absres/normr0;
01395         break;
01396     case STOP_REL_PRECRES:
01397         if ( pc == NULL )
01398             fasp_darray_cp(m, r, z);
01399         else
01400             pc->fct(r, z, pc->data);
01401         absres = sqrt(ABS(fasp_blas_darray_dotprod(m, r, z)));
01402         relres = absres/normr0;
01403         break;
01404     case STOP_MOD_REL_RES:
01405         absres = fasp_blas_darray_norm2(m, r);
01406         relres = absres/normu;
01407         break;
01408 }
01409
01410 // safety net check: save the best-so-far solution
01411 if ( fasp_dvec_isnan(u) ) {
01412     // If the solution is NAN, restore the best solution
01413     absres = BIGREAL;
01414     goto RESTORE_BESTSOL;
01415 }
01416
01417 if ( absres < absres_best - maxdiff ) {
01418     absres_best = absres;
01419     iter_best = iter;
01420     fasp_darray_cp(m, uval, u_best);
01421 }
01422
01423 // compute reduction factor of residual ||r||
01424 factor = absres/absres0;
01425
01426 // output iteration information if needed
01427 fasp_itinfo(PrtLvl, StopType, iter, relres, absres, factor);
01428
01429 // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
01430 normuinf = fasp_blas_darray_norminf(m, uval);
01431 if ( normuinf <= sol_inf_tol ) {
01432     if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01433     iter = ERROR_SOLVER_SOLSTAG;
01434     goto FINISHED;
01435 }
01436
01437 // Check II: if stagnated, try to restart
01438 if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {
01439
01440     if ( PrtLvl >= PRINT_MORE ) {
01441         ITS_DIFFRES(reldiff, relres);
01442         ITS_RESTART;
01443     }
01444
01445     // re-init iteration param
01446     fasp_darray_cp(m, bval, r);
01447     fasp_blas_dstr_aAxy(-1.0, A, uval, r);
01448
01449     // pp = precondition(p)
01450     fasp_darray_cp(m, r, p);
01451     if ( pc != NULL )
01452         pc->fct(p, pp, pc->data); /* Apply preconditioner */
01453     else
01454         fasp_darray_cp(m, p, pp); /* No preconditioner */
01455
01456     // rho = r* := r
01457     fasp_darray_cp(m, r, rho);
01458     temp1 = fasp_blas_darray_dotprod(m, r, rho);
01459
01460     // compute residuals

```

```

01461         switch (StopType) {
01462             case STOP_REL_RES:
01463                 absres = fasp_blas_darray_norm2(m,r);
01464                 relres = absres/normr0;
01465                 break;
01466             case STOP_REL_PRECRES:
01467                 if ( pc != NULL )
01468                     pc->fct(r,z,pc->data);
01469                 else
01470                     fasp_darray_cp(m,r,z);
01471                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
01472                 relres = absres/normr0;
01473                 break;
01474             case STOP_MOD_REL_RES:
01475                 absres = fasp_blas_darray_norm2(m,r);
01476                 relres = absres/normu;
01477                 break;
01478         }
01479         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01480
01481         if ( relres < tol )
01482             break;
01483         else {
01484             if ( stag >= MaxStag ) {
01485                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01486                 iter = ERROR_SOLVER_STAG;
01487                 goto FINISHED;
01488             }
01489             ++stag;
01490             ++restart_step;
01491         }
01492     }
01493 } // end of stagnation check!
01494
01495 // Check III: prevent false convergence
01496 if ( relres < tol ) {
01497     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01498
01499     // re-init iteration param
01500     fasp_darray_cp(m,bval,r);
01501     fasp_blas_dstr_aApy(-1.0,A,uval,r);
01502
01503     // pp = precondition(p)
01504     fasp_darray_cp(m,r,p);
01505     if ( pc != NULL )
01506         pc->fct(p,pp,pc->data); /* Apply preconditioner */
01507     else
01508         fasp_darray_cp(m,p,pp); /* No preconditioner */
01509
01510     // rho = r* := r
01511     fasp_darray_cp(m,r,rho);
01512     tempr = fasp_blas_darray_dotprod(m,r,rho);
01513
01514     // compute residuals
01515     switch (StopType) {
01516         case STOP_REL_RES:
01517             absres = fasp_blas_darray_norm2(m,r);
01518             relres = absres/normr0;
01519             break;
01520         case STOP_REL_PRECRES:
01521             if ( pc != NULL )
01522                 pc->fct(r,z,pc->data);
01523             else
01524                 fasp_darray_cp(m,r,z);
01525             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
01526             relres = tempr/normr0;
01527             break;
01528         case STOP_MOD_REL_RES:
01529             absres = fasp_blas_darray_norm2(m,r);
01530             relres = absres/normu;
01531             break;
01532     }
01533
01534     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01535
01536     // check convergence
01537     if ( relres < tol ) break;
01538
01539     if ( more_step >= MaxRestartStep ) {
01540         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01541     }

```

```

01542         iter = ERROR_SOLVER_TOLSMALL;
01543         goto FINISHED;
01544     }
01545     else {
01546         if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
01547     }
01548     ++more_step;
01549     ++restart_step;
01550 } // end if safe guard
01551
01552 absres0 = absres;
01553
01554 } // end of main BiCGstab loop
01555
01556 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01557 if ( iter != iter_best ) {
01558     // compute best residual
01559     fasp_darray_cp(m,b->val,r);
01560     fasp_blas_dstr_aAxy(-1.0,A,u_best,r);
01561
01562     switch ( StopType ) {
01563     case STOP_REL_RES:
01564         absres_best = fasp_blas_darray_norm2(m,r);
01565         break;
01566     case STOP_REL_PRECRES:
01567         // z = B(r)
01568         if ( pc != NULL )
01569             pc->fct(r,z,pc->data); /* Apply preconditioner */
01570         else
01571             fasp_darray_cp(m,r,z); /* No preconditioner */
01572         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01573         break;
01574     case STOP_MOD_REL_RES:
01575         absres_best = fasp_blas_darray_norm2(m,r);
01576         break;
01577     }
01578
01579     if ( absres > absres_best + maxdiff || isnan(absres) ) {
01580         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01581         fasp_darray_cp(m,u_best,u->val);
01582         relres = absres_best / normr0;
01583     }
01584 }
01585
01586 }
01587
01588 FINISHED: // finish the iterative method
01589 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01590
01591 // clean up temp memory
01592 fasp_mem_free(work); work = NULL;
01593
01594 #if DEBUG_MODE > 0
01595     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01596 #endif
01597
01598 if ( iter > MaxIt )
01599     return ERROR_SOLVER_MAXIT;
01600 else
01601     return iter;
01602 }
01603
01604 /*-----*/
01605 /*--          End of File          --*/
01606 /*-----*/

```

9.127 KrySPcg.c File Reference

Krylov subspace methods – Preconditioned CG with safety net.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dcsr_spcg](#) (const [dCSRmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned conjugate gradient method for solving $Au=b$ with safety net.
- [INT fasp_solver_dblc_spcg](#) (const [dBLCmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned conjugate gradient method for solving $Au=b$ with safety net.
- [INT fasp_solver_dstr_spcg](#) (const [dSTRmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned conjugate gradient method for solving $Au=b$ with safety net.

9.127.1 Detailed Description

Krylov subspace methods – Preconditioned CG with safety net.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvCSR.c](#), [BlaSpmvSTR.c](#), and [BlaVector.c](#)

The ‘best’ iterative solution will be saved and used upon exit; See [KryPcg.c](#) for a version without safety net

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KrySPcg.c](#).

9.127.2 Function Documentation

9.127.2.1 fasp_solver_dblc_spcg()

```
INT fasp_solver_dblc_spcg (
    const dBLCmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dBLCmat : the coefficient matrix
<i>b</i>	Pointer to dvector : the right hand side
<i>u</i>	Pointer to dvector : the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations

Parameters

<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/28/2013

Definition at line 393 of file [KrySPcg.c](#).

9.127.2.2 fasp_solver_dcsr_spcg()

```

INT fasp_solver_dcsr_spcg (
    const dCSRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/28/2013

Definition at line 60 of file [KrySPcg.c](#).**9.127.2.3 fasp_solver_dstr_spcg()**

```

INT fasp_solver_dstr_spcg (
    const dSTRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned conjugate gradient method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dSTRmat : the coefficient matrix
<i>b</i>	Pointer to dvector: the right hand side
<i>u</i>	Pointer to dvector: the unknowns
<i>MaxIt</i>	Maximal number of iterations
<i>tol</i>	Tolerance for stopping
<i>pc</i>	Pointer to the structure of precondition (precond)
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

03/28/2013

Definition at line 726 of file [KrySPcg.c](#).**9.128 KrySPcg.c**[Go to the documentation of this file.](#)

```

00001
00024 #include <math.h>
00025
00026 #include "fasp.h"
00027 #include "fasp_funcs.h"
00028
00029 /*-----*/
00030 /*--  Declare Private Functions  --*/
00031 /*-----*/

```

```

00032
00033 #include "KryUtil.inl"
00034
00035 /*-----*/
00036 /*--      Public Functions      --*/
00037 /*-----*/
00038
00060 INT fasp_solver_dcsr_spcg (const dCSRmat  *A,
00061                          const dvector  *b,
00062                          dvector        *u,
00063                          precondition    *pc,
00064                          const REAL      tol,
00065                          const INT       MaxIt,
00066                          const SHORT     StopType,
00067                          const SHORT     PrtLvl)
00068 {
00069     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00070     const INT    m = b->row;
00071     const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
00072     const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
00073
00074     // local variables
00075     INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
00076     REAL          absres0 = BIGREAL, absres = BIGREAL;
00077     REAL          relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00078     REAL          reldiff, factor, normuinf;
00079     REAL          alpha, beta, templ, temp2;
00080     INT           iter_best = 0; // initial best known iteration
00081     REAL          absres_best = BIGREAL; // initial best known residual
00082
00083     // allocate temp memory (need 5*m REAL numbers)
00084     REAL *work = (REAL *)fasp_mem_calloc(5*m, sizeof(REAL));
00085     REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;
00086
00087     // Output some info for debugging
00088     if (PrtLvl > PRINT_NONE) printf("\nCalling Safe CG solver (CSR) ...\n");
00089
00090     #if DEBUG_MODE > 0
00091     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00092     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00093     #endif
00094
00095     // r = b-A*u
00096     fasp_darray_cp(m, b->val, r);
00097     fasp_blas_dcsr_aAxy(-1.0, A, u->val, r);
00098
00099     if (pc != NULL)
00100         pc->fct(r, z, pc->data); /* Apply preconditioner */
00101     else
00102         fasp_darray_cp(m, r, z); /* No preconditioner */
00103
00104     // compute initial residuals
00105     switch (StopType) {
00106     case STOP_REL_RES:
00107         absres0 = fasp_blas_darray_norm2(m, r);
00108         normr0 = MAX(SMALLREAL, absres0);
00109         relres = absres0/normr0;
00110         break;
00111     case STOP_REL_PRECRES:
00112         absres0 = sqrt(fasp_blas_darray_dotprod(m, r, z));
00113         normr0 = MAX(SMALLREAL, absres0);
00114         relres = absres0/normr0;
00115         break;
00116     case STOP_MOD_REL_RES:
00117         absres0 = fasp_blas_darray_norm2(m, r);
00118         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(m, u->val));
00119         relres = absres0/normu;
00120         break;
00121     default:
00122         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00123         goto FINISHED;
00124     }
00125
00126     // if initial residual is small, no need to iterate!
00127     if (relres < tol) goto FINISHED;
00128
00129     // output iteration information if needed
00130     fasp_itinfo(PrtLvl, StopType, iter, relres, absres0, 0.0);
00131
00132     fasp_darray_cp(m, z, p);
00133     templ = fasp_blas_darray_dotprod(m, z, r);

```

```

00134
00135 // main PCG loop
00136 while ( iter++ < MaxIt ) {
00137
00138     // t=A*p
00139     fasp_blas_dcsr_mxv(A,p,t);
00140
00141     // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00142     temp2 = fasp_blas_darray_dotprod(m,t,p);
00143     if ( ABS(temp2) > SMALLREAL2 ) {
00144         alpha = temp1/temp2;
00145     }
00146     else { // Possible breakdown
00147         goto RESTORE_BESTSOL;
00148     }
00149
00150     // u_k=u_{k-1} + alpha_k*p_{k-1}
00151     fasp_blas_darray_axpy(m,alpha,p,u->val);
00152
00153     // r_k=r_{k-1} - alpha_k*A*p_{k-1}
00154     fasp_blas_darray_axpy(m,-alpha,t,r);
00155
00156     // compute residuals
00157     switch ( StopType ) {
00158         case STOP_REL_RES:
00159             absres = fasp_blas_darray_norm2(m,r);
00160             relres = absres/normr0;
00161             break;
00162         case STOP_REL_PRECRES:
00163             // z = B(r)
00164             if ( pc != NULL )
00165                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00166             else
00167                 fasp_darray_cp(m,r,z); /* No preconditioner */
00168             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00169             relres = absres/normr0;
00170             break;
00171         case STOP_MOD_REL_RES:
00172             absres = fasp_blas_darray_norm2(m,r);
00173             relres = absres/normu;
00174             break;
00175     }
00176
00177     // compute reduction factor of residual ||r||
00178     factor = absres/absres0;
00179
00180     // output iteration information if needed
00181     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00182
00183     // if the solution is NAN, restore the best solution
00184     if ( fasp_dvec_isnan(u) ) {
00185         absres = BIGREAL;
00186         goto RESTORE_BESTSOL;
00187     }
00188
00189     // safety net check: save the best-so-far solution
00190     if ( absres < absres_best - maxdiff ) {
00191         absres_best = absres;
00192         iter_best = iter;
00193         fasp_darray_cp(m,u->val,u_best);
00194     }
00195
00196     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00197     normuinf = fasp_blas_darray_norminf(m, u->val);
00198     if ( normuinf <= sol_inf_tol ) {
00199         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00200         iter = ERROR_SOLVER_SOLSTAG;
00201         break;
00202     }
00203
00204     // Check II: if staggenated, try to restart
00205     normu = fasp_blas_dvec_norm2(u);
00206
00207     // compute relative difference
00208     reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00209     if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00210
00211         if ( PrtLvl >= PRINT_MORE ) {
00212             ITS_DIFFRES(reldiff,relres);
00213             ITS_RESTART;
00214         }

```

```

00215
00216     fasp_darray_cp(m,b->val,r);
00217     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00218
00219     // compute residuals
00220     switch ( StopType ) {
00221     case STOP_REL_RES:
00222         absres = fasp_blas_darray_norm2(m,r);
00223         relres = absres/normr0;
00224         break;
00225     case STOP_REL_PRECRES:
00226         // z = B(r)
00227         if ( pc != NULL )
00228             pc->fct(r,z,pc->data); /* Apply preconditioner */
00229         else
00230             fasp_darray_cp(m,r,z); /* No preconditioner */
00231         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00232         relres = absres/normr0;
00233         break;
00234     case STOP_MOD_REL_RES:
00235         absres = fasp_blas_darray_norm2(m,r);
00236         relres = absres/normu;
00237         break;
00238     }
00239
00240     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00241
00242     if ( relres < tol )
00243         break;
00244     else {
00245         if ( stag >= MaxStag ) {
00246             if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00247             iter = ERROR_SOLVER_STAG;
00248             break;
00249         }
00250         fasp_darray_set(m,p,0.0);
00251         ++stag;
00252         ++restart_step;
00253     }
00254 } // end of stagnation check!
00255
00256 // Check III: prevent false convergence
00257 if ( relres < tol ) {
00258
00259     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00260
00261     // compute residual r = b - Ax again
00262     fasp_darray_cp(m,b->val,r);
00263     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00264
00265     // compute residuals
00266     switch ( StopType ) {
00267     case STOP_REL_RES:
00268         absres = fasp_blas_darray_norm2(m,r);
00269         relres = absres/normr0;
00270         break;
00271     case STOP_REL_PRECRES:
00272         // z = B(r)
00273         if ( pc != NULL )
00274             pc->fct(r,z,pc->data); /* Apply preconditioner */
00275         else
00276             fasp_darray_cp(m,r,z); /* No preconditioner */
00277         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00278         relres = absres/normr0;
00279         break;
00280     case STOP_MOD_REL_RES:
00281         absres = fasp_blas_darray_norm2(m,r);
00282         relres = absres/normu;
00283         break;
00284     }
00285
00286     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00287
00288     // check convergence
00289     if ( relres < tol ) break;
00290
00291     if ( more_step >= MaxRestartStep ) {
00292         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00293         iter = ERROR_SOLVER_TOLSMALL;
00294         break;
00295     }

```

```

00296
00297         // prepare for restarting the method
00298         fasp_darray_set(m,p,0.0);
00299         ++more_step;
00300         ++restart_step;
00301
00302     } // end of safe-guard check!
00303
00304     // save residual for next iteration
00305     absres0 = absres;
00306
00307     // compute z_k = B(r_k)
00308     if ( StopType != STOP_REL_PRECRES ) {
00309         if ( pc != NULL )
00310             pc->fct(r,z,pc->data); /* Apply preconditioner */
00311         else
00312             fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00313     }
00314
00315     // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
00316     temp2 = fasp_blas_darray_dotprod(m,z,r);
00317     beta = temp2/temp1;
00318     temp1 = temp2;
00319
00320     // compute p_k = z_k + beta_k*p_{k-1}
00321     fasp_blas_darray_axpby(m,1.0,z,beta,p);
00322
00323 } // end of main PCG loop.
00324
00325 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00326 if ( iter != iter_best ) {
00327     // compute best residual
00328     fasp_darray_cp(m,b->val,r);
00329     fasp_blas_dcsr_aAxpby(-1.0,A,u_best,r);
00330
00331     switch ( StopType ) {
00332     case STOP_REL_RES:
00333         absres_best = fasp_blas_darray_norm2(m,r);
00334         break;
00335     case STOP_REL_PRECRES:
00336         // z = B(r)
00337         if ( pc != NULL )
00338             pc->fct(r,z,pc->data); /* Apply preconditioner */
00339         else
00340             fasp_darray_cp(m,r,z); /* No preconditioner */
00341         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00342         break;
00343     case STOP_MOD_REL_RES:
00344         absres_best = fasp_blas_darray_norm2(m,r);
00345         break;
00346     }
00347
00348     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00349         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00350         fasp_darray_cp(m,u_best,u->val);
00351         relres = absres_best / normr0;
00352     }
00353 }
00354 }
00355
00356 FINISHED: // finish the iterative method
00357 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00358
00359 // clean up temp memory
00360 fasp_mem_free(work); work = NULL;
00361
00362 #if DEBUG_MODE > 0
00363     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00364 #endif
00365
00366     if ( iter > MaxIt )
00367         return ERROR_SOLVER_MAXIT;
00368     else
00369         return iter;
00370 }
00371
00393 INT fasp_solver_dblc_spcg (const dBLCmat *A,
00394                          const dvector *b,
00395                          dvector *u,
00396                          precondition *pc,
00397                          const REAL tol,

```

```

00398             const INT      MaxIt,
00399             const SHORT     StopType,
00400             const SHORT     PrtLvl)
00401 {
00402     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00403     const INT   m = b->row;
00404     const REAL  maxdiff = tol*STAG_RATIO; // stagantation tolerance
00405     const REAL  sol_inf_tol = SMALLREAL; // infinity norm tolerance
00406
00407     // local variables
00408     INT         iter = 0, stag = 1, more_step = 1, restart_step = 1;
00409     REAL         absres0 = BIGREAL, absres = BIGREAL;
00410     REAL         relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00411     REAL         reldiff, factor, normuinf;
00412     REAL         alpha, beta, templ, temp2;
00413     INT         iter_best = 0; // initial best known iteration
00414     REAL         absres_best = BIGREAL; // initial best known residual
00415
00416     // allocate temp memory (need 5*m REAL numbers)
00417     REAL *work = (REAL *)fasp_mem_calloc(5*m,sizeof(REAL));
00418     REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;
00419
00420     // Output some info for debugging
00421     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe CG solver (BLC) ...\n");
00422
00423     #if DEBUG_MODE > 0
00424     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00425     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00426     #endif
00427
00428     // r = b-A*u
00429     fasp_darray_cp(m,b->val,r);
00430     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00431
00432     if (pc != NULL)
00433         pc->fct(r,z,pc->data); /* Apply preconditioner */
00434     else
00435         fasp_darray_cp(m,r,z); /* No preconditioner */
00436
00437     // compute initial residuals
00438     switch (StopType) {
00439     case STOP_REL_RES:
00440         absres0 = fasp_blas_darray_norm2(m,r);
00441         normr0 = MAX(SMALLREAL,absres0);
00442         relres = absres0/normr0;
00443         break;
00444     case STOP_REL_PRECRES:
00445         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
00446         normr0 = MAX(SMALLREAL,absres0);
00447         relres = absres0/normr0;
00448         break;
00449     case STOP_MOD_REL_RES:
00450         absres0 = fasp_blas_darray_norm2(m,r);
00451         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00452         relres = absres0/normu;
00453         break;
00454     default:
00455         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00456         goto FINISHED;
00457     }
00458
00459     // if initial residual is small, no need to iterate!
00460     if ( relres < tol ) goto FINISHED;
00461
00462     // output iteration information if needed
00463     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00464
00465     fasp_darray_cp(m,z,p);
00466     templ = fasp_blas_darray_dotprod(m,z,r);
00467
00468     // main PCG loop
00469     while ( iter++ < MaxIt ) {
00470
00471         // t=A*p
00472         fasp_blas_dblc_mxv(A,p,t);
00473
00474         // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00475         temp2 = fasp_blas_darray_dotprod(m,t,p);
00476         if ( ABS(temp2) > SMALLREAL2 ) {
00477             alpha = templ/temp2;
00478         }

```

```

00479     else { // Possible breakdown
00480         goto RESTORE_BESTSOL;
00481     }
00482
00483     // u_k=u_{k-1} + alpha_k*p_{k-1}
00484     fasp_blas_darray_axpy(m,alpha,p,u->val);
00485
00486     // r_k=r_{k-1} - alpha_k*A*p_{k-1}
00487     fasp_blas_darray_axpy(m,-alpha,t,r);
00488
00489     // compute residuals
00490     switch ( StopType ) {
00491     case STOP_REL_RES:
00492         absres = fasp_blas_darray_norm2(m,r);
00493         relres = absres/normr0;
00494         break;
00495     case STOP_REL_PRECRES:
00496         // z = B(r)
00497         if ( pc != NULL )
00498             pc->fct(r,z,pc->data); /* Apply preconditioner */
00499         else
00500             fasp_darray_cp(m,r,z); /* No preconditioner */
00501         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00502         relres = absres/normr0;
00503         break;
00504     case STOP_MOD_REL_RES:
00505         absres = fasp_blas_darray_norm2(m,r);
00506         relres = absres/normu;
00507         break;
00508     }
00509
00510     // compute reduction factor of residual ||r||
00511     factor = absres/absres0;
00512
00513     // output iteration information if needed
00514     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00515
00516     // if the solution is NAN, restore the best solution
00517     if ( fasp_dvec_isnan(u) ) {
00518         absres = BIGREAL;
00519         goto RESTORE_BESTSOL;
00520     }
00521
00522     // safety net check: save the best-so-far solution
00523     if ( absres < absres_best - maxdiff ) {
00524         absres_best = absres;
00525         iter_best = iter;
00526         fasp_darray_cp(m,u->val,u_best);
00527     }
00528
00529     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00530     normuinf = fasp_blas_darray_norminf(m, u->val);
00531     if ( normuinf <= sol_inf_tol ) {
00532         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00533         iter = ERROR_SOLVER_SOLSTAG;
00534         break;
00535     }
00536
00537     // Check II: if staggenated, try to restart
00538     normu = fasp_blas_dvec_norm2(u);
00539
00540     // compute relative difference
00541     reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00542     if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00543
00544         if ( PrtLvl >= PRINT_MORE ) {
00545             ITS_DIFFRES(reldiff,relres);
00546             ITS_RESTART;
00547         }
00548
00549         fasp_darray_cp(m,b->val,r);
00550         fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00551
00552         // compute residuals
00553         switch ( StopType ) {
00554         case STOP_REL_RES:
00555             absres = fasp_blas_darray_norm2(m,r);
00556             relres = absres/normr0;
00557             break;
00558         case STOP_REL_PRECRES:
00559             // z = B(r)

```

```

00560         if ( pc != NULL )
00561             pc->fct(r,z,pc->data); /* Apply preconditioner */
00562         else
00563             fasp_darray_cp(m,r,z); /* No preconditioner */
00564         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00565         relres = absres/normr0;
00566         break;
00567     case STOP_MOD_REL_RES:
00568         absres = fasp_blas_darray_norm2(m,r);
00569         relres = absres/normu;
00570         break;
00571     }
00572
00573     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00574
00575     if ( relres < tol )
00576         break;
00577     else {
00578         if ( stag >= MaxStag ) {
00579             if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00580             iter = ERROR_SOLVER_STAG;
00581             break;
00582         }
00583         fasp_darray_set(m,p,0.0);
00584         ++stag;
00585         ++restart_step;
00586     }
00587 } // end of stagnation check!
00588
00589 // Check III: prevent false convergence
00590 if ( relres < tol ) {
00591
00592     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00593
00594     // compute residual r = b - Ax again
00595     fasp_darray_cp(m,b->val,r);
00596     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00597
00598     // compute residuals
00599     switch ( StopType ) {
00600     case STOP_REL_RES:
00601         absres = fasp_blas_darray_norm2(m,r);
00602         relres = absres/normr0;
00603         break;
00604     case STOP_REL_PRECRES:
00605         // z = B(r)
00606         if ( pc != NULL )
00607             pc->fct(r,z,pc->data); /* Apply preconditioner */
00608         else
00609             fasp_darray_cp(m,r,z); /* No preconditioner */
00610         absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00611         relres = absres/normr0;
00612         break;
00613     case STOP_MOD_REL_RES:
00614         absres = fasp_blas_darray_norm2(m,r);
00615         relres = absres/normu;
00616         break;
00617     }
00618
00619     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00620
00621     // check convergence
00622     if ( relres < tol ) break;
00623
00624     if ( more_step >= MaxRestartStep ) {
00625         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00626         iter = ERROR_SOLVER_TOLSMALL;
00627         break;
00628     }
00629
00630     // prepare for restarting the method
00631     fasp_darray_set(m,p,0.0);
00632     ++more_step;
00633     ++restart_step;
00634
00635 } // end of safe-guard check!
00636
00637 // save residual for next iteration
00638 absres0 = absres;
00639
00640 // compute z_k = B(r_k)

```



```

00641         if ( StopType != STOP_REL_PRECRES ) {
00642             if ( pc != NULL )
00643                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00644             else
00645                 fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00646         }
00647
00648         // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
00649         temp2 = fasp_blas_darray_dotprod(m,z,r);
00650         beta = temp2/temp1;
00651         temp1 = temp2;
00652
00653         // compute p_k = z_k + beta_k*p_{k-1}
00654         fasp_blas_darray_axpby(m,1.0,z,beta,p);
00655
00656     } // end of main PCG loop.
00657
00658 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00659     if ( iter != iter_best ) {
00660
00661         // compute best residual
00662         fasp_darray_cp(m,b->val,r);
00663         fasp_blas_dblc_aAxy(-1.0,A,u_best,r);
00664
00665         switch ( StopType ) {
00666             case STOP_REL_RES:
00667                 absres_best = fasp_blas_darray_norm2(m,r);
00668                 break;
00669             case STOP_REL_PRECRES:
00670                 // z = B(r)
00671                 if ( pc != NULL )
00672                     pc->fct(r,z,pc->data); /* Apply preconditioner */
00673                 else
00674                     fasp_darray_cp(m,r,z); /* No preconditioner */
00675                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00676                 break;
00677             case STOP_MOD_REL_RES:
00678                 absres_best = fasp_blas_darray_norm2(m,r);
00679                 break;
00680         }
00681
00682         if ( absres > absres_best + maxdiff || isnan(absres) ) {
00683             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00684             fasp_darray_cp(m,u_best,u->val);
00685             relres = absres_best / normr0;
00686         }
00687     }
00688
00689 FINISHED: // finish the iterative method
00690     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00691
00692     // clean up temp memory
00693     fasp_mem_free(work); work = NULL;
00694
00695 #if DEBUG_MODE > 0
00696     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00697 #endif
00698
00699     if ( iter > MaxIt )
00700         return ERROR_SOLVER_MAXIT;
00701     else
00702         return iter;
00703 }
00704
00726 INT fasp_solver_dstr_spcg (const dSTRmat *A,
00727                          const dvector *b,
00728                          dvector *u,
00729                          precondition *pc,
00730                          const REAL tol,
00731                          const INT MaxIt,
00732                          const SHORT StopType,
00733                          const SHORT PrtLvl)
00734 {
00735     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00736     const INT m = b->row;
00737     const REAL maxdiff = tol*STAG_RATIO; // staganation tolerance
00738     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00739
00740     // local variables
00741     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00742     REAL absres0 = BIGREAL, absres = BIGREAL;

```

```

00743     REAL      relres = BIGREAL, normu = BIGREAL, normr0 = BIGREAL;
00744     REAL      reldiff, factor, normuinf;
00745     REAL      alpha, beta, templ, temp2;
00746     INT       iter_best = 0; // initial best known iteration
00747     REAL      absres_best = BIGREAL; // initial best known residual
00748
00749     // allocate temp memory (need 5*m REAL numbers)
00750     REAL *work = (REAL *)fasp_mem_calloc(5*m,sizeof(REAL));
00751     REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;
00752
00753     // Output some info for debugging
00754     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe CG solver (STR) ...\n");
00755
00756 #if DEBUG_MODE > 0
00757     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00758     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00759 #endif
00760
00761     // r = b-A*u
00762     fasp_darray_cp(m,b->val,r);
00763     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
00764
00765     if (pc != NULL)
00766         pc->fct(r,z,pc->data); /* Apply preconditioner */
00767     else
00768         fasp_darray_cp(m,r,z); /* No preconditioner */
00769
00770     // compute initial residuals
00771     switch (StopType) {
00772     case STOP_REL_RES:
00773         absres0 = fasp_blas_darray_norm2(m,r);
00774         normr0 = MAX(SMALLREAL,absres0);
00775         relres = absres0/normr0;
00776         break;
00777     case STOP_REL_PRECRES:
00778         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
00779         normr0 = MAX(SMALLREAL,absres0);
00780         relres = absres0/normr0;
00781         break;
00782     case STOP_MOD_REL_RES:
00783         absres0 = fasp_blas_darray_norm2(m,r);
00784         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00785         relres = absres0/normu;
00786         break;
00787     default:
00788         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00789         goto FINISHED;
00790     }
00791
00792     // if initial residual is small, no need to iterate!
00793     if ( relres < tol ) goto FINISHED;
00794
00795     // output iteration information if needed
00796     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00797
00798     fasp_darray_cp(m,z,p);
00799     templ = fasp_blas_darray_dotprod(m,z,r);
00800
00801     // main PCG loop
00802     while ( iter++ < MaxIt ) {
00803
00804         // t=A*p
00805         fasp_blas_dstr_mxv(A,p,t);
00806
00807         // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
00808         temp2 = fasp_blas_darray_dotprod(m,t,p);
00809         if ( ABS(temp2) > SMALLREAL2 ) {
00810             alpha = templ/temp2;
00811         }
00812         else { // Possible breakdown
00813             goto RESTORE_BESTSOL;
00814         }
00815
00816         // u_k=u_{k-1} + alpha_k*p_{k-1}
00817         fasp_blas_darray_axpy(m,alpha,p,u->val);
00818
00819         // r_k=r_{k-1} - alpha_k*A*p_{k-1}
00820         fasp_blas_darray_axpy(m,-alpha,t,r);
00821
00822         // compute residuals
00823         switch ( StopType ) {

```

```

00824         case STOP_REL_RES:
00825             absres = fasp_blas_darray_norm2(m,r);
00826             relres = absres/normr0;
00827             break;
00828         case STOP_REL_PRECRES:
00829             // z = B(r)
00830             if ( pc != NULL )
00831                 pc->fct(r,z,pc->data); /* Apply preconditioner */
00832             else
00833                 fasp_darray_cp(m,r,z); /* No preconditioner */
00834             absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00835             relres = absres/normr0;
00836             break;
00837         case STOP_MOD_REL_RES:
00838             absres = fasp_blas_darray_norm2(m,r);
00839             relres = absres/normu;
00840             break;
00841     }
00842
00843     // compute reduction factor of residual ||r||
00844     factor = absres/absres0;
00845
00846     // output iteration information if needed
00847     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00848
00849     // if the solution is NAN, restore the best solution
00850     if ( fasp_dvec_isnan(u) ) {
00851         absres = BIGREAL;
00852         goto RESTORE_BESTSOL;
00853     }
00854
00855     // safety net check: save the best-so-far solution
00856     if ( absres < absres_best - maxdiff ) {
00857         absres_best = absres;
00858         iter_best = iter;
00859         fasp_darray_cp(m,u->val,u_best);
00860     }
00861
00862     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00863     normuinf = fasp_blas_darray_norminf(m, u->val);
00864     if ( normuinf <= sol_inf_tol ) {
00865         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00866         iter = ERROR_SOLVER_SOLSTAG;
00867         break;
00868     }
00869
00870     // Check II: if staggenated, try to restart
00871     normu = fasp_blas_dvec_norm2(u);
00872
00873     // compute relative difference
00874     reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
00875     if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {
00876
00877         if ( PrtLvl >= PRINT_MORE ) {
00878             ITS_DIFFRES(reldiff,relres);
00879             ITS_RESTART;
00880         }
00881
00882         fasp_darray_cp(m,b->val,r);
00883         fasp_blas_dstr_aApy(-1.0,A,u->val,r);
00884
00885         // compute residuals
00886         switch ( StopType ) {
00887             case STOP_REL_RES:
00888                 absres = fasp_blas_darray_norm2(m,r);
00889                 relres = absres/normr0;
00890                 break;
00891             case STOP_REL_PRECRES:
00892                 // z = B(r)
00893                 if ( pc != NULL )
00894                     pc->fct(r,z,pc->data); /* Apply preconditioner */
00895                 else
00896                     fasp_darray_cp(m,r,z); /* No preconditioner */
00897                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00898                 relres = absres/normr0;
00899                 break;
00900             case STOP_MOD_REL_RES:
00901                 absres = fasp_blas_darray_norm2(m,r);
00902                 relres = absres/normu;
00903                 break;
00904         }

```

```

00905
00906         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00907
00908         if ( relres < tol )
00909             break;
00910         else {
00911             if ( stag >= MaxStag ) {
00912                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00913                 iter = ERROR_SOLVER_STAG;
00914                 break;
00915             }
00916             fasp_darray_set(m,p,0.0);
00917             ++stag;
00918             ++restart_step;
00919         }
00920     } // end of stagnation check!
00921
00922     // Check III: prevent false convergence
00923     if ( relres < tol ) {
00924
00925         if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00926
00927         // compute residual r = b - Ax again
00928         fasp_darray_cp(m,b->val,r);
00929         fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
00930
00931         // compute residuals
00932         switch ( StopType ) {
00933             case STOP_REL_RES:
00934                 absres = fasp_blas_darray_norm2(m,r);
00935                 relres = absres/normr0;
00936                 break;
00937             case STOP_REL_PRECRES:
00938                 // z = B(r)
00939                 if ( pc != NULL )
00940                     pc->fct(r,z,pc->data); /* Apply preconditioner */
00941                 else
00942                     fasp_darray_cp(m,r,z); /* No preconditioner */
00943                 absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
00944                 relres = absres/normr0;
00945                 break;
00946             case STOP_MOD_REL_RES:
00947                 absres = fasp_blas_darray_norm2(m,r);
00948                 relres = absres/normu;
00949                 break;
00950         }
00951
00952         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00953
00954         // check convergence
00955         if ( relres < tol ) break;
00956
00957         if ( more_step >= MaxRestartStep ) {
00958             if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00959             iter = ERROR_SOLVER_TOLSMALL;
00960             break;
00961         }
00962
00963         // prepare for restarting the method
00964         fasp_darray_set(m,p,0.0);
00965         ++more_step;
00966         ++restart_step;
00967
00968     } // end of safe-guard check!
00969
00970     // save residual for next iteration
00971     absres0 = absres;
00972
00973     // compute z_k = B(r_k)
00974     if ( StopType != STOP_REL_PRECRES ) {
00975         if ( pc != NULL )
00976             pc->fct(r,z,pc->data); /* Apply preconditioner */
00977         else
00978             fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
00979     }
00980
00981     // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
00982     temp2 = fasp_blas_darray_dotprod(m,z,r);
00983     beta = temp2/temp1;
00984     temp1 = temp2;
00985

```

```

00986         // compute p_k = z_k + beta_k*p_{k-1}
00987         fasp_blas_darray_axpy(m,1.0,z,beta,p);
00988
00989     } // end of main PCG loop.
00990
00991 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00992     if ( iter != iter_best ) {
00993
00994         // compute best residual
00995         fasp_darray_cp(m,b->val,r);
00996         fasp_blas_dstr_aApy(-1.0,A,u_best,r);
00997
00998         switch ( StopType ) {
00999             case STOP_REL_RES:
01000                 absres_best = fasp_blas_darray_norm2(m,r);
01001                 break;
01002             case STOP_REL_PRECRES:
01003                 // z = B(r)
01004                 if ( pc != NULL )
01005                     pc->fct(r,z,pc->data); /* Apply preconditioner */
01006                 else
01007                     fasp_darray_cp(m,r,z); /* No preconditioner */
01008                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
01009                 break;
01010             case STOP_MOD_REL_RES:
01011                 absres_best = fasp_blas_darray_norm2(m,r);
01012                 break;
01013         }
01014
01015         if ( absres > absres_best + maxdiff || isnan(absres) ) {
01016             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01017             fasp_darray_cp(m,u_best,u->val);
01018             relres = absres_best / normr0;
01019         }
01020     }
01021
01022 FINISHED: // finish the iterative method
01023     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01024
01025     // clean up temp memory
01026     fasp_mem_free(work); work = NULL;
01027
01028 #if DEBUG_MODE > 0
01029     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01030 #endif
01031
01032     if ( iter > MaxIt )
01033         return ERROR_SOLVER_MAXIT;
01034     else
01035         return iter;
01036 }
01037
01038 /*-----*/
01039 /*--          End of File          --*/
01040 /*-----*/

```

9.129 KrySPgmres.c File Reference

Krylov subspace methods – Preconditioned GMRes with safety net.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"
#include "KryUtil.inl"

```

Functions

- `INT fasp_solver_dcsr_spgmres` (const `dCSRmat` *A, const `dvector` *b, `dvector` *x, `precond` *pc, const `REAL` tol, const `INT` MaxIt, `SHORT` restart, const `SHORT` StopType, const `SHORT` PrtLvl)

Preconditioned GMRES method for solving $Au=b$ with safe-guard.

- [INT fasp_solver_dbsr_spgmres](#) (const [dBSRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned GMRES method for solving $Au=b$ with safe-guard.
- [INT fasp_solver_dblc_spgmres](#) (const [dBLCmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned GMRES method for solving $Au=b$ with safe-guard.
- [INT fasp_solver_dstr_spgmres](#) (const [dSTRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned GMRES method for solving $Au=b$ with safe-guard.

9.129.1 Detailed Description

Krylov subspace methods – Preconditioned GMRes with safety net.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

See also [pgmres.c](#) for a variable restarting version.

The ‘best’ iterative solution will be saved and used upon exit; See [KryPgmres.c](#) for a version without safety net

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KrySPgmres.c](#).

9.129.2 Function Documentation

9.129.2.1 fasp_solver_dblc_spgmres()

```
INT fasp_solver_dblc_spgmres (
    const dBLCmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned GMRES method for solving $Au=b$ with safe-guard.

Parameters

<i>A</i>	Pointer to dBLCmat : coefficient matrix
<i>b</i>	Pointer to dvector : right hand side
<i>x</i>	Pointer to dvector : unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping

Parameters

<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/05/2013

Definition at line 752 of file [KrySPgmres.c](#).

9.129.2.2 fasp_solver_dbsr_spgmres()

```

INT fasp_solver_dbsr_spgmres (
    const dBSRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned GMRES method for solving $Au=b$ with safe-guard.

Parameters

<i>A</i>	Pointer to dBSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/05/2013

Definition at line 409 of file [KrySPgmres.c](#).**9.129.2.3 fasp_solver_dcsr_spgmres()**

```
INT fasp_solver_dcsr_spgmres (
    const dCSRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )
```

Preconditioned GMRES method for solving $Au=b$ with safe-guard.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/05/2013

Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate

Definition at line 66 of file [KrySPgmres.c](#).

9.129.2.4 fasp_solver_dstr_spgmres()

```

INT fasp_solver_dstr_spgmres (
    const dSTRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned GMRES method for solving $Au=b$ with safe-guard.

Parameters

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/05/2013

Definition at line [1095](#) of file [KrySPgmres.c](#).

9.130 KrySPgmres.c

[Go to the documentation of this file.](#)

```

00001
00026 #include <math.h>
00027
00028 #include "fasp.h"
00029 #include "fasp_funcs.h"
00030
00031 /*-----*/
00032 /*--  Declare Private Functions  --*/
00033 /*-----*/
00034
00035 #include "KryUtil.inl"
00036
00037 /*-----*/
00038 /*--      Public Functions      --*/
00039 /*-----*/

```

```

00040
00066 INT fasp_solver_dcsr_spgmres (const dCSRmat *A,
00067                                const dvector *b,
00068                                dvector *x,
00069                                precondition *pc,
00070                                const REAL tol,
00071                                const INT MaxIt,
00072                                SHORT restart,
00073                                const SHORT StopType,
00074                                const SHORT PrtLvl)
00075 {
00076     const INT n = b->row;
00077     const INT MIN_ITER = 0;
00078     const REAL maxdiff = tol*STAG_RATIO; // staganation tolerance
00079     const REAL epsmac = SMALLREAL;
00080
00081     // local variables
00082     INT iter = 0;
00083     INT restart1 = restart + 1;
00084     int i, j, k; // must be signed! -zcs
00085
00086     REAL r_norm, r_normb, gamma, t;
00087     REAL normr0 = BIGREAL, absres = BIGREAL;
00088     REAL relres = BIGREAL, normu = BIGREAL;
00089
00090     INT iter_best = 0; // initial best known iteration
00091     REAL absres_best = BIGREAL; // initial best known residual
00092
00093     // allocate temp memory (need about (restart+4)*n REAL numbers)
00094     REAL *c = NULL, *s = NULL, *rs = NULL;
00095     REAL *norms = NULL, *r = NULL, *w = NULL;
00096     REAL *work = NULL, *x_best = NULL;
00097     REAL *p = NULL, **hh = NULL;
00098
00099     // Output some info for debugging
00100     if (PrtLvl > PRINT_NONE) printf("\nCalling Safe GMRes solver (CSR) ...\n");
00101
00102     #if DEBUG_MODE > 0
00103     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00104     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00105     #endif
00106
00107     /* allocate memory and setup temp work space */
00108     work = (REAL *) fasp_mem_malloc((restart+4)*(restart+n)+1, sizeof(REAL));
00109
00110     /* check whether memory is enough for GMRES */
00111     while ( (work == NULL) && (restart > 5) ) {
00112         restart = restart - 5;
00113         work = (REAL *) fasp_mem_malloc((restart+4)*(restart+n)+1, sizeof(REAL));
00114         printf("### WARNING: GMRES restart number set to %d!\n", restart);
00115         restart1 = restart + 1;
00116     }
00117
00118     if (work == NULL) {
00119         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00120         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00121     }
00122
00123     p = (REAL **)fasp_mem_malloc(restart1, sizeof(REAL *));
00124     hh = (REAL **)fasp_mem_malloc(restart1, sizeof(REAL *));
00125     norms = (REAL *) fasp_mem_malloc(MaxIt+1, sizeof(REAL));
00126
00127     r = work; w = r + n; rs = w + n; c = rs + restart1;
00128     x_best = c + restart; s = x_best + n;
00129
00130     for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;
00131
00132     for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;
00133
00134     // r = b-A*x
00135     fasp_darray_cp(n, b->val, p[0]);
00136     fasp_blas_dcsr_aAxy(-1.0, A, x->val, p[0]);
00137
00138     r_norm = fasp_blas_darray_norm2(n,p[0]);
00139
00140     // compute initial residuals
00141     switch (StopType) {
00142     case STOP_REL_RES:
00143         normr0 = MAX(SMALLREAL, r_norm);
00144         relres = r_norm/normr0;
00145         break;

```

```

00146     case STOP_REL_PRECRES:
00147         if ( pc == NULL )
00148             fasp_darray_cp(n, p[0], r);
00149         else
00150             pc->fct(p[0], r, pc->data);
00151         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00152         normr0 = MAX(SMALLREAL,r_normb);
00153         relres = r_normb/normr0;
00154         break;
00155     case STOP_MOD_REL_RES:
00156         normu = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
00157         normr0 = r_norm;
00158         relres = normr0/normu;
00159         break;
00160     default:
00161         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00162         goto FINISHED;
00163 }
00164
00165 // if initial residual is small, no need to iterate!
00166 if ( relres < tol ) goto FINISHED;
00167
00168 // output iteration information if needed
00169 fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);
00170
00171 // store initial residual
00172 norms[0] = relres;
00173
00174 /* outer iteration cycle */
00175 while ( iter < MaxIt ) {
00176     rs[0] = r_norm;
00177
00178     t = 1.0 / r_norm;
00179
00180     fasp_blas_darray_ax(n, t, p[0]);
00181
00182     /* RESTART CYCLE (right-preconditioning) */
00183     i = 0;
00184     while ( i < restart && iter < MaxIt ) {
00185         i++; iter++;
00186
00187         /* apply preconditioner */
00188         if ( pc == NULL )
00189             fasp_darray_cp(n, p[i-1], r);
00190         else
00191             pc->fct(p[i-1], r, pc->data);
00192
00193         fasp_blas_dcsr_mxv(A, r, p[i]);
00194
00195         /* modified Gram_Schmidt */
00196         for ( j = 0; j < i; j++ ) {
00197             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00198             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00199         }
00200         t = fasp_blas_darray_norm2(n, p[i]);
00201         hh[i][i-1] = t;
00202         if ( t != 0.0 ) {
00203             t = 1.0/t;
00204             fasp_blas_darray_ax(n, t, p[i]);
00205         }
00206
00207         for ( j = 1; j < i; j++ ) {
00208             t = hh[j-1][i-1];
00209             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00210             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00211         }
00212         t = hh[i][i-1]*hh[i][i-1];
00213         t+= hh[i-1][i-1]*hh[i-1][i-1];
00214
00215         gamma = sqrt(t);
00216         if ( gamma == 0.0 ) gamma = epsmac;
00217         c[i-1] = hh[i-1][i-1] / gamma;
00218         s[i-1] = hh[i][i-1] / gamma;
00219         rs[i] = -s[i-1]*rs[i-1];
00220         rs[i-1] = c[i-1]*rs[i-1];
00221         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00222
00223         absres = r_norm = fabs(rs[i]);
00224
00225
00226

```

```

00227         relres = absres/normr0;
00228
00229         norms[iter] = relres;
00230
00231         // output iteration information if needed
00232         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00233                   norms[iter]/norms[iter-1]);
00234
00235         // should we exit restart cycle
00236         if ( relres <= tol && iter >= MIN_ITER ) break;
00237
00238     } /* end of restart cycle */
00239
00240     /* compute solution, first solve upper triangular system */
00241     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00242     for ( k = i-2; k >= 0; k-- ) {
00243         t = 0.0;
00244         for ( j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
00245
00246         t += rs[k];
00247         rs[k] = t / hh[k][k];
00248     }
00249
00250     fasp_darray_cp(n, p[i-1], w);
00251
00252     fasp_blas_darray_ax(n, rs[i-1], w);
00253
00254     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00255
00256     /* apply preconditioner */
00257     if ( pc == NULL )
00258         fasp_darray_cp(n, w, r);
00259     else
00260         pc->fct(w, r, pc->data);
00261
00262     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00263
00264     // safety net check: save the best-so-far solution
00265     if ( fasp_dvec_isnan(x) ) {
00266         // If the solution is NAN, restore the best solution
00267         absres = BIGREAL;
00268         goto RESTORE_BESTSOL;
00269     }
00270
00271     if ( absres < absres_best - maxdiff ) {
00272         absres_best = absres;
00273         iter_best = iter;
00274         fasp_darray_cp(n, x->val, x_best);
00275     }
00276
00277     // Check: prevent false convergence
00278     if ( relres <= tol && iter >= MIN_ITER ) {
00279
00280         fasp_darray_cp(n, b->val, r);
00281         fasp_blas_dcsr_aApy(-1.0, A, x->val, r);
00282
00283         r_norm = fasp_blas_darray_norm2(n, r);
00284
00285         switch ( StopType ) {
00286             case STOP_REL_RES:
00287                 absres = r_norm;
00288                 relres = absres/normr0;
00289                 break;
00290             case STOP_REL_PRECRES:
00291                 if ( pc == NULL )
00292                     fasp_darray_cp(n, r, w);
00293                 else
00294                     pc->fct(r, w, pc->data);
00295                 absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00296                 relres = absres/normr0;
00297                 break;
00298             case STOP_MOD_REL_RES:
00299                 absres = r_norm;
00300                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00301                 relres = absres/normu;
00302                 break;
00303         }
00304
00305         norms[iter] = relres;
00306
00307         if ( relres <= tol ) {

```

```

00308         break;
00309     }
00310     else {
00311         // Need to restart
00312         fasp_darray_cp(n, r, p[0]); i = 0;
00313     }
00314 } /* end of convergence check */
00315
00316 /* compute residual vector and continue loop */
00317 for (j = i; j > 0; j--) {
00318     rs[j-1] = -s[j-1]*rs[j];
00319     rs[j] = c[j-1]*rs[j];
00320 }
00321
00322 if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00323
00324 for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00325
00326 if ( i ) {
00327     fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00328     fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00329 }
00330 }
00331 } /* end of main while loop */
00332
00333 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00334 if ( iter != iter_best ) {
00335     // compute best residual
00336     fasp_darray_cp(n,b->val,r);
00337     fasp_blas_dcsr_aAxy(-1.0,A,x_best,r);
00338
00339     switch ( StopType ) {
00340     case STOP_REL_RES:
00341         absres_best = fasp_blas_darray_norm2(n,r);
00342         break;
00343     case STOP_REL_PRECRES:
00344         // z = B(r)
00345         if ( pc != NULL )
00346             pc->fct(r,w,pc->data); /* Apply preconditioner */
00347         else
00348             fasp_darray_cp(n,r,w); /* No preconditioner */
00349         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
00350         break;
00351     case STOP_MOD_REL_RES:
00352         absres_best = fasp_blas_darray_norm2(n,r);
00353         break;
00354     }
00355
00356 if ( absres > absres_best + maxdiff || isnan(absres) ) {
00357     if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00358     fasp_darray_cp(n,x_best,x->val);
00359     relres = absres_best / normr0;
00360 }
00361 }
00362 }
00363 }
00364
00365 FINISHED:
00366 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00367
00368 /*-----
00369 * Clean up workspace
00370 *-----*/
00371 fasp_mem_free(work); work = NULL;
00372 fasp_mem_free(p); p = NULL;
00373 fasp_mem_free(hh); hh = NULL;
00374 fasp_mem_free(norms); norms = NULL;
00375
00376 #if DEBUG_MODE > 0
00377 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00378 #endif
00379
00380 if ( iter >= MaxIt )
00381     return ERROR_SOLVER_MAXIT;
00382 else
00383     return iter;
00384 }
00385
00409 INT fasp_solver_dbsr_spgmres (const dBSRmat *A,
00410                             const dvector *b,
00411                             dvector *x,

```

```

00412             precondition      *pc,
00413             const REAL          tol,
00414             const INT           MaxIt,
00415             SHORT               restart,
00416             const SHORT         StopType,
00417             const SHORT         PrtLvl)
00418 {
00419     const INT n = b->row;
00420     const INT MIN_ITER = 0;
00421     const REAL maxdiff = tol*STAG_RATIO; // staganation tolerance
00422     const REAL epsmac = SMALLREAL;
00423
00424     // local variables
00425     INT iter = 0;
00426     INT restart1 = restart + 1;
00427     int i, j, k; // must be signed! -zcs
00428
00429     REAL r_norm, r_normb, gamma, t;
00430     REAL normr0 = BIGREAL, absres = BIGREAL;
00431     REAL relres = BIGREAL, normu = BIGREAL;
00432
00433     INT iter_best = 0; // initial best known iteration
00434     REAL absres_best = BIGREAL; // initial best known residual
00435
00436     // allocate temp memory (need about (restart+4)*n REAL numbers)
00437     REAL *c = NULL, *s = NULL, *rs = NULL;
00438     REAL *norms = NULL, *r = NULL, *w = NULL;
00439     REAL *work = NULL, *x_best = NULL;
00440     REAL **p = NULL, **hh = NULL;
00441
00442     // Output some info for debugging
00443     if (PrtLvl > PRINT_NONE) printf("\nCalling Safe GMRes solver (BSR) ...\n");
00444
00445     #if DEBUG_MODE > 0
00446     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00447     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00448     #endif
00449
00450     /* allocate memory and setup temp work space */
00451     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00452
00453     /* check whether memory is enough for GMRES */
00454     while ( (work == NULL) && (restart > 5) ) {
00455         restart = restart - 5;
00456         work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00457         printf("### WARNING: GMRES restart number set to %d!\n", restart);
00458         restart1 = restart + 1;
00459     }
00460
00461     if (work == NULL) {
00462         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00463         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00464     }
00465
00466     p = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00467     hh = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00468     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00469
00470     r = work; w = r + n; rs = w + n; c = rs + restart1;
00471     x_best = c + restart; s = x_best + n;
00472
00473     for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;
00474
00475     for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;
00476
00477     // r = b-A*x
00478     fasp_darray_cp(n, b->val, p[0]);
00479     fasp_blas_dbsr_aAxp(-1.0, A, x->val, p[0]);
00480
00481     r_norm = fasp_blas_darray_norm2(n, p[0]);
00482
00483     // compute initial residuals
00484     switch (StopType) {
00485     case STOP_REL_RES:
00486         normr0 = MAX(SMALLREAL, r_norm);
00487         relres = r_norm/normr0;
00488         break;
00489     case STOP_REL_PRECRES:
00490         if (pc == NULL)
00491             fasp_darray_cp(n, p[0], r);
00492         else

```

```

00493         pc->fct(p[0], r, pc->data);
00494         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00495         normr0  = MAX(SMALLREAL,r_normb);
00496         relres  = r_normb/normr0;
00497         break;
00498     case STOP_MOD_REL_RES:
00499         normu    = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
00500         normr0  = r_norm;
00501         relres  = normr0/normu;
00502         break;
00503     default:
00504         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00505         goto FINISHED;
00506 }
00507
00508 // if initial residual is small, no need to iterate!
00509 if ( relres < tol ) goto FINISHED;
00510
00511 // output iteration information if needed
00512 fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);
00513
00514 // store initial residual
00515 norms[0] = relres;
00516
00517 /* outer iteration cycle */
00518 while ( iter < MaxIt ) {
00519
00520     rs[0] = r_norm;
00521
00522     t = 1.0 / r_norm;
00523
00524     fasp_blas_darray_ax(n, t, p[0]);
00525
00526     /* RESTART CYCLE (right-preconditioning) */
00527     i = 0;
00528     while ( i < restart && iter < MaxIt ) {
00529
00530         i++; iter++;
00531
00532         /* apply preconditioner */
00533         if ( pc == NULL )
00534             fasp_darray_cp(n, p[i-1], r);
00535         else
00536             pc->fct(p[i-1], r, pc->data);
00537
00538         fasp_blas_dbsr_mxv(A, r, p[i]);
00539
00540         /* modified Gram_Schmidt */
00541         for ( j = 0; j < i; j++ ) {
00542             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00543             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00544         }
00545         t = fasp_blas_darray_norm2(n, p[i]);
00546         hh[i][i-1] = t;
00547         if ( t != 0.0 ) {
00548             t = 1.0/t;
00549             fasp_blas_darray_ax(n, t, p[i]);
00550         }
00551
00552         for ( j = 1; j < i; ++j ) {
00553             t = hh[j-1][i-1];
00554             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00555             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00556         }
00557         t = hh[i][i-1]*hh[i][i-1];
00558         t += hh[i-1][i-1]*hh[i-1][i-1];
00559
00560         gamma = sqrt(t);
00561         if (gamma == 0.0) gamma = epsmac;
00562         c[i-1] = hh[i-1][i-1] / gamma;
00563         s[i-1] = hh[i][i-1] / gamma;
00564         rs[i] = -s[i-1]*rs[i-1];
00565         rs[i-1] = c[i-1]*rs[i-1];
00566         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00567
00568         absres = r_norm = fabs(rs[i]);
00569
00570         relres = absres/normr0;
00571
00572         norms[iter] = relres;
00573

```

```

00574         // output iteration information if needed
00575         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00576                   norms[iter]/norms[iter-1]);
00577
00578         // should we exit restart cycle
00579         if ( relres <= tol && iter >= MIN_ITER ) break;
00580
00581     } /* end of restart cycle */
00582
00583     /* compute solution, first solve upper triangular system */
00584     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00585     for ( k = i-2; k >= 0; k-- ) {
00586         t = 0.0;
00587         for ( j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
00588
00589         t += rs[k];
00590         rs[k] = t / hh[k][k];
00591     }
00592
00593     fasp_darray_cp(n, p[i-1], w);
00594
00595     fasp_blas_darray_ax(n, rs[i-1], w);
00596
00597     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00598
00599     /* apply preconditioner */
00600     if ( pc == NULL )
00601         fasp_darray_cp(n, w, r);
00602     else
00603         pc->fct(w, r, pc->data);
00604
00605     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00606
00607     // safety net check: save the best-so-far solution
00608     if ( fasp_dvec_isnan(x) ) {
00609         // If the solution is NAN, restore the best solution
00610         absres = BIGREAL;
00611         goto RESTORE_BESTSOL;
00612     }
00613
00614     if ( absres < absres_best - maxdiff ) {
00615         absres_best = absres;
00616         iter_best = iter;
00617         fasp_darray_cp(n, x->val, x_best);
00618     }
00619
00620     // Check: prevent false convergence
00621     if ( relres <= tol && iter >= MIN_ITER ) {
00622
00623         fasp_darray_cp(n, b->val, r);
00624         fasp_blas_dbsr_aApy(-1.0, A, x->val, r);
00625
00626         r_norm = fasp_blas_darray_norm2(n, r);
00627
00628         switch ( StopType ) {
00629             case STOP_REL_RES:
00630                 absres = r_norm;
00631                 relres = absres/normr0;
00632                 break;
00633             case STOP_REL_PRECRES:
00634                 if ( pc == NULL )
00635                     fasp_darray_cp(n, r, w);
00636                 else
00637                     pc->fct(r, w, pc->data);
00638                 absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00639                 relres = absres/normr0;
00640                 break;
00641             case STOP_MOD_REL_RES:
00642                 absres = r_norm;
00643                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00644                 relres = absres/normu;
00645                 break;
00646         }
00647
00648         norms[iter] = relres;
00649
00650         if ( relres <= tol ) {
00651             break;
00652         }
00653         else {
00654             // Need to restart

```



```

00655         fasp_darray_cp(n, r, p[0]); i = 0;
00656     }
00657
00658     } /* end of convergence check */
00659
00660     /* compute residual vector and continue loop */
00661     for (j = i; j > 0; j--) {
00662         rs[j-1] = -s[j-1]*rs[j];
00663         rs[j] = c[j-1]*rs[j];
00664     }
00665
00666     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00667
00668     for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00669
00670     if ( i ) {
00671         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00672         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00673     }
00674
00675     } /* end of main while loop */
00676
00677 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00678     if ( iter != iter_best ) {
00679
00680         // compute best residual
00681         fasp_darray_cp(n,b->val,r);
00682         fasp_blas_dbsr_aAxy(-1.0,A,x_best,r);
00683
00684         switch ( StopType ) {
00685             case STOP_REL_RES:
00686                 absres_best = fasp_blas_darray_norm2(n,r);
00687                 break;
00688             case STOP_REL_PRECRES:
00689                 // z = B(r)
00690                 if ( pc != NULL )
00691                     pc->fct(r,w,pc->data); /* Apply preconditioner */
00692                 else
00693                     fasp_darray_cp(n,r,w); /* No preconditioner */
00694                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
00695                 break;
00696             case STOP_MOD_REL_RES:
00697                 absres_best = fasp_blas_darray_norm2(n,r);
00698                 break;
00699         }
00700
00701         if ( absres > absres_best + maxdiff || isnan(absres) ) {
00702             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00703             fasp_darray_cp(n,x_best,x->val);
00704             relres = absres_best / normr0;
00705         }
00706     }
00707
00708 FINISHED:
00709     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00710
00711     /*-----*/
00712 * Clean up workspace
00713 *-----*/
00714     fasp_mem_free(work); work = NULL;
00715     fasp_mem_free(p); p = NULL;
00716     fasp_mem_free(hh); hh = NULL;
00717     fasp_mem_free(norms); norms = NULL;
00718
00719 #if DEBUG_MODE > 0
00720     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00721 #endif
00722
00723     if ( iter >= MaxIt )
00724         return ERROR_SOLVER_MAXIT;
00725     else
00726         return iter;
00727 }
00728
00752 INT fasp_solver_dblc_spgmres (const dBLCmat *A,
00753                               const dvector *b,
00754                               dvector *x,
00755                               *pc,
00756                               const REAL tol,
00757                               const INT MaxIt,
00758                               SHORT restart,

```

```

00759             const SHORT      StopType,
00760             const SHORT      PrtLvl)
00761 {
00762     const INT  n          = b->row;
00763     const INT  MIN_ITER   = 0;
00764     const REAL maxdiff    = tol*STAG_RATIO; // staganation tolerance
00765     const REAL epsmac     = SMALLREAL;
00766
00767     // local variables
00768     INT      iter = 0;
00769     INT      restartl = restart + 1;
00770     int      i, j, k; // must be signed! -zcs
00771
00772     REAL      r_norm, r_normb, gamma, t;
00773     REAL      normr0 = BIGREAL, absres = BIGREAL;
00774     REAL      relres = BIGREAL, normu = BIGREAL;
00775
00776     INT      iter_best = 0; // initial best known iteration
00777     REAL      absres_best = BIGREAL; // initial best known residual
00778
00779     // allocate temp memory (need about (restart+4)*n REAL numbers)
00780     REAL      *c = NULL, *s = NULL, *rs = NULL;
00781     REAL      *norms = NULL, *r = NULL, *w = NULL;
00782     REAL      *work = NULL, *x_best = NULL;
00783     REAL      **p = NULL, **hh = NULL;
00784
00785     // Output some info for debugging
00786     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (BLC) ...\n");
00787
00788     #if DEBUG_MODE > 0
00789     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00790     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00791     #endif
00792
00793     /* allocate memory and setup temp work space */
00794     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00795
00796     /* check whether memory is enough for GMRES */
00797     while ( (work == NULL) && (restart > 5) ) {
00798         restart = restart - 5;
00799         work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00800         printf("### WARNING: GMRES restart number set to %d!\n", restart);
00801         restartl = restart + 1;
00802     }
00803
00804     if ( work == NULL ) {
00805         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00806         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00807     }
00808
00809     p = (REAL **)fasp_mem_calloc(restartl, sizeof(REAL *));
00810     hh = (REAL **)fasp_mem_calloc(restartl, sizeof(REAL *));
00811     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00812
00813     r = work; w = r + n; rs = w + n; c = rs + restartl;
00814     x_best = c + restart; s = x_best + n;
00815
00816     for ( i = 0; i < restartl; i++ ) p[i] = s + restart + i*n;
00817
00818     for ( i = 0; i < restartl; i++ ) hh[i] = p[restart] + n + i*restart;
00819
00820     // r = b-A*x
00821     fasp_darray_cp(n, b->val, p[0]);
00822     fasp_blas_dblc_aAxy(-1.0, A, x->val, p[0]);
00823
00824     r_norm = fasp_blas_darray_norm2(n,p[0]);
00825
00826     // compute initial residuals
00827     switch (StopType) {
00828     case STOP_REL_RES:
00829         normr0 = MAX(SMALLREAL, r_norm);
00830         relres = r_norm/normr0;
00831         break;
00832     case STOP_REL_PRECRES:
00833         if ( pc == NULL )
00834             fasp_darray_cp(n, p[0], r);
00835         else
00836             pc->fct(p[0], r, pc->data);
00837         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
00838         normr0 = MAX(SMALLREAL, r_normb);
00839         relres = r_normb/normr0;

```

```

00840         break;
00841     case STOP_MOD_REL_RES:
00842         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00843         normr0 = r_norm;
00844         relres = normr0/normu;
00845         break;
00846     default:
00847         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00848         goto FINISHED;
00849 }
00850
00851 // if initial residual is small, no need to iterate!
00852 if ( relres < tol ) goto FINISHED;
00853
00854 // output iteration information if needed
00855 fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
00856
00857 // store initial residual
00858 norms[0] = relres;
00859
00860 /* outer iteration cycle */
00861 while ( iter < MaxIt ) {
00862
00863     rs[0] = r_norm;
00864
00865     t = 1.0 / r_norm;
00866
00867     fasp_blas_darray_ax(n, t, p[0]);
00868
00869     /* RESTART CYCLE (right-preconditioning) */
00870     i = 0;
00871     while ( i < restart && iter < MaxIt ) {
00872
00873         i++; iter++;
00874
00875         /* apply preconditioner */
00876         if ( pc == NULL )
00877             fasp_darray_cp(n, p[i-1], r);
00878         else
00879             pc->fct(p[i-1], r, pc->data);
00880
00881         fasp_blas_dble_mxv(A, r, p[i]);
00882
00883         /* modified Gram_Schmidt */
00884         for ( j = 0; j < i; j++ ) {
00885             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00886             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00887         }
00888         t = fasp_blas_darray_norm2(n, p[i]);
00889         hh[i][i-1] = t;
00890         if ( t != 0.0 ) {
00891             t = 1.0/t;
00892             fasp_blas_darray_ax(n, t, p[i]);
00893         }
00894
00895         for ( j = 1; j < i; ++j ) {
00896             t = hh[j-1][i-1];
00897             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00898             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00899         }
00900         t = hh[i][i-1]*hh[i][i-1];
00901         t += hh[i-1][i-1]*hh[i-1][i-1];
00902
00903         gamma = sqrt(t);
00904         if ( gamma == 0.0 ) gamma = epsmac;
00905         c[i-1] = hh[i-1][i-1] / gamma;
00906         s[i-1] = hh[i][i-1] / gamma;
00907         rs[i] = -s[i-1]*rs[i-1];
00908         rs[i-1] = c[i-1]*rs[i-1];
00909         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00910
00911         absres = r_norm = fabs(rs[i]);
00912
00913         relres = absres/normr0;
00914
00915         norms[iter] = relres;
00916
00917         // output iteration information if needed
00918         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00919                     norms[iter]/norms[iter-1]);
00920

```

```

00921         // should we exit restart cycle
00922         if ( relres <= tol && iter >= MIN_ITER ) break;
00923
00924     } /* end of restart cycle */
00925
00926     /* compute solution, first solve upper triangular system */
00927     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00928     for ( k = i-2; k >= 0; k-- ) {
00929         t = 0.0;
00930         for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
00931
00932         t += rs[k];
00933         rs[k] = t / hh[k][k];
00934     }
00935
00936     fasp_darray_cp(n, p[i-1], w);
00937
00938     fasp_blas_darray_ax(n, rs[i-1], w);
00939
00940     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00941
00942     /* apply preconditioner */
00943     if ( pc == NULL )
00944         fasp_darray_cp(n, w, r);
00945     else
00946         pc->fct(w, r, pc->data);
00947
00948     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00949
00950     // safety net check: save the best-so-far solution
00951     if ( fasp_dvec_isnan(x) ) {
00952         // If the solution is NAN, restore the best solution
00953         absres = BIGREAL;
00954         goto RESTORE_BESTSOL;
00955     }
00956
00957     if ( absres < absres_best - maxdiff ) {
00958         absres_best = absres;
00959         iter_best = iter;
00960         fasp_darray_cp(n, x->val, x_best);
00961     }
00962
00963     // Check: prevent false convergence
00964     if ( relres <= tol && iter >= MIN_ITER ) {
00965
00966         fasp_darray_cp(n, b->val, r);
00967         fasp_blas_dblec_aAxy(-1.0, A, x->val, r);
00968
00969         r_norm = fasp_blas_darray_norm2(n, r);
00970
00971         switch ( StopType ) {
00972             case STOP_REL_RES:
00973                 absres = r_norm;
00974                 relres = absres/normr0;
00975                 break;
00976             case STOP_REL_PRECRES:
00977                 if ( pc == NULL )
00978                     fasp_darray_cp(n, r, w);
00979                 else
00980                     pc->fct(r, w, pc->data);
00981                 absres = sqrt(fasp_blas_darray_dotprod(n, w, r));
00982                 relres = absres/normr0;
00983                 break;
00984             case STOP_MOD_REL_RES:
00985                 absres = r_norm;
00986                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00987                 relres = absres/normu;
00988                 break;
00989         }
00990
00991         norms[iter] = relres;
00992
00993         if ( relres <= tol ) {
00994             break;
00995         }
00996         else {
00997             // Need to restart
00998             fasp_darray_cp(n, r, p[0]); i = 0;
00999         }
01000     } /* end of convergence check */

```

```

01002
01003     /* compute residual vector and continue loop */
01004     for ( j = i; j > 0; j-- ) {
01005         rs[j-1] = -s[j-1]*rs[j];
01006         rs[j] = c[j-1]*rs[j];
01007     }
01008
01009     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01010
01011     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01012
01013     if ( i ) {
01014         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01015         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01016     }
01017
01018     } /* end of main while loop */
01019
01020 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01021     if ( iter != iter_best ) {
01022
01023         // compute best residual
01024         fasp_darray_cp(n,b->val,r);
01025         fasp_blas_dblc_aAxy(-1.0,A,x_best,r);
01026
01027         switch ( StopType ) {
01028             case STOP_REL_RES:
01029                 absres_best = fasp_blas_darray_norm2(n,r);
01030                 break;
01031             case STOP_REL_PRECRES:
01032                 // z = B(r)
01033                 if ( pc != NULL )
01034                     pc->fct(r,w,pc->data); /* Apply preconditioner */
01035                 else
01036                     fasp_darray_cp(n,r,w); /* No preconditioner */
01037                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
01038                 break;
01039             case STOP_MOD_REL_RES:
01040                 absres_best = fasp_blas_darray_norm2(n,r);
01041                 break;
01042         }
01043
01044         if ( absres > absres_best + maxdiff || isnan(absres) ) {
01045             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01046             fasp_darray_cp(n,x_best,x->val);
01047             relres = absres_best / normr0;
01048         }
01049     }
01050
01051 FINISHED:
01052     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01053
01054     /*-----
01055  * Clean up workspace
01056  *-----*/
01057     fasp_mem_free(work); work = NULL;
01058     fasp_mem_free(p); p = NULL;
01059     fasp_mem_free(hh); hh = NULL;
01060     fasp_mem_free(norms); norms = NULL;
01061
01062 #if DEBUG_MODE > 0
01063     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01064 #endif
01065
01066     if ( iter >= MaxIt )
01067         return ERROR_SOLVER_MAXIT;
01068     else
01069         return iter;
01070 }
01071
01095 INT fasp_solver_dstr_spgmres (const dSTRmat *A,
01096                             const dvector *b,
01097                             dvector *x,
01098                             precondition *pc,
01099                             const REAL tol,
01100                             const INT MaxIt,
01101                             SHORT restart,
01102                             const SHORT StopType,
01103                             const SHORT PrtLvl)
01104 {
01105     const INT n = b->row;

```

```

01106     const INT  MIN_ITER   = 0;
01107     const REAL maxdiff    = tol*STAG_RATIO; // staganation tolerance
01108     const REAL epsmac     = SMALLREAL;
01109
01110     // local variables
01111     INT      iter = 0;
01112     INT      restartl = restart + 1;
01113     int      i, j, k; // must be signed!  -zcs
01114
01115     REAL      r_norm, r_normb, gamma, t;
01116     REAL      normr0 = BIGREAL, absres = BIGREAL;
01117     REAL      relres = BIGREAL, normu = BIGREAL;
01118
01119     INT      iter_best = 0; // initial best known iteration
01120     REAL      absres_best = BIGREAL; // initial best known residual
01121
01122     // allocate temp memory (need about (restart+4)*n REAL numbers)
01123     REAL      *c = NULL, *s = NULL, *rs = NULL;
01124     REAL      *norms = NULL, *r = NULL, *w = NULL;
01125     REAL      *work = NULL, *x_best = NULL;
01126     REAL      **p = NULL, **hh = NULL;
01127
01128     // Output some info for debugging
01129     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (STR) ...\n");
01130
01131     #if DEBUG_MODE > 0
01132     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01133     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01134     #endif
01135
01136     /* allocate memory and setup temp work space */
01137     work = (REAL *) fasp_mem_malloc((restart+4)*(restart+n)+1, sizeof(REAL));
01138
01139     /* check whether memory is enough for GMRES */
01140     while ( (work == NULL) && (restart > 5) ) {
01141         restart = restart - 5 ;
01142         work = (REAL *) fasp_mem_malloc((restart+4)*(restart+n)+1, sizeof(REAL));
01143         printf("### WARNING: GMRES restart number set to %d!\n", restart);
01144         restartl = restart + 1;
01145     }
01146
01147     if ( work == NULL ) {
01148         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
01149         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01150     }
01151
01152     p = (REAL **)fasp_mem_malloc(restartl, sizeof(REAL *));
01153     hh = (REAL **)fasp_mem_malloc(restartl, sizeof(REAL *));
01154     norms = (REAL *) fasp_mem_malloc(MaxIt+1, sizeof(REAL));
01155
01156     r = work; w = r + n; rs = w + n; c = rs + restartl;
01157     x_best = c + restart; s = x_best + n;
01158
01159     for ( i = 0; i < restartl; i++ ) p[i] = s + restart + i*n;
01160
01161     for ( i = 0; i < restartl; i++ ) hh[i] = p[restart] + n + i*restart;
01162
01163     // r = b-A*x
01164     fasp_darray_cp(n, b->val, p[0]);
01165     fasp_blas_dstr_aAxy(-1.0, A, x->val, p[0]);
01166
01167     r_norm = fasp_blas_darray_norm2(n,p[0]);
01168
01169     // compute initial residuals
01170     switch (StopType) {
01171     case STOP_REL_RES:
01172         normr0 = MAX(SMALLREAL, r_norm);
01173         relres = r_norm/normr0;
01174         break;
01175     case STOP_REL_PRECRES:
01176         if ( pc == NULL )
01177             fasp_darray_cp(n, p[0], r);
01178         else
01179             pc->fct(p[0], r, pc->data);
01180         r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
01181         normr0 = MAX(SMALLREAL, r_normb);
01182         relres = r_normb/normr0;
01183         break;
01184     case STOP_MOD_REL_RES:
01185         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n,x->val));
01186         normr0 = r_norm;

```

```

01187         relres = normr0/normu;
01188         break;
01189     default:
01190         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01191         goto FINISHED;
01192     }
01193
01194     // if initial residual is small, no need to iterate!
01195     if ( relres < tol ) goto FINISHED;
01196
01197     // output iteration information if needed
01198     fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
01199
01200     // store initial residual
01201     norms[0] = relres;
01202
01203     /* outer iteration cycle */
01204     while ( iter < MaxIt ) {
01205
01206         rs[0] = r_norm;
01207
01208         t = 1.0 / r_norm;
01209
01210         fasp_blas_darray_ax(n, t, p[0]);
01211
01212         /* RESTART CYCLE (right-preconditioning) */
01213         i = 0;
01214         while ( i < restart && iter < MaxIt ) {
01215
01216             i++; iter++;
01217
01218             /* apply preconditioner */
01219             if ( pc == NULL )
01220                 fasp_darray_cp(n, p[i-1], r);
01221             else
01222                 pc->fct(p[i-1], r, pc->data);
01223
01224             fasp_blas_dstr_mxdv(A, r, p[i]);
01225
01226             /* modified Gram_Schmidt */
01227             for ( j = 0; j < i; j++ ) {
01228                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01229                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01230             }
01231             t = fasp_blas_darray_norm2(n, p[i]);
01232             hh[i][i-1] = t;
01233             if ( t != 0.0 ) {
01234                 t = 1.0/t;
01235                 fasp_blas_darray_ax(n, t, p[i]);
01236             }
01237
01238             for ( j = 1; j < i; j++ ) {
01239                 t = hh[j-1][i-1];
01240                 hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01241                 hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01242             }
01243             t = hh[i][i-1]*hh[i][i-1];
01244             t += hh[i-1][i-1]*hh[i-1][i-1];
01245
01246             gamma = sqrt(t);
01247             if ( gamma == 0.0 ) gamma = epsmac;
01248             c[i-1] = hh[i-1][i-1] / gamma;
01249             s[i-1] = hh[i][i-1] / gamma;
01250             rs[i] = -s[i-1]*rs[i-1];
01251             rs[i-1] = c[i-1]*rs[i-1];
01252             hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01253
01254             absres = r_norm = fabs(rs[i]);
01255
01256             relres = absres/normr0;
01257
01258             norms[iter] = relres;
01259
01260             // output iteration information if needed
01261             fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
01262                 norms[iter]/norms[iter-1]);
01263
01264             // should we exit restart cycle
01265             if ( relres <= tol && iter >= MIN_ITER ) break;
01266
01267         } /* end of restart cycle */

```

```

01268
01269      /* compute solution, first solve upper triangular system */
01270      rs[i-1] = rs[i-1] / hh[i-1][i-1];
01271      for ( k = i-2; k >= 0; k-- ) {
01272          t = 0.0;
01273          for ( j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
01274          t += rs[k];
01275          rs[k] = t / hh[k][k];
01276      }
01277
01278      fasp_darray_cp(n, p[i-1], w);
01279
01280      fasp_blas_darray_ax(n, rs[i-1], w);
01281
01282      for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01283
01284      /* apply preconditioner */
01285      if ( pc == NULL )
01286          fasp_darray_cp(n, w, r);
01287      else
01288          pc->fct(w, r, pc->data);
01289
01290      fasp_blas_darray_axpy(n, 1.0, r, x->val);
01291
01292      // safety net check: save the best-so-far solution
01293      if ( fasp_dvec_isnan(x) ) {
01294          // If the solution is NAN, restore the best solution
01295          absres = BIGREAL;
01296          goto RESTORE_BESTSOL;
01297      }
01298
01299      if ( absres < absres_best - maxdiff ) {
01300          absres_best = absres;
01301          iter_best = iter;
01302          fasp_darray_cp(n, x->val, x_best);
01303      }
01304
01305      // Check: prevent false convergence
01306      if ( relres <= tol && iter >= MIN_ITER ) {
01307          fasp_darray_cp(n, b->val, r);
01308          fasp_blas_dstr_aAxp(-1.0, A, x->val, r);
01309
01310          r_norm = fasp_blas_darray_norm2(n, r);
01311
01312          switch ( StopType ) {
01313              case STOP_REL_RES:
01314                  absres = r_norm;
01315                  relres = absres/normr0;
01316                  break;
01317              case STOP_REL_PRECRES:
01318                  if ( pc == NULL )
01319                      fasp_darray_cp(n, r, w);
01320                  else
01321                      pc->fct(r, w, pc->data);
01322                  absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
01323                  relres = absres/normr0;
01324                  break;
01325              case STOP_MOD_REL_RES:
01326                  absres = r_norm;
01327                  normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
01328                  relres = absres/normu;
01329                  break;
01330          }
01331
01332          norms[iter] = relres;
01333
01334          if ( relres <= tol ) {
01335              break;
01336          }
01337          else {
01338              // Need to restart
01339              fasp_darray_cp(n, r, p[0]); i = 0;
01340          }
01341      } /* end of convergence check */
01342
01343      /* compute residual vector and continue loop */
01344      for ( j = i; j > 0; j-- ) {
01345          rs[j-1] = -s[j-1]*rs[j];
01346      }

```



```

01349         rs[j] = c[j-1]*rs[j];
01350     }
01351
01352     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01353
01354     for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01355
01356     if ( i ) {
01357         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01358         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01359     }
01360
01361 } /* end of main while loop */
01362
01363 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01364 if ( iter != iter_best ) {
01365
01366     // compute best residual
01367     fasp_darray_cp(n,b->val,r);
01368     fasp_blas_dstr_aAxy(-1.0,A,x_best,r);
01369
01370     switch ( StopType ) {
01371     case STOP_REL_RES:
01372         absres_best = fasp_blas_darray_norm2(n,r);
01373         break;
01374     case STOP_REL_PRECRES:
01375         // z = B(r)
01376         if ( pc != NULL )
01377             pc->fct(r,w,pc->data); /* Apply preconditioner */
01378         else
01379             fasp_darray_cp(n,r,w); /* No preconditioner */
01380         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
01381         break;
01382     case STOP_MOD_REL_RES:
01383         absres_best = fasp_blas_darray_norm2(n,r);
01384         break;
01385     }
01386
01387     if ( absres > absres_best + maxdiff || isnan(absres) ) {
01388         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01389         fasp_darray_cp(n,x_best,x->val);
01390         relres = absres_best / normr0;
01391     }
01392 }
01393
01394 FINISHED:
01395 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01396
01397 /*-----
01398 * Clean up workspace
01399 *-----*/
01400 fasp_mem_free(work); work = NULL;
01401 fasp_mem_free(p); p = NULL;
01402 fasp_mem_free(hh); hh = NULL;
01403 fasp_mem_free(norms); norms = NULL;
01404
01405 #if DEBUG_MODE > 0
01406 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01407 #endif
01408
01409 if ( iter >= MaxIt )
01410     return ERROR_SOLVER_MAXIT;
01411 else
01412     return iter;
01413 }
01414
01415 /*-----*/
01416 /*-- End of File --*/
01417 /*-----*/

```

9.131 KrySPminres.c File Reference

Krylov subspace methods – Preconditioned MINRES with safety net.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"

```

```
#include "KryUtil.inl"
```

Functions

- [INT fasp_solver_dcsr_spmminres](#) (const [dCSRmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.
- [INT fasp_solver_dblc_spmminres](#) (const [dBLCmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.
- [INT fasp_solver_dstr_spmminres](#) (const [dSTRmat](#) *A, const [dvector](#) *b, [dvector](#) *u, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.

9.131.1 Detailed Description

Krylov subspace methods – Preconditioned MINRES with safety net.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

The ‘best’ iterative solution will be saved and used upon exit; See [KryPminres.c](#) for a version without safety net

Reference: Y. Saad 2003 Iterative methods for sparse linear systems (2nd Edition), SIAM
Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KrySPminres.c](#).

9.131.2 Function Documentation

9.131.2.1 fasp_solver_dblc_spmminres()

```
INT fasp_solver_dblc_spmminres (
    const dBLCmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dBLCmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns

Parameters

<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/09/2013

Definition at line 511 of file [KrySPminres.c](#).

9.131.2.2 fasp_solver_dcsr_spminres()

```

INT fasp_solver_dcsr_spminres (
    const dCSRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/09/2013

Definition at line 60 of file [KrySPminres.c](#).**9.131.2.3 fasp_solver_dstr_spminres()**

```

INT fasp_solver_dstr_spminres (
    const dSTRmat * A,
    const dvector * b,
    dvector * u,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    const SHORT StopType,
    const SHORT PrtLvl )

```

A preconditioned minimal residual (Minres) method for solving $Au=b$ with safety net.**Parameters**

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>u</i>	Pointer to dvector: unknowns
<i>MaxIt</i>	Maximal number of iterations
<i>tol</i>	Tolerance for stopping
<i>pc</i>	Pointer to structure of precondition (precond)
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/09/2013

Definition at line 962 of file [KrySPminres.c](#).**9.132 KrySPminres.c**[Go to the documentation of this file.](#)

```

00001
00024 #include <math.h>
00025

```

```

00026 #include "fasp.h"
00027 #include "fasp_functs.h"
00028
00029 /*-----*/
00030 /*--  Declare Private Functions  --*/
00031 /*-----*/
00032
00033 #include "KryUtil.inl"
00034
00035 /*-----*/
00036 /*--      Public Functions      --*/
00037 /*-----*/
00038
00060 INT fasp_solver_dcsr_spminres (const dCSRmat *A,
00061                               const dvector *b,
00062                               dvector *u,
00063                               precondition *pc,
00064                               const REAL tol,
00065                               const INT MaxIt,
00066                               const SHORT StopType,
00067                               const SHORT PrtLvl)
00068 {
00069     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00070     const INT m = b->row;
00071     const REAL maxdiff = tol*STAG_RATIO; // stagnation tolerance
00072     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00073
00074     // local variables
00075     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00076     REAL absres0 = BIGREAL, absres = BIGREAL;
00077     REAL normr0 = BIGREAL, relres = BIGREAL;
00078     REAL normu2, normu, normp, normuinf, factor;
00079     REAL alpha, alpha0, alpha1, temp2;
00080     INT iter_best = 0; // initial best known iteration
00081     REAL absres_best = BIGREAL; // initial best known residual
00082
00083     // allocate temp memory (need 12*m REAL)
00084     REAL *work=(REAL *)fasp_mem_calloc(12*m,sizeof(REAL));
00085     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
00086     REAL *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m, *u_best = r+m;
00087
00088     // Output some info for debugging
00089     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (CSR) ...\n");
00090
00091     #if DEBUG_MODE > 0
00092         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00093         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00094     #endif
00095
00096     // p0 = 0
00097     fasp_darray_set(m,p0,0.0);
00098
00099     // r = b-A*u
00100     fasp_darray_cp(m,b->val,r);
00101     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00102
00103     // p1 = B(r)
00104     if ( pc != NULL )
00105         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00106     else
00107         fasp_darray_cp(m,r,p1); /* No preconditioner */
00108
00109     // compute initial residuals
00110     switch ( StopType ) {
00111     case STOP_REL_RES:
00112         absres0 = fasp_blas_darray_norm2(m,r);
00113         normr0 = MAX(SMALLREAL,absres0);
00114         relres = absres0/normr0;
00115         break;
00116     case STOP_REL_PRECRES:
00117         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
00118         normr0 = MAX(SMALLREAL,absres0);
00119         relres = absres0/normr0;
00120         break;
00121     case STOP_MOD_REL_RES:
00122         absres0 = fasp_blas_darray_norm2(m,r);
00123         normu2 = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00124         relres = absres0/normu2;
00125         break;
00126     default:
00127         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);

```

```

00128         goto FINISHED;
00129     }
00130
00131     // if initial residual is small, no need to iterate!
00132     if ( relres < tol ) goto FINISHED;
00133
00134     // output iteration information if needed
00135     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00136
00137     // tp = A*p1
00138     fasp_blas_dcsr_mxv(A,p1,tp);
00139
00140     // tz = B(tp)
00141     if ( pc != NULL )
00142         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00143     else
00144         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00145
00146     // p1 = p1/normp
00147     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00148     normp = sqrt(normp);
00149     fasp_darray_cp(m,p1,t);
00150     fasp_darray_set(m,p1,0.0);
00151     fasp_blas_darray_axpy(m,1/normp,t,p1);
00152
00153     // t0 = A*p0 = 0
00154     fasp_darray_set(m,t0,0.0);
00155     fasp_darray_cp(m,t0,z0);
00156     fasp_darray_cp(m,t0,t1);
00157     fasp_darray_cp(m,t0,z1);
00158
00159     // t1 = tp/normp, z1 = tz/normp
00160     fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
00161     fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
00162
00163     // main MinRes loop
00164     while ( iter++ < MaxIt ) {
00165
00166         // alpha = <r,z1>
00167         alpha=fasp_blas_darray_dotprod(m,r,z1);
00168
00169         // u = u+alpha*p1
00170         fasp_blas_darray_axpy(m,alpha,p1,u->val);
00171
00172         // r = r-alpha*A*p1
00173         fasp_blas_darray_axpy(m,-alpha,t1,r);
00174
00175         // compute t = A*z1 alpha1 = <z1,t>
00176         fasp_blas_dcsr_mxv(A,z1,t);
00177         alpha1=fasp_blas_darray_dotprod(m,z1,t);
00178
00179         // compute t = A*z0 alpha0 = <z1,t>
00180         fasp_blas_dcsr_mxv(A,z0,t);
00181         alpha0=fasp_blas_darray_dotprod(m,z1,t);
00182
00183         // p2 = z1-alpha1*p1-alpha0*p0
00184         fasp_darray_cp(m,z1,p2);
00185         fasp_blas_darray_axpy(m,-alpha1,p1,p2);
00186         fasp_blas_darray_axpy(m,-alpha0,p0,p2);
00187
00188         // tp = A*p2
00189         fasp_blas_dcsr_mxv(A,p2,tp);
00190
00191         // tz = B(tp)
00192         if ( pc != NULL )
00193             pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00194         else
00195             fasp_darray_cp(m,tp,tz); /* No preconditioner */
00196
00197         // p2 = p2/normp
00198         normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00199         normp = sqrt(normp);
00200         fasp_darray_cp(m,p2,t);
00201         fasp_darray_set(m,p2,0.0);
00202         fasp_blas_darray_axpy(m,1/normp,t,p2);
00203
00204         // prepare for next iteration
00205         fasp_darray_cp(m,p1,p0);
00206         fasp_darray_cp(m,p2,p1);
00207         fasp_darray_cp(m,t1,t0);
00208         fasp_darray_cp(m,z1,z0);

```

```

00209
00210 // t1=tp/normp, z1=tz/normp
00211 fasp_darray_set(m,t1,0.0);
00212 fasp_darray_cp(m,t1,z1);
00213 fasp_blas_darray_axpy(m,1/normp,tp,t1);
00214 fasp_blas_darray_axpy(m,1/normp,tz,z1);
00215
00216 normu2 = fasp_blas_darray_norm2(m,u->val);
00217
00218 // compute residuals
00219 switch ( StopType ) {
00220     case STOP_REL_RES:
00221         temp2 = fasp_blas_darray_dotprod(m,r,r);
00222         absres = sqrt(temp2);
00223         relres = absres/normr0;
00224         break;
00225     case STOP_REL_PRECRES:
00226         if (pc == NULL)
00227             fasp_darray_cp(m,r,t);
00228         else
00229             pc->fct(r,t,pc->data);
00230         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00231         absres = sqrt(temp2);
00232         relres = absres/normr0;
00233         break;
00234     case STOP_MOD_REL_RES:
00235         temp2 = fasp_blas_darray_dotprod(m,r,r);
00236         absres = sqrt(temp2);
00237         relres = absres/normu2;
00238         break;
00239 }
00240
00241 // compute reduction factor of residual ||r||
00242 factor = absres/absres0;
00243
00244 // output iteration information if needed
00245 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00246
00247 // safety net check: save the best-so-far solution
00248 if ( fasp_dvec_isnan(u) ) {
00249     // If the solution is NAN, restore the best solution
00250     absres = BIGREAL;
00251     goto RESTORE_BESTSOL;
00252 }
00253
00254 if ( absres < absres_best - maxdiff ) {
00255     absres_best = absres;
00256     iter_best = iter;
00257     fasp_darray_cp(m,u->val,u_best);
00258 }
00259
00260 // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00261 normuinf = fasp_blas_darray_norminf(m, u->val);
00262 if (normuinf <= sol_inf_tol) {
00263     if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00264     iter = ERROR_SOLVER_SOLSTAG;
00265     break;
00266 }
00267
00268 // Check II: if staggenated, try to restart
00269 normuu = fasp_blas_darray_norm2(m,p1);
00270 normuu = ABS(alpha)*(normuu/normu2);
00271
00272 if ( normuu < maxdiff ) {
00273     if ( stag < MaxStag ) {
00274         if ( PrtLvl >= PRINT_MORE ) {
00275             ITS_DIFFRES(normuu,relres);
00276             ITS_RESTART;
00277         }
00278     }
00279 }
00280
00281 fasp_darray_cp(m,b->val,r);
00282 fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00283
00284 // compute residuals
00285 switch (StopType) {
00286     case STOP_REL_RES:
00287         temp2 = fasp_blas_darray_dotprod(m,r,r);
00288         absres = sqrt(temp2);
00289         relres = absres/normr0;

```

```

00290         break;
00291     case STOP_REL_PRECRES:
00292         if (pc == NULL)
00293             fasp_darray_cp(m,r,t);
00294         else
00295             pc->fct(r,t,pc->data);
00296         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00297         absres = sqrt(temp2);
00298         relres = absres/normr0;
00299         break;
00300     case STOP_MOD_REL_RES:
00301         temp2 = fasp_blas_darray_dotprod(m,r,r);
00302         absres = sqrt(temp2);
00303         relres = absres/normu2;
00304         break;
00305     }
00306
00307     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00308
00309     if ( relres < tol )
00310         break;
00311     else {
00312         if ( stag >= MaxStag ) {
00313             if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00314             iter = ERROR_SOLVER_STAG;
00315             break;
00316         }
00317         fasp_darray_set(m,p0,0.0);
00318         ++stag;
00319         ++restart_step;
00320
00321         // p1 = B(r)
00322         if ( pc != NULL )
00323             pc->fct(r,p1,pc->data); /* Apply preconditioner */
00324         else
00325             fasp_darray_cp(m,r,p1); /* No preconditioner */
00326
00327         // tp = A*p1
00328         fasp_blas_dcsr_mnv(A,p1,tp);
00329
00330         // tz = B(tp)
00331         if ( pc != NULL )
00332             pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
00333         else
00334             fasp_darray_cp(m,tp,tz); /* No preconditioner */
00335
00336         // p1 = tp/normp
00337         normp = fasp_blas_darray_dotprod(m,tz,tp);
00338         normp = sqrt(normp);
00339         fasp_darray_cp(m,p1,t);
00340
00341         // t0 = A*p0=0
00342         fasp_darray_set(m,t0,0.0);
00343         fasp_darray_cp(m,t0,z0);
00344         fasp_darray_cp(m,t0,t1);
00345         fasp_darray_cp(m,t0,z1);
00346         fasp_darray_cp(m,t0,p1);
00347
00348         fasp_blas_darray_axpy(m,1/normp,t,p1);
00349
00350         // t1 = tp/normp, z1 = tz/normp
00351         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00352         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00353     }
00354 }
00355
00356 // Check III: prevent false convergence
00357 if ( relres < tol ) {
00358
00359     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00360
00361     // compute residual r = b - Ax again
00362     fasp_darray_cp(m,b->val,r);
00363     fasp_blas_dcsr_aAxy(-1.0,A,u->val,r);
00364
00365     // compute residuals
00366     switch (StopType) {
00367         case STOP_REL_RES:
00368             temp2 = fasp_blas_darray_dotprod(m,r,r);
00369             absres = sqrt(temp2);
00370             relres = absres/normr0;

```



```

00371         break;
00372     case STOP_REL_PRECRES:
00373         if (pc == NULL)
00374             fasp_darray_cp(m,r,t);
00375         else
00376             pc->fct(r,t,pc->data);
00377         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00378         absres = sqrt(temp2);
00379         relres = absres/normr0;
00380         break;
00381     case STOP_MOD_REL_RES:
00382         temp2 = fasp_blas_darray_dotprod(m,r,r);
00383         absres = sqrt(temp2);
00384         relres = absres/normu2;
00385         break;
00386     }
00387
00388     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00389
00390     // check convergence
00391     if ( relres < tol ) break;
00392
00393     if ( more_step >= MaxRestartStep ) {
00394         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00395         iter = ERROR_SOLVER_TOLSMALL;
00396         break;
00397     }
00398
00399     // prepare for restarting method
00400     fasp_darray_set(m,p0,0.0);
00401     ++more_step;
00402     ++restart_step;
00403
00404     // p1 = B(r)
00405     if ( pc != NULL )
00406         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00407     else
00408         fasp_darray_cp(m,r,p1); /* No preconditioner */
00409
00410     // tp = A*p1
00411     fasp_blas_dcsr_mxv(A,p1,tp);
00412
00413     // tz = B(tp)
00414     if ( pc != NULL )
00415         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00416     else
00417         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00418
00419     // p1 = tp/normp
00420     normp = fasp_blas_darray_dotprod(m,tz,tp);
00421     normp = sqrt(normp);
00422     fasp_darray_cp(m,p1,t);
00423
00424     // t0 = A*p0 = 0
00425     fasp_darray_set(m,t0,0.0);
00426     fasp_darray_cp(m,t0,z0);
00427     fasp_darray_cp(m,t0,t1);
00428     fasp_darray_cp(m,t0,z1);
00429     fasp_darray_cp(m,t0,p1);
00430
00431     fasp_blas_darray_axpy(m,1/normp,t,p1);
00432
00433     // t1=tp/normp,z1=tz/normp
00434     fasp_blas_darray_axpy(m,1/normp,tp,t1);
00435     fasp_blas_darray_axpy(m,1/normp,tz,z1);
00436
00437 } // end of convergence check
00438
00439 // update relative residual here
00440 absres0 = absres;
00441
00442 } // end of the main loop
00443
00444 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00445     if ( iter != iter_best ) {
00446
00447         // compute best residual
00448         fasp_darray_cp(m,b->val,r);
00449         fasp_blas_dcsr_aAxy(-1.0,A,u_best,r);
00450
00451         switch ( StopType ) {

```

```

00452         case STOP_REL_RES:
00453             absres_best = fasp_blas_darray_norm2(m,r);
00454             break;
00455         case STOP_REL_PRECRES:
00456             if ( pc != NULL )
00457                 pc->fct(r,t,pc->data); /* Apply preconditioner */
00458             else
00459                 fasp_darray_cp(m,r,t); /* No preconditioner */
00460             absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
00461             break;
00462         case STOP_MOD_REL_RES:
00463             absres_best = fasp_blas_darray_norm2(m,r);
00464             break;
00465     }
00466
00467     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00468         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00469         fasp_darray_cp(m,u_best,u->val);
00470         relres = absres_best / normr0;
00471     }
00472 }
00473
00474 FINISHED: // finish iterative method
00475 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00476
00477 // clean up temp memory
00478 fasp_mem_free(work); work = NULL;
00479
00480 #if DEBUG_MODE > 0
00481 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00482 #endif
00483
00484 if ( iter > MaxIt )
00485     return ERROR_SOLVER_MAXIT;
00486 else
00487     return iter;
00488 }
00489
00511 INT fasp_solver_dblc_spmminres (const dBLCmat *A,
00512                                const dvector *b,
00513                                dvector *u,
00514                                precondition *pc,
00515                                const REAL tol,
00516                                const INT MaxIt,
00517                                const SHORT StopType,
00518                                const SHORT PrtLvl)
00519 {
00520     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00521     const INT m = b->row;
00522     const REAL maxdiff = tol*STAG_RATIO; // stagantation tolerance
00523     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00524
00525     // local variables
00526     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00527     REAL absres0 = BIGREAL, absres = BIGREAL;
00528     REAL normr0 = BIGREAL, relres = BIGREAL;
00529     REAL normu2, normuu, normp, normuinf, factor;
00530     REAL alpha, alpha0, alphas, temp2;
00531     INT iter_best = 0; // initial best known iteration
00532     REAL absres_best = BIGREAL; // initial best known residual
00533
00534     // allocate temp memory (need 12*m REAL)
00535     REAL *work=(REAL *)fasp_mem_malloc(12*m,sizeof(REAL));
00536     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
00537     REAL *t1=t0+m, *t2=t1+m, *tp=t2+m, *tz=tp+m, *r=tz+m, *u_best = r+m;
00538
00539     // Output some info for debugging
00540     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (BLC) ...\n");
00541
00542     #if DEBUG_MODE > 0
00543         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00544         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00545     #endif
00546
00547     // p0 = 0
00548     fasp_darray_set(m,p0,0.0);
00549
00550     // r = b-A*u
00551     fasp_darray_cp(m,b->val,r);
00552     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00553

```

```

00554 // p1 = B(r)
00555 if ( pc != NULL )
00556     pc->fct(r,p1,pc->data); /* Apply preconditioner */
00557 else
00558     fasp_darray_cp(m,r,p1); /* No preconditioner */
00559
00560 // compute initial residuals
00561 switch ( StopType ) {
00562     case STOP_REL_RES:
00563         absres0 = fasp_blas_darray_norm2(m,r);
00564         normr0 = MAX(SMALLREAL,absres0);
00565         relres = absres0/normr0;
00566         break;
00567     case STOP_REL_PRECRES:
00568         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
00569         normr0 = MAX(SMALLREAL,absres0);
00570         relres = absres0/normr0;
00571         break;
00572     case STOP_MOD_REL_RES:
00573         absres0 = fasp_blas_darray_norm2(m,r);
00574         normu2 = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
00575         relres = absres0/normu2;
00576         break;
00577     default:
00578         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00579         goto FINISHED;
00580 }
00581
00582 // if initial residual is small, no need to iterate!
00583 if ( relres < tol ) goto FINISHED;
00584
00585 // output iteration information if needed
00586 fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
00587
00588 // tp = A*p1
00589 fasp_blas_dblc_mxv(A,p1,tp);
00590
00591 // tz = B(tp)
00592 if ( pc != NULL )
00593     pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00594 else
00595     fasp_darray_cp(m,tp,tz); /* No preconditioner */
00596
00597 // p1 = p1/normp
00598 normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00599 normp = sqrt(normp);
00600 fasp_darray_cp(m,p1,t);
00601 fasp_darray_set(m,p1,0.0);
00602 fasp_blas_darray_axpy(m,1/normp,t,p1);
00603
00604 // t0 = A*p0 = 0
00605 fasp_darray_set(m,t0,0.0);
00606 fasp_darray_cp(m,t0,z0);
00607 fasp_darray_cp(m,t0,t1);
00608 fasp_darray_cp(m,t0,z1);
00609
00610 // t1 = tp/normp, z1 = tz/normp
00611 fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
00612 fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
00613
00614 // main MinRes loop
00615 while ( iter++ < MaxIt ) {
00616
00617     // alpha = <r,z1>
00618     alpha=fasp_blas_darray_dotprod(m,r,z1);
00619
00620     // u = u+alpha*p1
00621     fasp_blas_darray_axpy(m,alpha,p1,u->val);
00622
00623     // r = r-alpha*Apl
00624     fasp_blas_darray_axpy(m,-alpha,t1,r);
00625
00626     // compute t = A*z1 alpha1 = <z1,t>
00627     fasp_blas_dblc_mxv(A,z1,t);
00628     alpha1=fasp_blas_darray_dotprod(m,z1,t);
00629
00630     // compute t = A*z0 alpha0 = <z1,t>
00631     fasp_blas_dblc_mxv(A,z0,t);
00632     alpha0=fasp_blas_darray_dotprod(m,z1,t);
00633
00634     // p2 = z1-alpha1*p1-alpha0*p0

```

```

00635     fasp_darray_cp(m,z1,p2);
00636     fasp_blas_darray_axpy(m,-alpha1,p1,p2);
00637     fasp_blas_darray_axpy(m,-alpha0,p0,p2);
00638
00639     // tp = A*p2
00640     fasp_blas_dblc_mxv(A,p2,tp);
00641
00642     // tz = B(tp)
00643     if ( pc != NULL )
00644         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00645     else
00646         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00647
00648     // p2 = p2/normp
00649     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
00650     normp = sqrt(normp);
00651     fasp_darray_cp(m,p2,t);
00652     fasp_darray_set(m,p2,0.0);
00653     fasp_blas_darray_axpy(m,1/normp,t,p2);
00654
00655     // prepare for next iteration
00656     fasp_darray_cp(m,p1,p0);
00657     fasp_darray_cp(m,p2,p1);
00658     fasp_darray_cp(m,t1,t0);
00659     fasp_darray_cp(m,z1,z0);
00660
00661     // t1=tp/normp, z1=tz/normp
00662     fasp_darray_set(m,t1,0.0);
00663     fasp_darray_cp(m,t1,z1);
00664     fasp_blas_darray_axpy(m,1/normp,tp,t1);
00665     fasp_blas_darray_axpy(m,1/normp,tz,z1);
00666
00667     normu2 = fasp_blas_darray_norm2(m,u->val);
00668
00669     // compute residuals
00670     switch ( StopType ) {
00671         case STOP_REL_RES:
00672             temp2 = fasp_blas_darray_dotprod(m,r,r);
00673             absres = sqrt(temp2);
00674             relres = absres/normr0;
00675             break;
00676         case STOP_REL_PRECRES:
00677             if (pc == NULL)
00678                 fasp_darray_cp(m,r,t);
00679             else
00680                 pc->fct(r,t,pc->data);
00681             temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00682             absres = sqrt(temp2);
00683             relres = absres/normr0;
00684             break;
00685         case STOP_MOD_REL_RES:
00686             temp2 = fasp_blas_darray_dotprod(m,r,r);
00687             absres = sqrt(temp2);
00688             relres = absres/normu2;
00689             break;
00690     }
00691
00692     // compute reduction factor of residual ||r||
00693     factor = absres/absres0;
00694
00695     // output iteration information if needed
00696     fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
00697
00698     // safety net check: save the best-so-far solution
00699     if ( fasp_dvec_isnan(u) ) {
00700         // If the solution is NAN, restore the best solution
00701         absres = BIGREAL;
00702         goto RESTORE_BESTSOL;
00703     }
00704
00705     if ( absres < absres_best - maxdiff ) {
00706         absres_best = absres;
00707         iter_best = iter;
00708         fasp_darray_cp(m,u->val,u_best);
00709     }
00710
00711     // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
00712     normuinf = fasp_blas_darray_norminf(m, u->val);
00713     if (normuinf <= sol_inf_tol) {
00714         if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
00715         iter = ERROR_SOLVER_SOLSTAG;

```

```

00716         break;
00717     }
00718
00719     // Check II: if staggenated, try to restart
00720     normuu = fasp_blas_darray_norm2(m,p1);
00721     normuu = ABS(alpha)*(normuu/normu2);
00722
00723     if ( normuu < maxdiff ) {
00724
00725         if ( stag < MaxStag ) {
00726             if ( PrtLvl >= PRINT_MORE ) {
00727                 ITS_DIFFRES(normuu,relres);
00728                 ITS_RESTART;
00729             }
00730         }
00731
00732         fasp_darray_cp(m,b->val,r);
00733         fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00734
00735         // compute residuals
00736         switch (StopType) {
00737             case STOP_REL_RES:
00738                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00739                 absres = sqrt(temp2);
00740                 relres = absres/normr0;
00741                 break;
00742             case STOP_REL_PRECRES:
00743                 if (pc == NULL)
00744                     fasp_darray_cp(m,r,t);
00745                 else
00746                     pc->fct(r,t,pc->data);
00747                 temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00748                 absres = sqrt(temp2);
00749                 relres = absres/normr0;
00750                 break;
00751             case STOP_MOD_REL_RES:
00752                 temp2 = fasp_blas_darray_dotprod(m,r,r);
00753                 absres = sqrt(temp2);
00754                 relres = absres/normu2;
00755                 break;
00756         }
00757
00758         if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00759
00760         if ( relres < tol )
00761             break;
00762         else {
00763             if ( stag >= MaxStag ) {
00764                 if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
00765                 iter = ERROR_SOLVER_STAG;
00766                 break;
00767             }
00768             fasp_darray_set(m,p0,0.0);
00769             ++stag;
00770             ++restart_step;
00771
00772             // p1 = B(r)
00773             if ( pc != NULL )
00774                 pc->fct(r,p1,pc->data); /* Apply preconditioner */
00775             else
00776                 fasp_darray_cp(m,r,p1); /* No preconditioner */
00777
00778             // tp = A*p1
00779             fasp_blas_dblc_mxy(A,p1,tp);
00780
00781             // tz = B(tp)
00782             if ( pc != NULL )
00783                 pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00784             else
00785                 fasp_darray_cp(m,tp,tz); /* No preconditioner */
00786
00787             // p1 = p1/normp
00788             normp = fasp_blas_darray_dotprod(m,tz,tp);
00789             normp = sqrt(normp);
00790             fasp_darray_cp(m,p1,t);
00791
00792             // t0 = A*p0=0
00793             fasp_darray_set(m,t0,0.0);
00794             fasp_darray_cp(m,t0,z0);
00795             fasp_darray_cp(m,t0,t1);
00796             fasp_darray_cp(m,t0,z1);

```

```

00797         fasp_darray_cp(m,t0,p1);
00798
00799         fasp_blas_darray_axpy(m,1/normp,t,p1);
00800
00801         // t1 = tp/normp, z1 = tz/normp
00802         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00803         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00804     }
00805 }
00806
00807 // Check III: prevent false convergence
00808 if ( relres < tol ) {
00809
00810     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
00811
00812     // compute residual r = b - Ax again
00813     fasp_darray_cp(m,b->val,r);
00814     fasp_blas_dblc_aAxy(-1.0,A,u->val,r);
00815
00816     // compute residuals
00817     switch (StopType) {
00818     case STOP_REL_RES:
00819         temp2 = fasp_blas_darray_dotprod(m,r,r);
00820         absres = sqrt(temp2);
00821         relres = absres/normr0;
00822         break;
00823     case STOP_REL_PRECRES:
00824         if (pc == NULL)
00825             fasp_darray_cp(m,r,t);
00826         else
00827             pc->fct(r,t,pc->data);
00828         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
00829         absres = sqrt(temp2);
00830         relres = absres/normr0;
00831         break;
00832     case STOP_MOD_REL_RES:
00833         temp2 = fasp_blas_darray_dotprod(m,r,r);
00834         absres = sqrt(temp2);
00835         relres = absres/normu2;
00836         break;
00837     }
00838
00839     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
00840
00841     // check convergence
00842     if ( relres < tol ) break;
00843
00844     if ( more_step >= MaxRestartStep ) {
00845         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
00846         iter = ERROR_SOLVER_TOLSMALL;
00847         break;
00848     }
00849
00850     // prepare for restarting method
00851     fasp_darray_set(m,p0,0.0);
00852     ++more_step;
00853     ++restart_step;
00854
00855     // p1 = B(r)
00856     if ( pc != NULL )
00857         pc->fct(r,p1,pc->data); /* Apply preconditioner */
00858     else
00859         fasp_darray_cp(m,r,p1); /* No preconditioner */
00860
00861     // tp = A*p1
00862     fasp_blas_dblc_mxv(A,p1,tp);
00863
00864     // tz = B(tp)
00865     if ( pc != NULL )
00866         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
00867     else
00868         fasp_darray_cp(m,tp,tz); /* No preconditioner */
00869
00870     // p1 = p1/normp
00871     normp = fasp_blas_darray_dotprod(m,tz,tp);
00872     normp = sqrt(normp);
00873     fasp_darray_cp(m,p1,t);
00874
00875     // t0 = A*p0 = 0
00876     fasp_darray_set(m,t0,0.0);
00877     fasp_darray_cp(m,t0,z0);

```

```

00878         fasp_darray_cp(m,t0,t1);
00879         fasp_darray_cp(m,t0,z1);
00880         fasp_darray_cp(m,t0,p1);
00881
00882         fasp_blas_darray_axpy(m,1/normp,t,p1);
00883
00884         // t1=tp/normp,z1=tz/normp
00885         fasp_blas_darray_axpy(m,1/normp,tp,t1);
00886         fasp_blas_darray_axpy(m,1/normp,tz,z1);
00887
00888     } // end of convergence check
00889
00890     // update relative residual here
00891     absres0 = absres;
00892
00893 } // end of the main loop
00894
00895 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00896 if ( iter != iter_best ) {
00897     // compute best residual
00898     fasp_darray_cp(m,b->val,r);
00899     fasp_blas_dblc_aAxy(-1.0,A,u_best,r);
00900
00901     switch ( StopType ) {
00902     case STOP_REL_RES:
00903         absres_best = fasp_blas_darray_norm2(m,r);
00904         break;
00905     case STOP_REL_PRECRES:
00906         if ( pc != NULL )
00907             pc->fct(r,t,pc->data); /* Apply preconditioner */
00908         else
00909             fasp_darray_cp(m,r,t); /* No preconditioner */
00910         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
00911         break;
00912     case STOP_MOD_REL_RES:
00913         absres_best = fasp_blas_darray_norm2(m,r);
00914         break;
00915     }
00916
00917     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00918         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00919         fasp_darray_cp(m,u_best,u->val);
00920         relres = absres_best / normr0;
00921     }
00922 }
00923 }
00924
00925 FINISHED: // finish iterative method
00926 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00927
00928 // clean up temp memory
00929 fasp_mem_free(work); work = NULL;
00930
00931 #if DEBUG_MODE > 0
00932 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00933 #endif
00934
00935 if ( iter > MaxIt )
00936     return ERROR_SOLVER_MAXIT;
00937 else
00938     return iter;
00939 }
00940
00962 INT fasp_solver_dstr_spmminres (const dSTRmat *A,
00963                                const dvector *b,
00964                                dvector *u,
00965                                precondition *pc,
00966                                const REAL tol,
00967                                const INT MaxIt,
00968                                const SHORT StopType,
00969                                const SHORT PrtLvl)
00970 {
00971     const SHORT MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
00972     const INT m = b->row;
00973     const REAL maxdiff = tol*STAG_RATIO; // stagantation tolerance
00974     const REAL sol_inf_tol = SMALLREAL; // infinity norm tolerance
00975
00976     // local variables
00977     INT iter = 0, stag = 1, more_step = 1, restart_step = 1;
00978     REAL absres0 = BIGREAL, absres = BIGREAL;
00979     REAL normr0 = BIGREAL, relres = BIGREAL;

```

```

00980     REAL      normu2, normuu, normp, normuinf, factor;
00981     REAL      alpha, alpha0, alpha1, temp2;
00982     INT       iter_best = 0; // initial best known iteration
00983     REAL      absres_best = BIGREAL; // initial best known residual
00984
00985     // allocate temp memory (need 12*m REAL)
00986     REAL *work=(REAL *)fasp_mem_calloc(12*m,sizeof(REAL));
00987     REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
00988     REAL *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m, *u_best = r+m;
00989
00990     // Output some info for debugging
00991     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (STR) ...\n");
00992
00993     #if DEBUG_MODE > 0
00994     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00995     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00996     #endif
00997
00998     // p0 = 0
00999     fasp_darray_set(m,p0,0.0);
01000
01001     // r = b-A*u
01002     fasp_darray_cp(m,b->val,r);
01003     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01004
01005     // p1 = B(r)
01006     if ( pc != NULL )
01007         pc->fct(r,p1,pc->data); /* Apply preconditioner */
01008     else
01009         fasp_darray_cp(m,r,p1); /* No preconditioner */
01010
01011     // compute initial residuals
01012     switch ( StopType ) {
01013     case STOP_REL_RES:
01014         absres0 = fasp_blas_darray_norm2(m,r);
01015         normr0 = MAX(SMALLREAL,absres0);
01016         relres = absres0/normr0;
01017         break;
01018     case STOP_REL_PRECRES:
01019         absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
01020         normr0 = MAX(SMALLREAL,absres0);
01021         relres = absres0/normr0;
01022         break;
01023     case STOP_MOD_REL_RES:
01024         absres0 = fasp_blas_darray_norm2(m,r);
01025         normu2 = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
01026         relres = absres0/normu2;
01027         break;
01028     default:
01029         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01030         goto FINISHED;
01031     }
01032
01033     // if initial residual is small, no need to iterate!
01034     if ( relres < tol ) goto FINISHED;
01035
01036     // output iteration information if needed
01037     fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);
01038
01039     // tp = A*p1
01040     fasp_blas_dstr_m xv(A,p1,tp);
01041
01042     // tz = B(tp)
01043     if ( pc != NULL )
01044         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
01045     else
01046         fasp_darray_cp(m,tp,tz); /* No preconditioner */
01047
01048     // p1 = p1/normp
01049     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
01050     normp = sqrt(normp);
01051     fasp_darray_cp(m,p1,t);
01052     fasp_darray_set(m,p1,0.0);
01053     fasp_blas_darray_axpy(m,1/normp,t,p1);
01054
01055     // t0 = A*p0 = 0
01056     fasp_darray_set(m,t0,0.0);
01057     fasp_darray_cp(m,t0,z0);
01058     fasp_darray_cp(m,t0,t1);
01059     fasp_darray_cp(m,t0,z1);
01060

```



```

01061 // t1 = tp/normp, z1 = tz/normp
01062 fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
01063 fasp_blas_darray_axpy(m,1.0/normp,tz,z1);
01064
01065 // main MinRes loop
01066 while ( iter++ < MaxIt ) {
01067     // alpha = <r,z1>
01068     alpha=fasp_blas_darray_dotprod(m,r,z1);
01069
01070     // u = u+alpha*p1
01071     fasp_blas_darray_axpy(m,alpha,p1,u->val);
01072
01073     // r = r-alpha*Ap1
01074     fasp_blas_darray_axpy(m,-alpha,t1,r);
01075
01076     // compute t = A*z1 alpha1 = <z1,t>
01077     fasp_blas_dstr_mxv(A,z1,t);
01078     alpha1=fasp_blas_darray_dotprod(m,z1,t);
01079
01080     // compute t = A*z0 alpha0 = <z1,t>
01081     fasp_blas_dstr_mxv(A,z0,t);
01082     alpha0=fasp_blas_darray_dotprod(m,z1,t);
01083
01084     // p2 = z1-alpha1*p1-alpha0*p0
01085     fasp_darray_cp(m,z1,p2);
01086     fasp_blas_darray_axpy(m,-alpha1,p1,p2);
01087     fasp_blas_darray_axpy(m,-alpha0,p0,p2);
01088
01089     // tp = A*p2
01090     fasp_blas_dstr_mxv(A,p2,tp);
01091
01092     // tz = B(tp)
01093     if ( pc != NULL )
01094         pc->fct(tp,tz,pc->data); /* Apply preconditioner */
01095     else
01096         fasp_darray_cp(m,tp,tz); /* No preconditioner */
01097
01098     // p2 = p2/normp
01099     normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
01100     normp = sqrt(normp);
01101     fasp_darray_cp(m,p2,t);
01102     fasp_darray_set(m,p2,0.0);
01103     fasp_blas_darray_axpy(m,1/normp,t,p2);
01104
01105     // prepare for next iteration
01106     fasp_darray_cp(m,p1,p0);
01107     fasp_darray_cp(m,p2,p1);
01108     fasp_darray_cp(m,t1,t0);
01109     fasp_darray_cp(m,z1,z0);
01110
01111     // t1=tp/normp,z1=tz/normp
01112     fasp_darray_set(m,t1,0.0);
01113     fasp_darray_cp(m,t1,z1);
01114     fasp_blas_darray_axpy(m,1/normp,tp,t1);
01115     fasp_blas_darray_axpy(m,1/normp,tz,z1);
01116
01117     normu2 = fasp_blas_darray_norm2(m,u->val);
01118
01119     // compute residuals
01120     switch ( StopType ) {
01121     case STOP_REL_RES:
01122         temp2 = fasp_blas_darray_dotprod(m,r,r);
01123         absres = sqrt(temp2);
01124         relres = absres/normr0;
01125         break;
01126     case STOP_REL_PRECRES:
01127         if (pc == NULL)
01128             fasp_darray_cp(m,r,t);
01129         else
01130             pc->fct(r,t,pc->data);
01131         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
01132         absres = sqrt(temp2);
01133         relres = absres/normr0;
01134         break;
01135     case STOP_MOD_REL_RES:
01136         temp2 = fasp_blas_darray_dotprod(m,r,r);
01137         absres = sqrt(temp2);
01138         relres = absres/normu2;
01139         break;
01140     }
01141 }

```

```

01142
01143 // compute reduction factor of residual ||r||
01144 factor = absres/absres0;
01145
01146 // output iteration information if needed
01147 fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);
01148
01149 // safety net check: save the best-so-far solution
01150 if ( fasp_dvec_isnan(u) ) {
01151     // If the solution is NAN, restore the best solution
01152     absres = BIGREAL;
01153     goto RESTORE_BESTSOL;
01154 }
01155
01156 if ( absres < absres_best - maxdiff ) {
01157     absres_best = absres;
01158     iter_best = iter;
01159     fasp_darray_cp(m,u->val,u_best);
01160 }
01161
01162 // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
01163 normuinf = fasp_blas_darray_norminf(m, u->val);
01164 if (normuinf <= sol_inf_tol) {
01165     if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
01166     iter = ERROR_SOLVER_SOLSTAG;
01167     break;
01168 }
01169
01170 // Check II: if staggenated, try to restart
01171 normuu = fasp_blas_darray_norm2(m,p1);
01172 normuu = ABS(alpha)*(normuu/normu2);
01173
01174 if ( normuu < maxdiff ) {
01175     if ( stag < MaxStag ) {
01176         if ( PrtLvl >= PRINT_MORE ) {
01177             ITS_DIFFRES(normuu,relres);
01178             ITS_RESTART;
01179         }
01180     }
01181 }
01182
01183 fasp_darray_cp(m,b->val,r);
01184 fasp_blas_dstr_aAxp(-1.0,A,u->val,r);
01185
01186 // compute residuals
01187 switch (StopType) {
01188     case STOP_REL_RES:
01189         temp2 = fasp_blas_darray_dotprod(m,r,r);
01190         absres = sqrt(temp2);
01191         relres = absres/normr0;
01192         break;
01193     case STOP_REL_PRECRES:
01194         if (pc == NULL)
01195             fasp_darray_cp(m,r,t);
01196         else
01197             pc->fct(r,t,pc->data);
01198         temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
01199         absres = sqrt(temp2);
01200         relres = absres/normr0;
01201         break;
01202     case STOP_MOD_REL_RES:
01203         temp2 = fasp_blas_darray_dotprod(m,r,r);
01204         absres = sqrt(temp2);
01205         relres = absres/normu2;
01206         break;
01207 }
01208
01209 if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01210
01211 if ( relres < tol )
01212     break;
01213 else {
01214     if ( stag >= MaxStag ) {
01215         if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
01216         iter = ERROR_SOLVER_STAG;
01217         break;
01218     }
01219     fasp_darray_set(m,p0,0.0);
01220     ++stag;
01221     ++restart_step;
01222 }

```

```

01223         // p1 = B(r)
01224         if ( pc != NULL )
01225             pc->fct(r,p1,pc->data); /* Apply preconditioner */
01226         else
01227             fasp_darray_cp(m,r,p1); /* No preconditioner */
01228
01229         // tp = A*p1
01230         fasp_blas_dstr_mxv(A,p1,tp);
01231
01232         // tz = B(tp)
01233         if ( pc != NULL )
01234             pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
01235         else
01236             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01237
01238         // p1 = p1/normp
01239         normp = fasp_blas_darray_dotprod(m,tz,tp);
01240         normp = sqrt(normp);
01241         fasp_darray_cp(m,p1,t);
01242
01243         // t0 = A*p0=0
01244         fasp_darray_set(m,t0,0.0);
01245         fasp_darray_cp(m,t0,z0);
01246         fasp_darray_cp(m,t0,t1);
01247         fasp_darray_cp(m,t0,z1);
01248         fasp_darray_cp(m,t0,p1);
01249
01250         fasp_blas_darray_axpy(m,1/normp,t,p1);
01251
01252         // t1 = tp/normp, z1 = tz/normp
01253         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01254         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01255     }
01256 }
01257
01258 // Check III: prevent false convergence
01259 if ( relres < tol ) {
01260
01261     if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);
01262
01263     // compute residual r = b - Ax again
01264     fasp_darray_cp(m,b->val,r);
01265     fasp_blas_dstr_aAxy(-1.0,A,u->val,r);
01266
01267     // compute residuals
01268     switch (StopType) {
01269         case STOP_REL_RES:
01270             temp2 = fasp_blas_darray_dotprod(m,r,r);
01271             absres = sqrt(temp2);
01272             relres = absres/normr0;
01273             break;
01274         case STOP_REL_PRECRES:
01275             if (pc == NULL)
01276                 fasp_darray_cp(m,r,t);
01277             else
01278                 pc->fct(r,t,pc->data);
01279             temp2 = ABS(fasp_blas_darray_dotprod(m,r,t));
01280             absres = sqrt(temp2);
01281             relres = absres/normr0;
01282             break;
01283         case STOP_MOD_REL_RES:
01284             temp2 = fasp_blas_darray_dotprod(m,r,r);
01285             absres = sqrt(temp2);
01286             relres = absres/normu2;
01287             break;
01288     }
01289
01290     if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);
01291
01292     // check convergence
01293     if ( relres < tol ) break;
01294
01295     if ( more_step >= MaxRestartStep ) {
01296         if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
01297         iter = ERROR_SOLVER_TOLSMALL;
01298         break;
01299     }
01300
01301     // prepare for restarting method
01302     fasp_darray_set(m,p0,0.0);
01303     ++more_step;

```

```

01304         ++restart_step;
01305
01306         // p1 = B(r)
01307         if ( pc != NULL )
01308             pc->fct(r,p1,pc->data); /* Apply preconditioner */
01309         else
01310             fasp_darray_cp(m,r,p1); /* No preconditioner */
01311
01312         // tp = A*p1
01313         fasp_blas_dstr_mvx(A,p1,tp);
01314
01315         // tz = B(tp)
01316         if ( pc != NULL )
01317             pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
01318         else
01319             fasp_darray_cp(m,tp,tz); /* No preconditioner */
01320
01321         // p1 = p1/normp
01322         normp = fasp_blas_darray_dotprod(m,tz,tp);
01323         normp = sqrt(normp);
01324         fasp_darray_cp(m,p1,t);
01325
01326         // t0 = A*p0 = 0
01327         fasp_darray_set(m,t0,0.0);
01328         fasp_darray_cp(m,t0,z0);
01329         fasp_darray_cp(m,t0,t1);
01330         fasp_darray_cp(m,t0,z1);
01331         fasp_darray_cp(m,t0,p1);
01332
01333         fasp_blas_darray_axpy(m,1/normp,t,p1);
01334
01335         // t1=tp/normp,z1=tz/normp
01336         fasp_blas_darray_axpy(m,1/normp,tp,t1);
01337         fasp_blas_darray_axpy(m,1/normp,tz,z1);
01338
01339     } // end of convergence check
01340
01341     // update relative residual here
01342     absres0 = absres;
01343
01344 } // end of the main loop
01345
01346 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01347 if ( iter != iter_best ) {
01348
01349     // compute best residual
01350     fasp_darray_cp(m,b->val,r);
01351     fasp_blas_dstr_aAxy(-1.0,A,u_best,r);
01352
01353     switch ( StopType ) {
01354     case STOP_REL_RES:
01355         absres_best = fasp_blas_darray_norm2(m,r);
01356         break;
01357     case STOP_REL_PRECRES:
01358         if ( pc != NULL )
01359             pc->fct(r,t,pc->data); /* Apply preconditioner */
01360         else
01361             fasp_darray_cp(m,r,t); /* No preconditioner */
01362         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
01363         break;
01364     case STOP_MOD_REL_RES:
01365         absres_best = fasp_blas_darray_norm2(m,r);
01366         break;
01367     }
01368
01369     if ( absres > absres_best + maxdiff || isnan(absres) ) {
01370         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01371         fasp_darray_cp(m,u_best,u->val);
01372         relres = absres_best / normr0;
01373     }
01374 }
01375
01376 FINISHED: // finish iterative method
01377 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01378
01379 // clean up temp memory
01380 fasp_mem_free(work); work = NULL;
01381
01382 #if DEBUG_MODE > 0
01383 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01384 #endif

```

```

01385
01386     if ( iter > MaxIt )
01387         return ERROR_SOLVER_MAXIT;
01388     else
01389         return iter;
01390 }
01391
01392 /*-----*/
01393 /*--      End of File      --*/
01394 /*-----*/

```

9.133 KrySPvgmres.c File Reference

Krylov subspace methods – Preconditioned variable-restart GMRes with safety net.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dcsr_spvgmres](#) (const [dCSRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.
- [INT fasp_solver_dbsr_spvgmres](#) (const [dBSRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.
- [INT fasp_solver_dblc_spvgmres](#) (const [dBLCmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Preconditioned GMRES method for solving Au=b.
- [INT fasp_solver_dstr_spvgmres](#) (const [dSTRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, const [REAL](#) tol, const [INT](#) MaxIt, [SHORT](#) restart, const [SHORT](#) StopType, const [SHORT](#) PrtLvl)
Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

9.133.1 Detailed Description

Krylov subspace methods – Preconditioned variable-restart GMRes with safety net.

Note

This file contains Level-3 (Kry) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [BlaSpmvSTR.c](#)

The 'best' iterative solution will be saved and used upon exit; See [KryPvgmres.c](#) a version without safety net

Reference: A.H. Baker, E.R. Jessup, and Tz.V. Kolev A Simple Strategy for Varying the Restart Parameter in GMRES(m) *Journal of Computational and Applied Mathematics*, 230 (2009) pp. 751-761. UCRL-JRNL-235266.
Copyright (C) 2013–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Use one single function for all! –Chensong

Definition in file [KrySPvgmres.c](#).

9.133.2 Function Documentation

9.133.2.1 fasp_solver_dblc_spvgmres()

```

INT fasp_solver_dblc_spvgmres (
    const dBLMat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Preconditioned GMRES method for solving $Au=b$.

Parameters

<i>A</i>	Pointer to dBLMat : coefficient matrix
<i>b</i>	Pointer to dvector : right hand side
<i>x</i>	Pointer to dvector : unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/06/2013

Definition at line [829](#) of file [KrySPvgmres.c](#).

9.133.2.2 fasp_solver_dbsr_spvgmres()

```

INT fasp_solver_dbsr_spvgmres (
    const dBSRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,

```

```

    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dBSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/06/2013

Definition at line [449](#) of file [KrySPvgmres.c](#).

9.133.2.3 fasp_solver_dcsr_spvgmres()

```

INT fasp_solver_dcsr_spvgmres (
    const dCSRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dCSRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns

Parameters

<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/06/2013

Modified by Chunsheng Feng on 07/22/2013: Add adapt memory allocate
Definition at line 68 of file [KrySPvgmres.c](#).

9.133.2.4 fasp_solver_dstr_spvgmres()

```

INT fasp_solver_dstr_spvgmres (
    const dSTRmat * A,
    const dvector * b,
    dvector * x,
    precondition * pc,
    const REAL tol,
    const INT MaxIt,
    SHORT restart,
    const SHORT StopType,
    const SHORT PrtLvl )

```

Solve "Ax=b" using PGMRES(right preconditioned) iterative method in which the restart parameter can be adaptively modified during iteration.

Parameters

<i>A</i>	Pointer to dSTRmat : coefficient matrix
<i>b</i>	Pointer to dvector: right hand side
<i>x</i>	Pointer to dvector: unknowns
<i>pc</i>	Pointer to structure of precondition (precond)
<i>tol</i>	Tolerance for stopping
<i>MaxIt</i>	Maximal number of iterations
<i>restart</i>	Restarting steps
<i>StopType</i>	Stopping criteria type
<i>PrtLvl</i>	How much information to print out

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/06/2013

Definition at line 1210 of file KrySPvgmres.c.

9.134 KrySPvgmres.c

[Go to the documentation of this file.](#)

```

00001
00027 #include <math.h>
00028
00029 #include "fasp.h"
00030 #include "fasp_functs.h"
00031
00032 /*-----*/
00033 /*--  Declare Private Functions  --*/
00034 /*-----*/
00035
00036 #include "KryUtil.inl"
00037
00038 /*-----*/
00039 /*--      Public Functions      --*/
00040 /*-----*/
00041
00068 INT fasp_solver_dcsr_spvgmres (const dCSRmat *A,
00069                               const dvector *b,
00070                               dvector *X,
00071                               precondition *pc,
00072                               const REAL tol,
00073                               const INT MaxIt,
00074                               SHORT restart,
00075                               const SHORT StopType,
00076                               const SHORT PrtLvl)
00077 {
00078     const INT n          = b->row;
00079     const INT MIN_ITER    = 0;
00080     const REAL maxdiff    = tol*STAG_RATIO; // stagnation tolerance
00081     const REAL epsmac     = SMALLREAL;
00082
00083     //-----//
00084     //  Newly added parameters to monitor when  //
00085     //  to change the restart parameter         //
00086     //-----//
00087     const REAL cr_max     = 0.99;    // = cos(8^o) (experimental)
00088     const REAL cr_min     = 0.174;   // = cos(80^o) (experimental)
00089
00090     // local variables
00091     INT iter              = 0;
00092     INT restart1          = restart + 1;
00093     int i, j, k; // must be signed! -zcs
00094
00095     REAL r_norm, r_normb, gamma, t;
00096     REAL normr0 = BIGREAL, absres = BIGREAL;
00097     REAL relres = BIGREAL, normu = BIGREAL;
00098
00099     REAL cr          = 1.0;    // convergence rate
00100     REAL r_norm_old  = 0.0;    // save residual norm of previous restart cycle
00101     INT d            = 3;      // reduction for restart parameter
00102     INT restart_max   = restart; // upper bound for restart in each restart cycle
00103     INT restart_min   = 3;      // lower bound for restart (should be small)
00104     INT Restart       = restart; // real restart in some fixed restarted cycle
00105
00106     INT iter_best = 0;    // initial best known iteration
00107     REAL absres_best = BIGREAL; // initial best known residual
00108

```

```

00109 // allocate temp memory (need about (restart+4)*n REAL numbers)
00110 REAL *c = NULL, *s = NULL, *rs = NULL;
00111 REAL *norms = NULL, *r = NULL, *w = NULL;
00112 REAL *work = NULL, *x_best = NULL;
00113 REAL **p = NULL, **hh = NULL;
00114
00115 // Output some info for debugging
00116 if (PrtLvl > PRINT_NONE) printf("\nCalling Safe VGMRes solver (CSR) ...\n");
00117
00118 #if DEBUG_MODE > 0
00119 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00120 printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00121 #endif
00122
00123 /* allocate memory and setup temp work space */
00124 work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00125
00126 /* check whether memory is enough for GMRES */
00127 while ( (work == NULL) && (restart > 5) ) {
00128     restart = restart - 5;
00129     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00130     printf("### WARNING: vGMRES restart number set to %d!\n", restart);
00131     restart1 = restart + 1;
00132 }
00133
00134 if (work == NULL) {
00135     printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00136     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00137 }
00138
00139 p = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00140 hh = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00141 norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00142
00143 r = work; w = r + n; rs = w + n; c = rs + restart1;
00144 x_best = c + restart; s = x_best + n;
00145
00146 for (i = 0; i < restart1; i++) p[i] = s + restart + i*n;
00147
00148 for (i = 0; i < restart1; i++) hh[i] = p[restart] + n + i*restart;
00149
00150 // r = b-A*x
00151 fasp_darray_cp(n, b->val, p[0]);
00152 fasp_blas_dcsr_aAxy(-1.0, A, x->val, p[0]);
00153
00154 r_norm = fasp_blas_darray_norm2(n, p[0]);
00155
00156 // compute initial residuals
00157 switch (StopType) {
00158     case STOP_REL_RES:
00159         normr0 = MAX(SMALLREAL, r_norm);
00160         relres = r_norm/normr0;
00161         break;
00162     case STOP_REL_PRECRES:
00163         if (pc == NULL)
00164             fasp_darray_cp(n, p[0], r);
00165         else
00166             pc->fct(p[0], r, pc->data);
00167         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
00168         normr0 = MAX(SMALLREAL, r_normb);
00169         relres = r_normb/normr0;
00170         break;
00171     case STOP_MOD_REL_RES:
00172         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00173         normr0 = r_norm;
00174         relres = normr0/normu;
00175         break;
00176     default:
00177         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00178         goto FINISHED;
00179 }
00180
00181 // if initial residual is small, no need to iterate!
00182 if (relres < tol) goto FINISHED;
00183
00184 // output iteration information if needed
00185 fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
00186
00187 // store initial residual
00188 norms[0] = relres;
00189

```

```

00190      /* outer iteration cycle */
00191      while ( iter < MaxIt ) {
00192
00193          rs[0] = r_norm_old = r_norm;
00194
00195          t = 1.0 / r_norm;
00196
00197          fasp_blas_darray_ax(n, t, p[0]);
00198
00199          //-----//
00200          //   adjust the restart parameter   //
00201          //-----//
00202          if ( cr > cr_max || iter == 0 ) {
00203              Restart = restart_max;
00204          }
00205          else if ( cr < cr_min ) {
00206              // Restart = Restart;
00207          }
00208          else {
00209              if ( Restart - d > restart_min ) {
00210                  Restart -= d;
00211              }
00212              else {
00213                  Restart = restart_max;
00214              }
00215          }
00216
00217          /* RESTART CYCLE (right-preconditioning) */
00218          i = 0;
00219          while ( i < Restart && iter < MaxIt ) {
00220
00221              i++; iter++;
00222
00223              /* apply preconditioner */
00224              if (pc == NULL)
00225                  fasp_darray_cp(n, p[i-1], r);
00226              else
00227                  pc->fct(p[i-1], r, pc->data);
00228
00229              fasp_blas_dcsr_mnv(A, r, p[i]);
00230
00231              /* modified Gram_Schmidt */
00232              for (j = 0; j < i; j++) {
00233                  hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00234                  fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00235              }
00236              t = fasp_blas_darray_norm2(n, p[i]);
00237              hh[i][i-1] = t;
00238              if (t != 0.0) {
00239                  t = 1.0/t;
00240                  fasp_blas_darray_ax(n, t, p[i]);
00241              }
00242
00243              for (j = 1; j < i; ++j) {
00244                  t = hh[j-1][i-1];
00245                  hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00246                  hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00247              }
00248              t = hh[i][i-1]*hh[i][i-1];
00249              t += hh[i-1][i-1]*hh[i-1][i-1];
00250
00251              gamma = sqrt(t);
00252              if (gamma == 0.0) gamma = epsmac;
00253              c[i-1] = hh[i-1][i-1] / gamma;
00254              s[i-1] = hh[i][i-1] / gamma;
00255              rs[i] = -s[i-1]*rs[i-1];
00256              rs[i-1] = c[i-1]*rs[i-1];
00257              hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00258
00259              absres = r_norm = fabs(rs[i]);
00260
00261              relres = absres/normr0;
00262
00263              norms[iter] = relres;
00264
00265              // output iteration information if needed
00266              fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00267                          norms[iter]/norms[iter-1]);
00268
00269              // should we exit restart cycle
00270              if ( relres <= tol && iter >= MIN_ITER ) break;

```

```

00271
00272     } /* end of restart cycle */
00273
00274     /* now compute solution, first solve upper triangular system */
00275     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00276     for (k = i-2; k >= 0; k --) {
00277         t = 0.0;
00278         for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
00279
00280         t += rs[k];
00281         rs[k] = t / hh[k][k];
00282     }
00283
00284     fasp_darray_cp(n, p[i-1], w);
00285
00286     fasp_blas_darray_ax(n, rs[i-1], w);
00287
00288     for (j = i-2; j >= 0; j--) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00289
00290     /* apply preconditioner */
00291     if ( pc == NULL )
00292         fasp_darray_cp(n, w, r);
00293     else
00294         pc->fct(w, r, pc->data);
00295
00296     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00297
00298     // safety net check: save the best-so-far solution
00299     if ( fasp_dvec_isnan(x) ) {
00300         // If the solution is NAN, restore the best solution
00301         absres = BIGREAL;
00302         goto RESTORE_BESTSOL;
00303     }
00304
00305     if ( absres < absres_best - maxdiff ) {
00306         absres_best = absres;
00307         iter_best = iter;
00308         fasp_darray_cp(n, x->val, x_best);
00309     }
00310
00311     // Check: prevent false convergence
00312     if ( relres <= tol && iter >= MIN_ITER ) {
00313
00314         fasp_darray_cp(n, b->val, r);
00315         fasp_blas_dcsr_aAxp(-1.0, A, x->val, r);
00316
00317         r_norm = fasp_blas_darray_norm2(n, r);
00318
00319         switch ( StopType ) {
00320             case STOP_REL_RES:
00321                 absres = r_norm;
00322                 relres = absres/normr0;
00323                 break;
00324             case STOP_REL_PRECRES:
00325                 if ( pc == NULL )
00326                     fasp_darray_cp(n, r, w);
00327                 else
00328                     pc->fct(r, w, pc->data);
00329                 absres = sqrt(fasp_blas_darray_dotprod(n, w, r));
00330                 relres = absres/normr0;
00331                 break;
00332             case STOP_MOD_REL_RES:
00333                 absres = r_norm;
00334                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00335                 relres = absres/normu;
00336                 break;
00337         }
00338
00339         norms[iter] = relres;
00340
00341         if ( relres <= tol ) {
00342             break;
00343         }
00344         else {
00345             // Need to restart
00346             fasp_darray_cp(n, r, p[0]); i = 0;
00347         }
00348     }
00349     } /* end of convergence check */
00350
00351     /* compute residual vector and continue loop */

```

```

00352     for ( j = i; j > 0; j-- ) {
00353         rs[j-1] = -s[j-1]*rs[j];
00354         rs[j]   = c[j-1]*rs[j];
00355     }
00356
00357     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00358
00359     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00360
00361     if ( i ) {
00362         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00363         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00364     }
00365
00366     //-----//
00367     // compute the convergence rate //
00368     //-----//
00369     cr = r_norm / r_norm_old;
00370
00371 } /* end of iteration while loop */
00372
00373 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00374 if ( iter != iter_best ) {
00375
00376     // compute best residual
00377     fasp_darray_cp(n,b->val,r);
00378     fasp_blas_dcsr_aAxy(-1.0,A,x_best,r);
00379
00380     switch ( StopType ) {
00381     case STOP_REL_RES:
00382         absres_best = fasp_blas_darray_norm2(n,r);
00383         break;
00384     case STOP_REL_PRECRES:
00385         // z = B(r)
00386         if ( pc != NULL )
00387             pc->fct(r,w,pc->data); /* Apply preconditioner */
00388         else
00389             fasp_darray_cp(n,r,w); /* No preconditioner */
00390         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
00391         break;
00392     case STOP_MOD_REL_RES:
00393         absres_best = fasp_blas_darray_norm2(n,r);
00394         break;
00395     }
00396
00397     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00398         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00399         fasp_darray_cp(n,x_best,x->val);
00400         relres = absres_best / normr0;
00401     }
00402 }
00403
00404 FINISHED:
00405 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00406
00407 /*-----*/
00408 * Free some stuff
00409 *-----*/
00410 fasp_mem_free(work); work = NULL;
00411 fasp_mem_free(p); p = NULL;
00412 fasp_mem_free(hh); hh = NULL;
00413 fasp_mem_free(norms); norms = NULL;
00414
00415 #if DEBUG_MODE > 0
00416 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00417 #endif
00418
00419 if (iter >= MaxIt)
00420     return ERROR_SOLVER_MAXIT;
00421 else
00422     return iter;
00423 }
00424
00449 INT fasp_solver_dbsr_spvgmres (const dBSRmat *A,
00450                                const dvector *b,
00451                                dvector *x,
00452                                precondition *pc,
00453                                const REAL tol,
00454                                const INT MaxIt,
00455                                SHORT restart,
00456                                const SHORT StopType,

```

```

00457                                     const SHORT      PrtLvl)
00458 {
00459     const INT    n          = b->row;
00460     const INT    MIN_ITER   = 0;
00461     const REAL   maxdiff    = tol*STAG_RATIO; // staganation tolerance
00462     const REAL   epsmac     = SMALLREAL;
00463
00464     //-----//
00465     // Newly added parameters to monitor when //
00466     // to change the restart parameter //
00467     //-----//
00468     const REAL cr_max      = 0.99; // = cos(8^o) (experimental)
00469     const REAL cr_min      = 0.174; // = cos(80^o) (experimental)
00470
00471     // local variables
00472     INT iter               = 0;
00473     INT restart1          = restart + 1;
00474     int i, j, k; // must be signed! -zcs
00475
00476     REAL r_norm, r_normb, gamma, t;
00477     REAL normr0 = BIGREAL, absres = BIGREAL;
00478     REAL relres = BIGREAL, normu = BIGREAL;
00479
00480     REAL cr          = 1.0; // convergence rate
00481     REAL r_norm_old  = 0.0; // save residual norm of previous restart cycle
00482     INT d            = 3; // reduction for restart parameter
00483     INT restart_max   = restart; // upper bound for restart in each restart cycle
00484     INT restart_min   = 3; // lower bound for restart (should be small)
00485     INT Restart       = restart; // real restart in some fixed restarted cycle
00486
00487     INT iter_best = 0; // initial best known iteration
00488     REAL absres_best = BIGREAL; // initial best known residual
00489
00490     // allocate temp memory (need about (restart+4)*n REAL numbers)
00491     REAL *c = NULL, *s = NULL, *rs = NULL;
00492     REAL *norms = NULL, *r = NULL, *w = NULL;
00493     REAL *work = NULL, *x_best = NULL;
00494     REAL **p = NULL, **hh = NULL;
00495
00496     // Output some info for debugging
00497     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (BSR) ...\n");
00498
00499     #if DEBUG_MODE > 0
00500         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00501         printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00502     #endif
00503
00504     /* allocate memory and setup temp work space */
00505     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00506
00507     /* check whether memory is enough for GMRES */
00508     while ( (work == NULL) && (restart > 5) ) {
00509         restart = restart - 5;
00510         work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00511         printf("### WARNING: vGMRES restart number set to %d!\n", restart);
00512         restart1 = restart + 1;
00513     }
00514
00515     if ( work == NULL ) {
00516         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00517         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00518     }
00519
00520     p = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00521     hh = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00522     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00523
00524     r = work; w = r + n; rs = w + n; c = rs + restart1;
00525     x_best = c + restart; s = x_best + n;
00526
00527     for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;
00528
00529     for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;
00530
00531     // r = b-A*x
00532     fasp_darray_cp(n, b->val, p[0]);
00533     fasp_blas_dbsr_aAxy(-1.0, A, x->val, p[0]);
00534
00535     r_norm = fasp_blas_darray_norm2(n, p[0]);
00536
00537     // compute initial residuals

```

```

00538     switch (StopType) {
00539         case STOP_REL_RES:
00540             normr0 = MAX(SMALLREAL, r_norm);
00541             relres = r_norm/normr0;
00542             break;
00543         case STOP_REL_PRECRES:
00544             if ( pc == NULL )
00545                 fasp_darray_cp(n, p[0], r);
00546             else
00547                 pc->fct(p[0], r, pc->data);
00548             r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
00549             normr0 = MAX(SMALLREAL, r_normb);
00550             relres = r_normb/normr0;
00551             break;
00552         case STOP_MOD_REL_RES:
00553             normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00554             normr0 = r_norm;
00555             relres = normr0/normu;
00556             break;
00557         default:
00558             printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00559             goto FINISHED;
00560     }
00561
00562     // if initial residual is small, no need to iterate!
00563     if ( relres < tol ) goto FINISHED;
00564
00565     // output iteration information if needed
00566     fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
00567
00568     // store initial residual
00569     norms[0] = relres;
00570
00571     /* outer iteration cycle */
00572     while ( iter < MaxIt ) {
00573
00574         rs[0] = r_norm_old = r_norm;
00575
00576         t = 1.0 / r_norm;
00577
00578         fasp_blas_darray_ax(n, t, p[0]);
00579
00580         //-----//
00581         // adjust the restart parameter //
00582         //-----//
00583         if ( cr > cr_max || iter == 0 ) {
00584             Restart = restart_max;
00585         }
00586         else if ( cr < cr_min ) {
00587             // Restart = Restart;
00588         }
00589         else {
00590             if ( Restart - d > restart_min ) {
00591                 Restart -= d;
00592             }
00593             else {
00594                 Restart = restart_max;
00595             }
00596         }
00597
00598         /* RESTART CYCLE (right-preconditioning) */
00599         i = 0;
00600         while ( i < Restart && iter < MaxIt ) {
00601
00602             i++; iter++;
00603
00604             /* apply preconditioner */
00605             if (pc == NULL)
00606                 fasp_darray_cp(n, p[i-1], r);
00607             else
00608                 pc->fct(p[i-1], r, pc->data);
00609
00610             fasp_blas_dbsr_mxv(A, r, p[i]);
00611
00612             /* modified Gram_Schmidt */
00613             for (j = 0; j < i; j++) {
00614                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00615                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00616             }
00617             t = fasp_blas_darray_norm2(n, p[i]);
00618             hh[i][i-1] = t;

```

```

00619         if (t != 0.0) {
00620             t = 1.0/t;
00621             fasp_blas_darray_ax(n, t, p[i]);
00622         }
00623
00624         for (j = 1; j < i; ++j) {
00625             t = hh[j-1][i-1];
00626             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
00627             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
00628         }
00629         t = hh[i][i-1]*hh[i][i-1];
00630         t += hh[i-1][i-1]*hh[i-1][i-1];
00631
00632         gamma = sqrt(t);
00633         if (gamma == 0.0) gamma = epsmac;
00634         c[i-1] = hh[i-1][i-1] / gamma;
00635         s[i-1] = hh[i][i-1] / gamma;
00636         rs[i] = -s[i-1]*rs[i-1];
00637         rs[i-1] = c[i-1]*rs[i-1];
00638         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
00639
00640         absres = r_norm = fabs(rs[i]);
00641
00642         relres = absres/normr0;
00643
00644         norms[iter] = relres;
00645
00646         // output iteration information if needed
00647         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
00648                   norms[iter]/norms[iter-1]);
00649
00650         // should we exit restart cycle
00651         if ( relres <= tol && iter >= MIN_ITER ) break;
00652
00653     } /* end of restart cycle */
00654
00655     /* now compute solution, first solve upper triangular system */
00656     rs[i-1] = rs[i-1] / hh[i-1][i-1];
00657     for (k = i-2; k >= 0; k --) {
00658         t = 0.0;
00659         for (j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
00660
00661         t += rs[k];
00662         rs[k] = t / hh[k][k];
00663     }
00664
00665     fasp_darray_cp(n, p[i-1], w);
00666
00667     fasp_blas_darray_ax(n, rs[i-1], w);
00668
00669     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
00670
00671     /* apply preconditioner */
00672     if ( pc == NULL )
00673         fasp_darray_cp(n, w, r);
00674     else
00675         pc->fct(w, r, pc->data);
00676
00677     fasp_blas_darray_axpy(n, 1.0, r, x->val);
00678
00679     // safety net check: save the best-so-far solution
00680     if ( fasp_dvec_isnan(x) ) {
00681         // If the solution is NAN, restore the best solution
00682         absres = BIGREAL;
00683         goto RESTORE_BESTSOL;
00684     }
00685
00686     if ( absres < absres_best - maxdiff) {
00687         absres_best = absres;
00688         iter_best = iter;
00689         fasp_darray_cp(n,x->val,x_best);
00690     }
00691
00692     // Check: prevent false convergence
00693     if ( relres <= tol && iter >= MIN_ITER ) {
00694
00695         fasp_darray_cp(n, b->val, r);
00696         fasp_blas_dbsr_aApy(-1.0, A, x->val, r);
00697
00698         r_norm = fasp_blas_darray_norm2(n, r);
00699

```



```

00700         switch ( StopType ) {
00701             case STOP_REL_RES:
00702                 absres = r_norm;
00703                 relres = absres/normr0;
00704                 break;
00705             case STOP_REL_PRECRES:
00706                 if ( pc == NULL )
00707                     fasp_darray_cp(n, r, w);
00708                 else
00709                     pc->fct(r, w, pc->data);
00710                 absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
00711                 relres = absres/normr0;
00712                 break;
00713             case STOP_MOD_REL_RES:
00714                 absres = r_norm;
00715                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00716                 relres = absres/normu;
00717                 break;
00718         }
00719
00720         norms[iter] = relres;
00721
00722         if ( relres <= tol ) {
00723             break;
00724         }
00725         else {
00726             // Need to restart
00727             fasp_darray_cp(n, r, p[0]); i = 0;
00728         }
00729
00730     } /* end of convergence check */
00731
00732     /* compute residual vector and continue loop */
00733     for ( j = i; j > 0; j-- ) {
00734         rs[j-1] = -s[j-1]*rs[j];
00735         rs[j] = c[j-1]*rs[j];
00736     }
00737
00738     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
00739
00740     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
00741
00742     if ( i ) {
00743         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
00744         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
00745     }
00746
00747     //-----//
00748     // compute the convergence rate //
00749     //-----//
00750     cr = r_norm / r_norm_old;
00751
00752 } /* end of iteration while loop */
00753
00754 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
00755 if ( iter != iter_best ) {
00756
00757     // compute best residual
00758     fasp_darray_cp(n, b->val, r);
00759     fasp_blas_dbsr_aAxy(-1.0, A, x_best, r);
00760
00761     switch ( StopType ) {
00762         case STOP_REL_RES:
00763             absres_best = fasp_blas_darray_norm2(n, r);
00764             break;
00765         case STOP_REL_PRECRES:
00766             // z = B(r)
00767             if ( pc != NULL )
00768                 pc->fct(r, w, pc->data); /* Apply preconditioner */
00769             else
00770                 fasp_darray_cp(n, r, w); /* No preconditioner */
00771             absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n, w, r)));
00772             break;
00773         case STOP_MOD_REL_RES:
00774             absres_best = fasp_blas_darray_norm2(n, r);
00775             break;
00776     }
00777
00778     if ( absres > absres_best + maxdiff || isnan(absres) ) {
00779         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
00780         fasp_darray_cp(n, x_best, x->val);

```

```

00781         relres = absres_best / normr0;
00782     }
00783 }
00784
00785 FINISHED:
00786     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
00787
00788     /*-----
00789 * Free some stuff
00790 *-----*/
00791     fasp_mem_free(work); work = NULL;
00792     fasp_mem_free(p); p = NULL;
00793     fasp_mem_free(hh); hh = NULL;
00794     fasp_mem_free(norms); norms = NULL;
00795
00796 #if DEBUG_MODE > 0
00797     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00798 #endif
00799
00800     if (iter>=MaxIt)
00801         return ERROR_SOLVER_MAXIT;
00802     else
00803         return iter;
00804 }
00805
00829 INT fasp_solver_dblc_spvgmres (const dBLCmat *A,
00830                                const dvector *b,
00831                                dvector *x,
00832                                precondition *pc,
00833                                const REAL tol,
00834                                const INT MaxIt,
00835                                SHORT restart,
00836                                const SHORT StopType,
00837                                const SHORT PrtLvl)
00838 {
00839     const INT n = b->row;
00840     const INT MIN_ITER = 0;
00841     const REAL maxdiff = tol*STAG_RATIO; // staganation tolerance
00842     const REAL epsmac = SMALLREAL;
00843
00844     /*-----
00845 // Newly added parameters to monitor when //
00846 // to change the restart parameter //
00847 //-----*/
00848     const REAL cr_max = 0.99; // = cos(8^o) (experimental)
00849     const REAL cr_min = 0.174; // = cos(80^o) (experimental)
00850
00851     // local variables
00852     INT iter = 0;
00853     INT restart1 = restart + 1;
00854     int i, j, k; // must be signed! -zcs
00855
00856     REAL r_norm, r_normb, gamma, t;
00857     REAL normr0 = BIGREAL, absres = BIGREAL;
00858     REAL relres = BIGREAL, normu = BIGREAL;
00859
00860     REAL cr = 1.0; // convergence rate
00861     REAL r_norm_old = 0.0; // save residual norm of previous restart cycle
00862     INT d = 3; // reduction for restart parameter
00863     INT restart_max = restart; // upper bound for restart in each restart cycle
00864     INT restart_min = 3; // lower bound for restart (should be small)
00865     INT Restart = restart; // real restart in some fixed restarted cycle
00866
00867     INT iter_best = 0; // initial best known iteration
00868     REAL absres_best = BIGREAL; // initial best known residual
00869
00870     // allocate temp memory (need about (restart+4)*n REAL numbers)
00871     REAL *c = NULL, *s = NULL, *rs = NULL;
00872     REAL *norms = NULL, *r = NULL, *w = NULL;
00873     REAL *work = NULL, *x_best = NULL;
00874     REAL **p = NULL, **hh = NULL;
00875
00876     // Output some info for debugging
00877     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (BLC) ...\n");
00878
00879 #if DEBUG_MODE > 0
00880     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00881     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
00882 #endif
00883
00884     /* allocate memory and setup temp work space */

```

```

00885     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00886
00887     /* check whether memory is enough for GMRES */
00888     while ( (work == NULL) && (restart > 5) ) {
00889         restart = restart - 5;
00890         work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
00891         printf("### WARNING: vGMRES restart number set to %d!\n", restart);
00892         restart1 = restart + 1;
00893     }
00894
00895     if ( work == NULL ) {
00896         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
00897         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00898     }
00899
00900     p = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00901     hh = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
00902     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
00903
00904     r = work; w = r + n; rs = w + n; c = rs + restart1;
00905     x_best = c + restart; s = x_best + n;
00906
00907     for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;
00908
00909     for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;
00910
00911     // r = b-A*x
00912     fasp_darray_cp(n, b->val, p[0]);
00913     fasp_blas_dblc_aAxy(-1.0, A, x->val, p[0]);
00914
00915     r_norm = fasp_blas_darray_norm2(n, p[0]);
00916
00917     // compute initial residuals
00918     switch (StopType) {
00919     case STOP_REL_RES:
00920         normr0 = MAX(SMALLREAL, r_norm);
00921         relres = r_norm/normr0;
00922         break;
00923     case STOP_REL_PRECRES:
00924         if ( pc == NULL )
00925             fasp_darray_cp(n, p[0], r);
00926         else
00927             pc->fct(p[0], r, pc->data);
00928         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
00929         normr0 = MAX(SMALLREAL, r_normb);
00930         relres = r_normb/normr0;
00931         break;
00932     case STOP_MOD_REL_RES:
00933         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
00934         normr0 = r_norm;
00935         relres = normr0/normu;
00936         break;
00937     default:
00938         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
00939         goto FINISHED;
00940     }
00941
00942     // if initial residual is small, no need to iterate!
00943     if ( relres < tol ) goto FINISHED;
00944
00945     // output iteration information if needed
00946     fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
00947
00948     // store initial residual
00949     norms[0] = relres;
00950
00951     /* outer iteration cycle */
00952     while ( iter < MaxIt ) {
00953
00954         rs[0] = r_norm_old = r_norm;
00955
00956         t = 1.0 / r_norm;
00957
00958         fasp_blas_darray_ax(n, t, p[0]);
00959
00960         //-----//
00961         // adjust the restart parameter //
00962         //-----//
00963         if ( cr > cr_max || iter == 0 ) {
00964             Restart = restart_max;
00965         }

```

```

00966     else if ( cr < cr_min ) {
00967         // Restart = Restart;
00968     }
00969     else {
00970         if ( Restart - d > restart_min ) {
00971             Restart -= d;
00972         }
00973         else {
00974             Restart = restart_max;
00975         }
00976     }
00977
00978     /* RESTART CYCLE (right-preconditioning) */
00979     i = 0;
00980     while ( i < Restart && iter < MaxIt ) {
00981         i++; iter++;
00982
00983         /* apply preconditioner */
00984         if (pc == NULL)
00985             fasp_darray_cp(n, p[i-1], r);
00986         else
00987             pc->fct(p[i-1], r, pc->data);
00988
00989         fasp_blas_dble_mxv(A, r, p[i]);
00990
00991         /* modified Gram_Schmidt */
00992         for (j = 0; j < i; j++) {
00993             hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
00994             fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
00995         }
00996         t = fasp_blas_darray_norm2(n, p[i]);
00997         hh[i][i-1] = t;
00998         if (t != 0.0) {
00999             t = 1.0/t;
01000             fasp_blas_darray_ax(n, t, p[i]);
01001         }
01002
01003         for (j = 1; j < i; ++j) {
01004             t = hh[j-1][i-1];
01005             hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01006             hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01007         }
01008         t = hh[i][i-1]*hh[i][i-1];
01009         t += hh[i-1][i-1]*hh[i-1][i-1];
01010
01011         gamma = sqrt(t);
01012         if (gamma == 0.0) gamma = epsmac;
01013         c[i-1] = hh[i-1][i-1] / gamma;
01014         s[i-1] = hh[i][i-1] / gamma;
01015         rs[i] = -s[i-1]*rs[i-1];
01016         rs[i-1] = c[i-1]*rs[i-1];
01017         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01018
01019         absres = r_norm = fabs(rs[i]);
01020
01021         relres = absres/normr0;
01022
01023         norms[iter] = relres;
01024
01025         /* output iteration information if needed
01026         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
01027             norms[iter]/norms[iter-1]);
01028
01029         // should we exit restart cycle
01030         if ( relres <= tol && iter >= MIN_ITER ) break;
01031
01032     } /* end of restart cycle */
01033
01034     /* now compute solution, first solve upper triangular system */
01035     rs[i-1] = rs[i-1] / hh[i-1][i-1];
01036     for (k = i-2; k >= 0; k--) {
01037         t = 0.0;
01038         for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];
01039
01040         t += rs[k];
01041         rs[k] = t / hh[k][k];
01042     }
01043
01044     fasp_darray_cp(n, p[i-1], w);
01045
01046

```

```

01047     fasp_blas_darray_ax(n, rs[i-1], w);
01048
01049     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01050
01051     /* apply preconditioner */
01052     if ( pc == NULL )
01053         fasp_darray_cp(n, w, r);
01054     else
01055         pc->fct(w, r, pc->data);
01056
01057     fasp_blas_darray_axpy(n, 1.0, r, x->val);
01058
01059     // safety net check: save the best-so-far solution
01060     if ( fasp_dvec_isnan(x) ) {
01061         // If the solution is NAN, restore the best solution
01062         absres = BIGREAL;
01063         goto RESTORE_BESTSOL;
01064     }
01065
01066     if ( absres < absres_best - maxdiff ) {
01067         absres_best = absres;
01068         iter_best = iter;
01069         fasp_darray_cp(n, x->val, x_best);
01070     }
01071
01072     // Check: prevent false convergence
01073     if ( relres <= tol && iter >= MIN_ITER ) {
01074
01075         fasp_darray_cp(n, b->val, r);
01076         fasp_blas_dblc_aApy(-1.0, A, x->val, r);
01077
01078         r_norm = fasp_blas_darray_norm2(n, r);
01079
01080         switch ( StopType ) {
01081             case STOP_REL_RES:
01082                 absres = r_norm;
01083                 relres = absres/normr0;
01084                 break;
01085             case STOP_REL_PRECRES:
01086                 if ( pc == NULL )
01087                     fasp_darray_cp(n, r, w);
01088                 else
01089                     pc->fct(r, w, pc->data);
01090                 absres = sqrt(fasp_blas_darray_dotprod(n, w, r));
01091                 relres = absres/normr0;
01092                 break;
01093             case STOP_MOD_REL_RES:
01094                 absres = r_norm;
01095                 normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
01096                 relres = absres/normu;
01097                 break;
01098         }
01099
01100         norms[iter] = relres;
01101
01102         if ( relres <= tol ) {
01103             break;
01104         }
01105         else {
01106             // Need to restart
01107             fasp_darray_cp(n, r, p[0]); i = 0;
01108         }
01109     }
01110     /* end of convergence check */
01111
01112     /* compute residual vector and continue loop */
01113     for ( j = i; j > 0; j-- ) {
01114         rs[j-1] = -s[j-1]*rs[j];
01115         rs[j] = c[j-1]*rs[j];
01116     }
01117
01118     if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01119
01120     for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01121
01122     if ( i ) {
01123         fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01124         fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01125     }
01126
01127     //-----//

```

```

01128         // compute the convergence rate //
01129         //-----//
01130         cr = r_norm / r_norm_old;
01131
01132     } /* end of iteration while loop */
01133
01134 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01135     if ( iter != iter_best ) {
01136
01137         // compute best residual
01138         fasp_darray_cp(n,b->val,r);
01139         fasp_blas_dblc_aAxy(-1.0,A,x_best,r);
01140
01141         switch ( StopType ) {
01142             case STOP_REL_RES:
01143                 absres_best = fasp_blas_darray_norm2(n,r);
01144                 break;
01145             case STOP_REL_PRECRES:
01146                 // z = B(r)
01147                 if ( pc != NULL )
01148                     pc->fct(r,w,pc->data); /* Apply preconditioner */
01149                 else
01150                     fasp_darray_cp(n,r,w); /* No preconditioner */
01151                 absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
01152                 break;
01153             case STOP_MOD_REL_RES:
01154                 absres_best = fasp_blas_darray_norm2(n,r);
01155                 break;
01156         }
01157
01158         if ( absres > absres_best + maxdiff || isnan(absres) ) {
01159             if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01160             fasp_darray_cp(n,x_best,x->val);
01161             relres = absres_best / normr0;
01162         }
01163     }
01164
01165 FINISHED:
01166     if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);
01167
01168     /*-----
01169     * Free some stuff
01170     *-----*/
01171     fasp_mem_free(work); work = NULL;
01172     fasp_mem_free(p); p = NULL;
01173     fasp_mem_free(hh); hh = NULL;
01174     fasp_mem_free(norms); norms = NULL;
01175
01176 #if DEBUG_MODE > 0
01177     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
01178 #endif
01179
01180     if (iter >= MaxIt)
01181         return ERROR_SOLVER_MAXIT;
01182     else
01183         return iter;
01184 }
01185
01210 INT fasp_solver_dstr_spvgmres (const dSTRmat *A,
01211                               const dvector *b,
01212                               dvector *x,
01213                               precondition *pc,
01214                               const REAL tol,
01215                               const INT MaxIt,
01216                               SHORT restart,
01217                               const SHORT StopType,
01218                               const SHORT PrtLvl)
01219 {
01220     const INT n = b->row;
01221     const INT MIN_ITER = 0;
01222     const REAL maxdiff = tol*STAG_RATIO; // stagnation tolerance
01223     const REAL epsmac = SMALLREAL;
01224
01225     //-----//
01226     // Newly added parameters to monitor when //
01227     // to change the restart parameter //
01228     //-----//
01229     const REAL cr_max = 0.99; // = cos(8^o) (experimental)
01230     const REAL cr_min = 0.174; // = cos(80^o) (experimental)
01231
01232     // local variables

```

```

01233     INT      iter          = 0;
01234     INT      restart1     = restart + 1;
01235     int       i, j, k; // must be signed! -zcs
01236
01237     REAL      r_norm, r_normb, gamma, t;
01238     REAL      normr0 = BIGREAL, absres = BIGREAL;
01239     REAL      relres = BIGREAL, normu = BIGREAL;
01240
01241     REAL      cr           = 1.0; // convergence rate
01242     REAL      r_norm_old = 0.0; // save residual norm of previous restart cycle
01243     INT      d             = 3; // reduction for restart parameter
01244     INT      restart_max = restart; // upper bound for restart in each restart cycle
01245     INT      restart_min = 3; // lower bound for restart (should be small)
01246     INT      Restart      = restart; // real restart in some fixed restarted cycle
01247
01248     INT      iter_best = 0; // initial best known iteration
01249     REAL      absres_best = BIGREAL; // initial best known residual
01250
01251     // allocate temp memory (need about (restart+4)*n REAL numbers)
01252     REAL      *c = NULL, *s = NULL, *rs = NULL;
01253     REAL      *norms = NULL, *r = NULL, *w = NULL;
01254     REAL      *work = NULL, *x_best = NULL;
01255     REAL      **p = NULL, **hh = NULL;
01256
01257     // Output some info for debugging
01258     if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (STR) ...\n");
01259
01260 #if DEBUG_MODE > 0
01261     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01262     printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
01263 #endif
01264
01265     /* allocate memory and setup temp work space */
01266     work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
01267
01268     /* check whether memory is enough for GMRES */
01269     while ( (work == NULL) && (restart > 5) ) {
01270         restart = restart - 5;
01271         work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
01272         printf("### WARNING: vGMRES restart number set to %d!\n", restart);
01273         restart1 = restart + 1;
01274     }
01275
01276     if ( work == NULL ) {
01277         printf("### ERROR: No enough memory! [%s:%d]\n", __FILE__, __LINE__);
01278         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
01279     }
01280
01281     p = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
01282     hh = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
01283     norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));
01284
01285     r = work; w = r + n; rs = w + n; c = rs + restart1;
01286     x_best = c + restart; s = x_best + n;
01287
01288     for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;
01289
01290     for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;
01291
01292     // r = b-A*x
01293     fasp_darray_cp(n, b->val, p[0]);
01294     fasp_blas_dstx_aAxy(-1.0, A, x->val, p[0]);
01295
01296     r_norm = fasp_blas_darray_norm2(n, p[0]);
01297
01298     // compute initial residuals
01299     switch (StopType) {
01300     case STOP_REL_RES:
01301         normr0 = MAX(SMALLREAL, r_norm);
01302         relres = r_norm/normr0;
01303         break;
01304     case STOP_REL_PRECRES:
01305         if ( pc == NULL )
01306             fasp_darray_cp(n, p[0], r);
01307         else
01308             pc->fct(p[0], r, pc->data);
01309         r_normb = sqrt(fasp_blas_darray_dotprod(n, p[0], r));
01310         normr0 = MAX(SMALLREAL, r_normb);
01311         relres = r_normb/normr0;
01312         break;
01313     case STOP_MOD_REL_RES:

```

```

01314         normu   = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
01315         normr0  = r_norm;
01316         relres  = normr0/normu;
01317         break;
01318     default:
01319         printf("### ERROR: Unknown stopping type! [%s]\n", __FUNCTION__);
01320         goto FINISHED;
01321     }
01322
01323     // if initial residual is small, no need to iterate!
01324     if ( relres < tol ) goto FINISHED;
01325
01326     // output iteration information if needed
01327     fasp_itinfo(PrtLvl, StopType, 0, relres, normr0, 0.0);
01328
01329     // store initial residual
01330     norms[0] = relres;
01331
01332     /* outer iteration cycle */
01333     while ( iter < MaxIt ) {
01334
01335         rs[0] = r_norm_old = r_norm;
01336
01337         t = 1.0 / r_norm;
01338
01339         fasp_blas_darray_ax(n, t, p[0]);
01340
01341         //-----//
01342         //   adjust the restart parameter   //
01343         //-----//
01344         if ( cr > cr_max || iter == 0 ) {
01345             Restart = restart_max;
01346         }
01347         else if ( cr < cr_min ) {
01348             // Restart = Restart;
01349         }
01350         else {
01351             if ( Restart - d > restart_min ) {
01352                 Restart -= d;
01353             }
01354             else {
01355                 Restart = restart_max;
01356             }
01357         }
01358
01359         /* RESTART CYCLE (right-preconditioning) */
01360         i = 0;
01361         while ( i < Restart && iter < MaxIt ) {
01362
01363             i++; iter++;
01364
01365             /* apply preconditioner */
01366             if (pc == NULL)
01367                 fasp_darray_cp(n, p[i-1], r);
01368             else
01369                 pc->fct(p[i-1], r, pc->data);
01370
01371             fasp_blas_dstr_mmv(A, r, p[i]);
01372
01373             /* modified Gram_Schmidt */
01374             for (j = 0; j < i; j++) {
01375                 hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
01376                 fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
01377             }
01378             t = fasp_blas_darray_norm2(n, p[i]);
01379             hh[i][i-1] = t;
01380             if (t != 0.0) {
01381                 t = 1.0/t;
01382                 fasp_blas_darray_ax(n, t, p[i]);
01383             }
01384
01385             for (j = 1; j < i; ++j) {
01386                 t = hh[j-1][i-1];
01387                 hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
01388                 hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
01389             }
01390             t = hh[i][i-1]*hh[i][i-1];
01391             t += hh[i-1][i-1]*hh[i-1][i-1];
01392
01393             gamma = sqrt(t);
01394             if (gamma == 0.0) gamma = epsmac;

```



```

01395         c[i-1] = hh[i-1][i-1] / gamma;
01396         s[i-1] = hh[i][i-1] / gamma;
01397         rs[i] = -s[i-1]*rs[i-1];
01398         rs[i-1] = c[i-1]*rs[i-1];
01399         hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
01400
01401         absres = r_norm = fabs(rs[i]);
01402
01403         relres = absres/normr0;
01404
01405         norms[iter] = relres;
01406
01407         // output iteration information if needed
01408         fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
01409                   norms[iter]/norms[iter-1]);
01410
01411         // should we exit restart cycle
01412         if ( relres <= tol && iter >= MIN_ITER ) break;
01413
01414     } /* end of restart cycle */
01415
01416     /* now compute solution, first solve upper triangular system */
01417     rs[i-1] = rs[i-1] / hh[i-1][i-1];
01418     for (k = i-2; k >= 0; k --) {
01419         t = 0.0;
01420         for (j = k+1; j < i; j ++ ) t -= hh[k][j]*rs[j];
01421
01422         t += rs[k];
01423         rs[k] = t / hh[k][k];
01424     }
01425
01426     fasp_darray_cp(n, p[i-1], w);
01427
01428     fasp_blas_darray_ax(n, rs[i-1], w);
01429
01430     for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);
01431
01432     /* apply preconditioner */
01433     if ( pc == NULL )
01434         fasp_darray_cp(n, w, r);
01435     else
01436         pc->fct(w, r, pc->data);
01437
01438     fasp_blas_darray_axpy(n, 1.0, r, x->val);
01439
01440     // safety net check: save the best-so-far solution
01441     if ( fasp_dvec_isnan(x) ) {
01442         // If the solution is NAN, restore the best solution
01443         absres = BIGREAL;
01444         goto RESTORE_BESTSOL;
01445     }
01446
01447     if ( absres < absres_best - maxdiff ) {
01448         absres_best = absres;
01449         iter_best = iter;
01450         fasp_darray_cp(n, x->val, x_best);
01451     }
01452
01453     // Check: prevent false convergence
01454     if ( relres <= tol && iter >= MIN_ITER ) {
01455
01456         fasp_darray_cp(n, b->val, r);
01457         fasp_blas_dstr_aApy(-1.0, A, x->val, r);
01458
01459         r_norm = fasp_blas_darray_norm2(n, r);
01460
01461         switch ( StopType ) {
01462             case STOP_REL_RES:
01463                 absres = r_norm;
01464                 relres = absres/normr0;
01465                 break;
01466             case STOP_REL_PRECRES:
01467                 if ( pc == NULL )
01468                     fasp_darray_cp(n, r, w);
01469                 else
01470                     pc->fct(r, w, pc->data);
01471                 absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
01472                 relres = absres/normr0;
01473                 break;
01474             case STOP_MOD_REL_RES:
01475                 absres = r_norm;

```

```

01476         normu = MAX(SMALLREAL, fasp_blas_darray_norm2(n, x->val));
01477         relres = absres/normu;
01478         break;
01479     }
01480
01481     norms[iter] = relres;
01482
01483     if ( relres <= tol ) {
01484         break;
01485     }
01486     else {
01487         // Need to restart
01488         fasp_darray_cp(n, r, p[0]); i = 0;
01489     }
01490
01491 } /* end of convergence check */
01492
01493 /* compute residual vector and continue loop */
01494 for ( j = i; j > 0; j-- ) {
01495     rs[j-1] = -s[j-1]*rs[j];
01496     rs[j] = c[j-1]*rs[j];
01497 }
01498
01499 if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);
01500
01501 for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);
01502
01503 if ( i ) {
01504     fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
01505     fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
01506 }
01507
01508 //-----//
01509 // compute the convergence rate //
01510 //-----//
01511 cr = r_norm / r_norm_old;
01512
01513 } /* end of iteration while loop */
01514
01515 RESTORE_BESTSOL: // restore the best-so-far solution if necessary
01516 if ( iter != iter_best ) {
01517
01518     // compute best residual
01519     fasp_darray_cp(n, b->val, r);
01520     fasp_blas_dstr_aAxy(-1.0, A, x_best, r);
01521
01522     switch ( StopType ) {
01523     case STOP_REL_RES:
01524         absres_best = fasp_blas_darray_norm2(n, r);
01525         break;
01526     case STOP_REL_PRECRES:
01527         // z = B(r)
01528         if ( pc != NULL )
01529             pc->fct(r, w, pc->data); /* Apply preconditioner */
01530         else
01531             fasp_darray_cp(n, r, w); /* No preconditioner */
01532         absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n, w, r)));
01533         break;
01534     case STOP_MOD_REL_RES:
01535         absres_best = fasp_blas_darray_norm2(n, r);
01536         break;
01537     }
01538
01539     if ( absres > absres_best + maxdiff || isnan(absres) ) {
01540         if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
01541         fasp_darray_cp(n, x_best, x->val);
01542         relres = absres_best / normr0;
01543     }
01544 }
01545
01546 FINISHED:
01547 if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter, MaxIt, relres);
01548
01549 /*-----
01550 * Free some stuff
01551 *-----*/
01552 fasp_mem_free(work); work = NULL;
01553 fasp_mem_free(p); p = NULL;
01554 fasp_mem_free(hh); hh = NULL;
01555 fasp_mem_free(norms); norms = NULL;
01556

```

```

01557 #if DEBUG_MODE > 0
01558     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01559 #endif
01560
01561     if (iter>=MaxIt)
01562         return ERROR_SOLVER_MAXIT;
01563     else
01564         return iter;
01565 }
01566
01567 /*-----*/
01568 /*--      End of File      --*/
01569 /*-----*/

```

9.135 PreAMGCoarsenCR.c File Reference

Coarsening with Brannick-Falgout strategy.

```

#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreAMGUtil.inl"

```

Functions

- [INT fasp_amg_coarsening_cr](#) (const [INT](#) i_0, const [INT](#) i_n, [dCSRmat](#) *A, [ivector](#) *vertices, [AMG_param](#) *param)
CR coarsening.

9.135.1 Detailed Description

Coarsening with Brannick-Falgout strategy.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxMemory.c](#), [AuxThreads.c](#), and [ltrSmootherCSRcr.c](#)

Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

// TODO: Not completed! –Chensong
Definition in file [PreAMGCoarsenCR.c](#).

9.135.2 Macro Definition Documentation

9.135.2.1 AMG_COARSEN_CR

```
#define AMG_COARSEN_CR
```

Definition at line 25 of file [PreAMGCoarsenCR.c](#).

9.135.3 Function Documentation

9.135.3.1 fasp_amg_coarsening_cr()

```
INT fasp_amg_coarsening_cr (
    const INT i_0,
    const INT i_n,
    dCSRmat * A,
    ivector * vertices,
    AMG_param * param )
```

CR coarsening.

Parameters

<i>i_0</i>	Starting index
<i>i_n</i>	Ending index
<i>A</i>	Pointer to dCSRmat : the coefficient matrix (index starts from 0)
<i>vertices</i>	Pointer to CF, 0: Fpt (current level) or 1: Cpt
<i>param</i>	Pointer to AMG_param : AMG parameters

Returns

Number of coarse level points

Author

James Brannick

Date

04/21/2010

Note

vertices = 0: fine; 1: coarse; 2: isolated or special

Modified by Chunsheng Feng, Zheng Li on 10/14/2012 CR STAGES

Definition at line 62 of file [PreAMGCoarsenCR.c](#).

9.136 PreAMGCoarsenCR.c

[Go to the documentation of this file.](#)

```
00001
00016 #include <math.h>
00017
00018 #ifdef _OPENMP
00019 #include <omp.h>
00020 #endif
00021
00022 #include "fasp.h"
00023 #include "fasp_funcs.h"
00024
00025 #define AMG_COARSEN_CR
00026
00027 /*-----*/
00028 /*--  Declare Private Functions  --*/
00029 /*-----*/
00030
00031 #include "PreAMGUtil.inl"
00032
00033 static INT GraphAdd(Link *, INT *, INT *, INT, INT);
00034 static INT GraphRemove(Link *, INT *, INT *, INT );
00035 static INT indset(INT, INT, INT, INT *, INT *, INT, INT *, REAL *);
```

```

00036
00037 /*-----*/
00038 /*--      Public Functions      --*/
00039 /*-----*/
00040
00062 INT fasp_amg_coarsening_cr (const INT    i_0,
00063                           const INT    i_n,
00064                           dCSRmat      *A,
00065                           ivector      *vertices,
00066                           AMG_param    *param)
00067 {
00068     const SHORT prtlvl = param->print_level;
00069
00070     // local variables
00071     INT cand=0, cpt=-1, fpt=1;           // internal labeling
00072     INT nc, ns=1;                       // # cpts, # stages
00073     INT i, j, inl, nu=3, numl = nu-1;    // nu is number of cr sweeps
00074     INT *cf=NULL, *ia=NULL, *ja=NULL;
00075
00076     REAL temp0=0.0e0, temp1=0.0e0, rho=0.0e0, tg=8.0e-01;
00077     REAL *a=NULL;
00078
00079     /* WORKING MEMORY -- b not needed, remove later */
00080     REAL *b=NULL, *u=NULL, *ma=NULL;
00081
00082     ia = A->IA;
00083     ja = A->JA;
00084     a  = A->val;
00085
00086     if (i_0 == 0) {
00087         inl = i_n+1;
00088     } else {
00089         inl = i_n;
00090     }
00091
00092     /* CF, RHS, INITIAL GUESS, and MEAS. ARRAY */
00093     cf = (INT*)fasp_mem_calloc(inl, sizeof(INT));
00094     b  = (REAL*)fasp_mem_calloc(inl, sizeof(REAL));
00095     u  = (REAL*)fasp_mem_calloc(inl, sizeof(REAL));
00096     ma = (REAL*)fasp_mem_calloc(inl, sizeof(REAL));
00097
00098 #ifdef _OPENMP
00099 #pragma omp parallel for if(i_n>OPENMP_HOLDS)
00100 #endif
00101     for(i=i_0; i<=i_n; ++i) {
00102         b[i] = 0.0e0; // ZERO RHS
00103         cf[i] = fpt;  // ALL FPTs
00104     }
00105
00107     while (TRUE) {
00108
00109         nc = 0;
00110 #ifdef _OPENMP
00111 #pragma omp parallel for if(i_n>OPENMP_HOLDS)
00112 #endif
00113         for(i=i_0; i<=i_n; ++i) {
00114             if (cf[i] == cpt) {
00115                 nc += 1;
00116                 u[i] = 0.0e0;
00117             } else {
00118                 u[i] = 1.0e0;
00119             }
00120         }
00121
00122         for (i=i_0; i<=nu; ++i) {
00123
00124             if (i == numl)
00125                 for (j = i_0; j<= i_n; ++j) {
00126                     if (cf[j] == fpt) {
00127                         temp0 += u[j]*u[j];
00128                     }
00129                 }
00130             fasp_smoother_dcsr_gscr(fpt, i_n, u, ia, ja, a, b, 1, cf);
00131         }
00132
00133 #ifdef _OPENMP
00134 #pragma omp parallel for reduction(+:temp1) if(i_n>OPENMP_HOLDS)
00135 #endif
00136         for (i = i_0; i<= i_n; ++i) {
00137             if (cf[i] == fpt) {
00138                 temp1 += u[i]*u[i];

```

```

00139     }
00140 }
00141 rho = sqrt(temp1)/sqrt(temp0);
00142
00143 if ( prtlvl > PRINT_MIN ) printf("rho=%2.13lf\n",rho);
00144
00145 if ( rho > tg ) {
00146     /* FORM CAND. SET & COMPUTE IND SET */
00147     temp0 = 0.0e0;
00148
00149     for (i = i_0; i<= i_n; ++i) {
00150         temp1 = fabs(u[i]);
00151         if (cf[i] == cpt && temp1 > 0.0e0) {
00152             temp0 = temp1; // max.
00153         }
00154     }
00155     if (ns == 1) {
00156         temp1 = pow(0.3, nu);
00157     } else {
00158         temp1 = 0.5;
00159     }
00160
00161 #ifdef _OPENMP
00162 #pragma omp parallel for if(i_n>OPENMP_HOLDS)
00163 #endif
00164     for (i = i_0; i <= i_n; ++i) {
00165         if (cf[i] == fpt && fabs(u[i])/temp0 > temp1 && ia[i+1]-ia[i] > 1)
00166             cf[i] = cand;
00167     }
00168     temp1 = 0.0e0;
00169     indset(cand,cpt,fpt,ia,ja,i_n,cf,ma);
00170     ns++;
00171 }
00172 else {
00173     /* back to fasp labeling */
00174
00175 #ifdef _OPENMP
00176 #pragma omp parallel for if(i_n>OPENMP_HOLDS)
00177 #endif
00178     for (i = i_0; i<= i_n; ++i) {
00179         if (cf[i] == cpt) {
00180             cf[i] = 1; // cpt
00181         } else {
00182             cf[i] = 0; // fpt
00183         }
00184         // printf("cf[%i] = %i\n",i,cf[i]);
00185     }
00186     vertices->row=i_n;
00187     if ( prtlvl >= PRINT_MORE ) printf("vertices = %i\n",vertices->row);
00188     vertices->val= cf;
00189     if ( prtlvl >= PRINT_MORE ) printf("nc=%i\n",nc);
00190     break;
00191 }
00192 }
00193
00194 fasp_mem_free(u); u = NULL;
00195 fasp_mem_free(b); b = NULL;
00196 fasp_mem_free(ma); ma = NULL;
00197
00198 return nc;
00199 }
00200
00201 /*-----*/
00202 /*--      Private Functions      --*/
00203 /*-----*/
00204
00209 static INT GraphAdd (Link *list,
00210                     INT *head,
00211                     INT *tail,
00212                     INT index,
00213                     INT istack)
00214 {
00215     INT prev = tail[-istack];
00216
00217     list[index].prev = prev;
00218     if (prev < 0)
00219         head[-istack] = index;
00220     else
00221         list[prev].next = index;
00222     list[index].next = -istack;
00223     tail[-istack] = index;

```

```

00224
00225     return 0;
00226 }
00227
00232 static INT GraphRemove (Link  *list,
00233                          INT   *head,
00234                          INT   *tail,
00235                          INT   index)
00236 {
00237     INT prev = list[index].prev;
00238     INT next = list[index].next;
00239
00240     if (prev < 0)
00241         head[prev] = next;
00242     else
00243         list[prev].next = next;
00244     if (next < 0)
00245         tail[next] = prev;
00246     else
00247         list[next].prev = prev;
00248
00249     return 0;
00250 }
00251
00272 static INT indset (INT   cand,
00273                   INT   cpt,
00274                   INT   fpt,
00275                   INT   *ia,
00276                   INT   *ja,
00277                   INT   n,
00278                   INT   *cf,
00279                   REAL   *ma)
00280 {
00281     /* ma: candidates >= 1, cpts = -1, otherwise = 0
00282     * Note: graph contains candidates only */
00283
00284     Link *list;
00285     INT *head, *head_mem;
00286     INT *tail, *tail_mem;
00287
00288     INT i, ji, jj, jl, index, istack, stack_size;
00289
00290     for (istack = i = 0; i < n; ++i) {
00291
00292         if (cf[i] == cand) {
00293             ma[i] = 1;
00294             for (ji = ia[i]+1; ji < ia[i+1]; ++ji) {
00295                 jj = ja[ji];
00296                 if (cf[jj] != cpt) {
00297                     ma[i]++;
00298                 }
00299             }
00300
00301             if (ma[i] > istack) {
00302                 istack = (INT) ma[i];
00303             }
00304
00305             else if (cf[i] == cpt) {
00306                 ma[i] = -1;
00307             }
00308             else {
00309                 ma[i] = 0;
00310             }
00311         }
00312
00313         stack_size = 2*istack;
00314
00315         /* INITIALIZE GRAPH */
00316         list = (Link*) fasp_mem_malloc(n, sizeof(Link));
00317         head_mem = (INT*) fasp_mem_malloc(stack_size, sizeof(INT));
00318         tail_mem = (INT*) fasp_mem_malloc(stack_size, sizeof(INT));
00319         head = head_mem + stack_size;
00320         tail = tail_mem + stack_size;
00321
00322 #ifdef _OPENMP
00323 #pragma omp parallel for if(stack_size>OPENMP_HOLDS)
00324 #endif
00325         for (i = -1; i >= -stack_size; i--) {
00326             head[i] = i;
00327             tail[i] = i;
00328         }

```

```

00329
00330 #ifndef _OPENMP
00331 #pragma omp parallel for if(stack_size>OPENMP_HOLDS)
00332 #endif
00333     for (i = 0; i < n; ++i) {
00334         if (ma[i] > 0) {
00335             GraphAdd(list, head, tail, i, (INT) ma[i]);
00336         }
00337     }
00338
00339     while (istack > 0) {
00340         /* i with maximal measure is at the head of the stacks */
00341         i = head[-istack];
00342         /* make i a c-point */
00343         cf[i] = cpt;
00344         ma[i] = -1;
00345         /* remove i from graph */
00346         GraphRemove(list, head, tail, i);
00347
00348         /* update neighbors and neighbors-of-neighbors */
00349         for (ji = ia[i]+1; ji < ia[i+1]; ++ji) {
00350             jj = ja[ji];
00351             /* if not "decided" c or f */
00352             if (ma[jj] > -1) {
00353                 /* if a candidate, remove jj from graph */
00354                 if (ma[jj] > 0) {
00355                     GraphRemove(list, head, tail, jj);
00356                 }
00357                 /* make jj an f-point and mark "decided" */
00358                 cf[jj] = fpt;
00359                 ma[jj] = -1;
00360
00361                 for (jl = ia[jj]+1; jl < ia[jj+1]; jl++) {
00362                     index = ja[jl];
00363                     /* if a candidate, increase likelihood of being chosen */
00364                     if (ma[index] > 0) {
00365                         ma[index]++;
00366                         /* move index in graph */
00367                         GraphRemove(list, head, tail, index);
00368                         GraphAdd(list, head, tail, index, (INT) ma[index]);
00369                         if (ma[index] > istack) {
00370                             istack = (INT) ma[index];
00371                         }
00372                     }
00373                 }
00374             }
00375         }
00376
00377         /* reset istack to point to the biggest non-empty stack */
00378         for (; istack > 0; istack--) {
00379             /* if non-negative, break */
00380             if (head[-istack] > -1) {
00381                 break;
00382             }
00383         }
00384     }
00385
00386     fasp_mem_free(list);    list      = NULL;
00387     fasp_mem_free(head_mem); head_mem = NULL;
00388     fasp_mem_free(tail_mem); tail_mem = NULL;
00389
00390     return 0;
00391 }
00392
00393 /*-----*/
00394 /*--          End of File          --*/
00395 /*-----*/

```

9.137 PreAMGCoarsenRS.c File Reference

Coarsening with a modified Ruge-Stuben strategy.

```

#include "fasp.h"
#include "fasp_funcs.h"
#include "PreAMGUtil.inl"

```


Functions

- [SHORT fasp_amg_coarsening_rs](#) ([dCSRmat](#) *A, [ivector](#) *vertices, [dCSRmat](#) *P, [iCSRmat](#) *S, [AMG_param](#) *param)

Standard and aggressive coarsening schemes.

9.137.1 Detailed Description

Coarsening with a modified Ruge-Stuben strategy.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), [AuxVector.c](#), [BlaSparseCSR.c](#), and [PreAMGCoarsenCR.c](#)

Reference: Multigrid by U. Trottenberg, C. W. Oosterlee and A. Schuller Appendix P475 A.7 (by A. Brandt, P. Oswald and K. Stuben) Academic Press Inc., San Diego, CA, 2001.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGCoarsenRS.c](#).

9.137.2 Function Documentation

9.137.2.1 fasp_amg_coarsening_rs()

```
SHORT fasp_amg_coarsening_rs (
    dCSRmat * A,
    ivector * vertices,
    dCSRmat * P,
    iCSRmat * S,
    AMG_param * param )
```

Standard and aggressive coarsening schemes.

Parameters

<i>A</i>	Pointer to dCSRmat : Coefficient matrix (index starts from 0)
<i>vertices</i>	Indicator vector for the C/F splitting of the variables
<i>P</i>	Interpolation matrix (nonzero pattern only)
<i>S</i>	Strong connection matrix
<i>param</i>	Pointer to AMG_param : AMG parameters

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xuehai Huang, Chensong Zhang, Xiaozhe Hu, Ludmil Zikatanov

Date

09/06/2010

Note

vertices = 0: fine; 1: coarse; 2: isolated or special

Modified by Xiaozhe Hu on 05/23/2011: add strength matrix as an argument Modified by Xiaozhe Hu on 04/24/2013: modify aggressive coarsening Modified by Chensong Zhang on 04/28/2013: remove linked list Modified by Chensong Zhang on 05/11/2013: restructure the code
 Definition at line 73 of file [PreAMGCoarsenRS.c](#).

9.138 PreAMGCoarsenRS.c

[Go to the documentation of this file.](#)

```

00001
00020 #ifdef _OPENMP
00021 #include <omp.h>
00022 #endif
00023
00024 #include "fasp.h"
00025 #include "fasp_functs.h"
00026
00027 /*-----*/
00028 /*--  Declare Private Functions  --*/
00029 /*-----*/
00030
00031 #include "PreAMGUtil.inl"
00032
00033 static INT cfsplitting_cls      (dCSRmat *, iCSRmat *, ivector *);
00034 static INT cfsplitting_clsp    (dCSRmat *, iCSRmat *, ivector *);
00035 static INT cfsplitting_agg     (dCSRmat *, iCSRmat *, ivector *, INT);
00036 static INT cfsplitting_mis     (iCSRmat *, ivector *, ivector *);
00037 static INT clean_ff_couplings  (iCSRmat *, ivector *, INT, INT);
00038 static INT compress_S          (iCSRmat *);
00039
00040 static void strong_couplings    (dCSRmat *, iCSRmat *, AMG_param *);
00041 static void form_P_pattern_dir  (dCSRmat *, iCSRmat *, ivector *, INT, INT);
00042 static void form_P_pattern_std  (dCSRmat *, iCSRmat *, ivector *, INT, INT);
00043 static void ordering1           (iCSRmat *, ivector *);
00044
00045 /*-----*/
00046 /*--      Public Functions      --*/
00047 /*-----*/
00048
00073 SHORT fasp_amg_coarsening_rs (dCSRmat      *A,
00074                               ivector      *vertices,
00075                               dCSRmat      *P,
00076                               iCSRmat      *S,
00077                               AMG_param    *param)
00078 {
00079     const SHORT coarse_type = param->coarsening_type;
00080     const INT   agg_path    = param->aggressive_path;
00081     const INT   row         = A->row;
00082
00083     // local variables
00084     SHORT      interp_type = param->interpolation_type;
00085     INT        col         = 0;
00086
00087     #if DEBUG_MODE > 0
00088         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00089     #endif
00090
00091     #if DEBUG_MODE > 1
00092         printf("### DEBUG: Step 1. Find strong connections ..... \n");
00093     #endif
00094
00095     // make sure standard interp is used for aggressive coarsening
00096     if ( coarse_type == COARSE_AC ) interp_type = INTERP_STD;
00097
00098     // find strong couplings and return them in S
00099     strong_couplings(A, S, param);
00100

```

```

00101 #if DEBUG_MODE > 1
00102     printf("### DEBUG: Step 2. C/F splitting .....\\n");
00103 #endif
00104     switch ( coarse_type ) {
00105     case COARSE_RSP: // Classical coarsening with positive connections
00106         col = cfsplitting_clsp(A, S, vertices); break;
00107     case COARSE_AC: // Aggressive coarsening
00108         col = cfsplitting_agg(A, S, vertices, agg_path); break;
00109     case COARSE_CR: // Compatible relaxation
00110         col = fasp_amg_coarsening_cr(0, row-1, A, vertices, param); break;
00111     case COARSE_MIS: // Maximal independent set
00112     {
00113         ivector order = fasp_ivec_create(row);
00114         compress_S(S);
00115         ordering1(S, &order);
00116         col = cfsplitting_mis(S, vertices, &order);
00117         fasp_ivec_free(&order);
00118         break;
00119     }
00120     default: // Classical coarsening
00121         col = cfsplitting_cls(A, S, vertices);
00122     }
00123     }
00124     if ( col <= 0 ) return ERROR_UNKNOWN;
00125 #if DEBUG_MODE > 1
00126     printf("### DEBUG: Step 3. Find support of C points .....\\n");
00127 #endif
00128     switch ( interp_type ) {
00129     case INTERP_DIR: // Direct interpolation or ...
00130     case INTERP_ENG: // Energy-min interpolation
00131         col = clean_ff_couplings(S, vertices, row, col);
00132         form_P_pattern_dir(P, S, vertices, row, col);
00133         break;
00134     case INTERP_STD: // Standard interpolation
00135     case INTERP_EXT: // Extended interpolation
00136         form_P_pattern_std(P, S, vertices, row, col); break;
00137     default:
00138         fasp_chkerr(ERROR_AMG_INTERP_TYPE, __FUNCTION__);
00139     }
00140     }
00141     if ( col <= 0 ) return ERROR_UNKNOWN;
00142 #if DEBUG_MODE > 0
00143     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00144 #endif
00145     return FASP_SUCCESS;
00146 }
00147
00148 /*-----*/
00149 /*--      Private Functions      --*/
00150 /*-----*/
00151
00152 static void strong_couplings (dCSRmat *A,
00153                             iCSRmat *S,
00154                             AMG_param *param )
00155 {
00156     const SHORT coarse_type = param->coarsening_type;
00157     const REAL max_row_sum = param->max_row_sum;
00158     const REAL epsilon_str = param->strong_threshold;
00159     const INT row = A->row, col = A->col, rowl = row+1;
00160     const INT nnz = A->nnz;
00161
00162     INT *ia = A->IA, *ja = A->JA;
00163     REAL *aj = A->val;

```

```

00199 // local variables
00200 INT i, j, begin_row, end_row;
00201 REAL row_scl, row_sum;
00202
00203 SHORT nthreads = 1, use_omp = FALSE;
00204
00205 #ifdef _OPENMP
00206 if ( row > OPENMP_HOLDS ) {
00207     use_omp = TRUE;
00208     nthreads = fasp_get_num_threads();
00209 }
00210 #endif
00211
00212 // get the diagonal entry of A: assume all connections are strong
00213 dvector diag; fasp_dcsr_getdiag(0, A, &diag);
00214
00215 // copy the structure of A to S
00216 S->row = row; S->col = col; S->nnz = nnz; S->val = NULL;
00217 S->IA = (INT *)fasp_mem_calloc(row1, sizeof(INT));
00218 S->JA = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
00219 fasp_iarray_cp(row1, ia, S->IA);
00220 fasp_iarray_cp(nnz, ja, S->JA);
00221
00222 if ( use_omp ) {
00223
00224     // This part is still old! Need to be updated. --Chensong 09/18/2016
00225
00226     INT mybegin, myend, myid;
00227 #ifdef _OPENMP
00228 #pragma omp parallel for private(myid, mybegin, myend, i, row_scl, row_sum, begin_row, end_row, j)
00229 #endif
00230     for ( myid = 0; myid < nthreads; myid++ ) {
00231         fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
00232         for ( i = mybegin; i < myend; i++ ) {
00233
00234             // Compute most negative entry in each row and row sum
00235             row_scl = row_sum = 0.0;
00236             begin_row = ia[i]; end_row = ia[i+1];
00237             for ( j = begin_row; j < end_row; j++ ) {
00238                 row_scl = MIN(row_scl, aj[j]);
00239                 row_sum += aj[j];
00240             }
00241
00242             // Find diagonal entries of S and remove them later
00243             for ( j = begin_row; j < end_row; j++ ) {
00244                 if ( ja[j] == i ) { S->JA[j] = -1; break; }
00245             }
00246
00247             // Mark entire row as weak couplings if strongly diagonal-dominant
00248             if ( ABS(row_sum) > max_row_sum * ABS(diag.val[i]) ) {
00249                 for ( j = begin_row; j < end_row; j++ ) S->JA[j] = -1;
00250             }
00251             else {
00252                 for ( j = begin_row; j < end_row; j++ ) {
00253                     // If a_{ij} >= epsilon_{str} * \min a_{ij}, the connection
00254                     // j->i is set to be weak; positive entries result in weak
00255                     // connections
00256                     if ( A->val[j] >= epsilon_str*row_scl ) S->JA[j] = -1;
00257                 }
00258             }
00259         } // end for i
00260     } // end for myid
00261
00262 }
00263
00264 else {
00265     for ( i = 0; i < row; ++i ) {
00266
00267         // Compute row scale and row sum
00268         row_scl = row_sum = 0.0;
00269         begin_row = ia[i]; end_row = ia[i+1];
00270
00271         for ( j = begin_row; j < end_row; j++ ) {
00272
00273             // Originally: Not consider positive entries
00274             // row_sum += aj[j];
00275             // Now changed to --Chensong 05/17/2013
00276             row_sum += ABS(aj[j]);
00277
00278         }
00279     }

```

```

00280         // Originally: Not consider positive entries
00281         // row_scl = MAX(row_scl, -aj[j]); // smallest negative
00282         // Now changed to --Chensong 06/01/2013
00283         if ( ja[j] != i ) row_scl = MAX(row_scl, ABS(aj[j])); // largest abs
00284     }
00285 }
00286
00287 // Multiply by the strength threshold
00288 row_scl *= epsilon_str;
00289
00290 // Find diagonal entries of S and remove them later
00291 for ( j = begin_row; j < end_row; j++ ) {
00292     if ( ja[j] == i ) { S->JA[j] = -1; break; }
00293 }
00294
00295 // Mark entire row as weak couplings if strongly diagonal-dominant
00296 // Originally: Not consider positive entries
00297 // if ( ABS(row_sum) > max_row_sum * ABS(diag.val[i]) ) {
00298 // Now changed to --Chensong 05/17/2013
00299 if ( row_sum < (2 - max_row_sum) * ABS(diag.val[i]) ) {
00300
00301     for ( j = begin_row; j < end_row; j++ ) S->JA[j] = -1;
00302 }
00303 else {
00304     switch ( coarse_type ) {
00305     case COARSE_RSP: // consider positive off-diag as well
00306         for ( j = begin_row; j < end_row; j++ ) {
00307             if ( ABS(A->val[j]) <= row_scl ) S->JA[j] = -1;
00308         }
00309         break;
00310     default: // only consider n-couplings
00311         for ( j = begin_row; j < end_row; j++ ) {
00312             if ( -A->val[j] <= row_scl ) S->JA[j] = -1;
00313         }
00314         break;
00315     }
00316 } // end for i
00317 } // end if openmp
00318 fasp_dvec_free(&diag);
00319 }
00320
00321 static INT compress_S (iCSRmat *S)
00322 {
00323     const INT row = S->row;
00324     INT * ia = S->IA;
00325
00326     // local variables
00327     INT index, i, j, begin_row, end_row;
00328
00329     // compress S: remove weak connections and form strong coupling matrix
00330     for ( index = i = 0; i < row; ++i ) {
00331         begin_row = ia[i]; end_row = ia[i+1];
00332         ia[i] = index;
00333         for ( j = begin_row; j < end_row; j++ ) {
00334             if ( S->JA[j] > -1 ) S->JA[index++] = S->JA[j]; // strong couplings
00335         }
00336     }
00337     S->nnz = S->IA[row] = index;
00338     if ( S->nnz <= 0 ) {
00339         return ERROR_UNKNOWN;
00340     }
00341     else {
00342         return FASP_SUCCESS;
00343     }
00344 }
00345
00346 static void rem_positive_ff (dCSRmat *A,
00347                             iCSRmat *Stemp,

```

```

00387                                     ivector    *vertices)
00388 {
00389     const INT    row = A->row;
00390     INT          *ia  = A->IA, *vec = vertices->val;
00391
00392     REAL         row_scl, max_entry;
00393     INT          i, j, ji, max_index;
00394
00395     for ( i = 0; i < row; ++i ) {
00396
00397         if ( vec[i] != FGPT ) continue; // skip non F-variables
00398
00399         row_scl = 0.0;
00400         for ( ji = ia[i]; ji < ia[i+1]; ++ji ) {
00401             j = A->JA[ji];
00402             if ( j == i ) continue; // skip diagonal
00403             row_scl = MAX(row_scl, ABS(A->val[ji])); // max abs entry
00404         } // end for ji
00405         row_scl *= 0.75;
00406
00407         // looking for strong F-F connections
00408         max_index = -1; max_entry = 0.0;
00409         for ( ji = ia[i]; ji < ia[i+1]; ++ji ) {
00410             j = A->JA[ji];
00411             if ( j == i ) continue; // skip diagonal
00412             if ( vec[j] != FGPT ) continue; // skip F-C connections
00413             if ( A->val[ji] > row_scl ) {
00414                 Stemp->JA[ji] = j;
00415                 if ( A->val[ji] > max_entry ) {
00416                     max_entry = A->val[ji];
00417                     max_index = j; // max positive entry
00418                 }
00419             }
00420         } // end for ji
00421
00422         // mark max positive entry as C-point
00423         if ( max_index != -1 ) vec[max_index] = CGPT;
00424     } // end for i
00425 }
00426
00427 }
00428
00450 static INT cfsplitting_cls (dCSRmat    *A,
00451                             iCSRmat    *S,
00452                             ivector    *vertices)
00453 {
00454     const INT    row = A->row;
00455
00456     // local variables
00457     INT col = 0;
00458     INT maxmeas, maxnode, num_left = 0;
00459     INT measure, newmeas;
00460     INT i, j, k, l;
00461     INT myid, mybegin, myend;
00462     INT *vec = vertices->val;
00463     INT *work = (INT*)fasp_mem_calloc(3*row, sizeof(INT));
00464     INT *lists = work, *where = lists+row, *lambda = where+row;
00465
00466     #if RS_C1
00467         INT set_empty = 1;
00468         INT jkeep = 0, cnt, index;
00469         INT row_end_S, ji, row_end_S_nabor, jj;
00470         INT *graph_array = lambda;
00471     #else
00472         INT *ia = A->IA;
00473     #endif
00474
00475     LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;
00476
00477     SHORT nthreads = 1, use_openmp = FALSE;
00478
00479     #if DEBUG_MODE > 0
00480         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00481     #endif
00482
00483     #ifdef _OPENMP
00484         if ( row > OPENMP_HOLDS ) {
00485             use_openmp = TRUE;
00486             nthreads = fasp_get_num_threads();
00487         }
00488     #endif

```

```

00489
00490 // 0. Compress S and form S_transpose
00491 col = compress_S(S);
00492 if ( col < 0 ) goto FINISHED; // compression failed!!!
00493
00494 iCSRmat ST; fasp_iccsr_trans(S, &ST);
00495
00496 // 1. Initialize lambda
00497 if ( use_omp ) {
00498 #ifdef _OPENMP
00499 #pragma omp parallel for private(myid, mybegin, myend, i)
00500 #endif
00501     for ( myid = 0; myid < nthreads; myid++ ) {
00502         fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
00503         for ( i = mybegin; i < myend; i++ ) lambda[i] = ST.IA[i+1] - ST.IA[i];
00504     }
00505 }
00506 else {
00507     for ( i = 0; i < row; ++i ) lambda[i] = ST.IA[i+1] - ST.IA[i];
00508 }
00509
00510 // 2. Before C/F splitting algorithm starts, filter out the variables which
00511 // have no connections at all and mark them as special F-variables.
00512 if ( use_omp ) {
00513 #ifdef _OPENMP
00514 #pragma omp parallel for reduction(+:num_left) private(myid, mybegin, myend, i)
00515 #endif
00516 #endif
00517     for ( myid = 0; myid < nthreads; myid++ ) {
00518         fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
00519         for ( i = mybegin; i < myend; i++ ) {
00520 #if RS_C1 // Check C1 criteria or not
00521             if ( S->IA[i+1] == S->IA[i] )
00522 #else
00523                 if ( (ia[i+1]-ia[i]) <= 1 )
00524 #endif
00525                 {
00526                     vec[i] = ISPT; // set i as an ISOLATED fine node
00527                     lambda[i] = 0;
00528                 }
00529                 else {
00530                     vec[i] = UNPT; // set i as a undecided node
00531                     num_left++;
00532                 }
00533             }
00534         } // end for myid
00535     }
00536 }
00537
00538 else {
00539     for ( i = 0; i < row; ++i ) {
00540 #if RS_C1
00541         if ( S->IA[i+1] == S->IA[i] )
00542 #else
00543             if ( (ia[i+1]-ia[i]) <= 1 )
00544 #endif
00545             {
00546                 vec[i] = ISPT; // set i as an ISOLATED fine node
00547                 lambda[i] = 0;
00548             }
00549             else {
00550                 vec[i] = UNPT; // set i as a undecided node
00551                 num_left++;
00552             }
00553         } // end for i
00554     }
00555 }
00556
00557 // 3. Form linked list for lambda (max to min)
00558 for ( i = 0; i < row; ++i ) {
00559     if ( vec[i] == ISPT ) continue; // skip isolated variables
00560     measure = lambda[i];
00561     if ( measure > 0 ) {
00562         enter_list(&LoL_head, &LoL_tail, lambda[i], i, lists, where);
00563     }
00564     else {

```

```

00570
00571     if ( measure < 0 ) printf("### WARNING: Negative lambda[%d]!\n", i);
00572
00573     // Set variables with non-positive measure as F-variables
00574     vec[i] = FGPT; // no strong connections, set i as fine node
00575     --num_left;
00576
00577     // Update lambda and linked list after i->F
00578     for ( k = S->IA[i]; k < S->IA[i+1]; ++k ) {
00579         j = S->JA[k];
00580         if ( vec[j] == ISPT ) continue; // skip isolate variables
00581         if ( j < i ) {
00582             newmeas = lambda[j];
00583             if ( newmeas > 0 ) {
00584                 remove_node(&LoL_head, &LoL_tail, newmeas, j, lists, where);
00585             }
00586             newmeas = ++(lambda[j]);
00587             enter_list(&LoL_head, &LoL_tail, newmeas, j, lists, where);
00588         }
00589         else {
00590             newmeas = ++(lambda[j]);
00591         }
00592     }
00593
00594     } // end if measure
00595
00596 } // end for i
00597
00598 // 4. Main loop
00599 while ( num_left > 0 ) {
00600
00601     // pick $i$ in $U$ with $\max \lambda_i$: $C:=C\cup\{i\}$, $U:=U\setminus\{i\}$
00602     maxnode = LoL_head->head;
00603     maxmeas = lambda[maxnode];
00604     if ( maxmeas == 0 )
00605         printf("### WARNING: Head of the list has measure 0!\n");
00606
00607     vec[maxnode] = CGPT; // set maxnode as coarse node
00608     lambda[maxnode] = 0;
00609     --num_left;
00610     remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
00611     col++;
00612
00613     // for all $j$ in $S_i^T \cap U$: $F:=F\cup\{j\}$, $U:=U\setminus\{j\}$
00614     for ( i = ST.IA[maxnode]; i < ST.IA[maxnode+1]; ++i ) {
00615
00616         j = ST.JA[i];
00617
00618         if ( vec[j] != UNPT ) continue; // skip decided variables
00619
00620         vec[j] = FGPT; // set j as fine node
00621         remove_node(&LoL_head, &LoL_tail, lambda[j], j, lists, where);
00622         --num_left;
00623
00624         // Update lambda and linked list after j->F
00625         for ( l = S->IA[j]; l < S->IA[j+1]; ++l ) {
00626             k = S->JA[l];
00627             if ( vec[k] == UNPT ) { // k is unknown
00628                 remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
00629                 newmeas = ++(lambda[k]);
00630                 enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
00631             }
00632         }
00633
00634     } // end for i
00635
00636     // Update lambda and linked list after maxnode->C
00637     for ( i = S->IA[maxnode]; i < S->IA[maxnode+1]; ++i ) {
00638
00639         j = S->JA[i];
00640
00641         if ( vec[j] != UNPT ) continue; // skip decided variables
00642
00643         measure = lambda[j];
00644         remove_node(&LoL_head, &LoL_tail, measure, j, lists, where);
00645         lambda[j] = --measure;
00646
00647         if ( measure > 0 ) {
00648             enter_list(&LoL_head, &LoL_tail, measure, j, lists, where);
00649         }
00650         else { // j is the only point left, set as fine variable

```



```

00651         vec[j] = FGPT;
00652         --num_left;
00653
00654         // Update lambda and linked list after j->F
00655         for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
00656             k = S->JA[l];
00657             if ( vec[k] == UNPT ) { // k is unknown
00658                 remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
00659                 newmeas = ++(lambda[k]);
00660                 enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
00661             }
00662         } // end for l
00663     } // end if
00664 } // end for
00665 } // end while
00666 } // end while
00667 } // end while
00668
00669 #if RS_C1
00670
00671 // C/F splitting of RS coarsening check C1 Criterion
00672 fasp_iarray_set(row, graph_array, -1);
00673 for (i = 0; i < row; i++)
00674 {
00675     if (vec[i] == FGPT)
00676     {
00677         row_end_S = S->IA[i+1];
00678         for (ji = S->IA[i]; ji < row_end_S; ji++)
00679         {
00680             j = S->JA[ji];
00681             if (vec[j] == CGPT)
00682             {
00683                 graph_array[j] = i;
00684             }
00685         }
00686         cnt = 0;
00687         for (ji = S->IA[i]; ji < row_end_S; ji++)
00688         {
00689             j = S->JA[ji];
00690             if (vec[j] == FGPT)
00691             {
00692                 set_empty = 1;
00693                 row_end_S_nabor = S->IA[j+1];
00694                 for (jj = S->IA[j]; jj < row_end_S_nabor; jj++)
00695                 {
00696                     index = S->JA[jj];
00697                     if (graph_array[index] == i)
00698                     {
00699                         set_empty = 0;
00700                         break;
00701                     }
00702                 }
00703                 if (set_empty)
00704                 {
00705                     if (cnt == 0)
00706                     {
00707                         vec[j] = CGPT;
00708                         col++;
00709                         graph_array[j] = i;
00710                         jkeep = j;
00711                         cnt = 1;
00712                     }
00713                     else
00714                     {
00715                         vec[i] = CGPT;
00716                         vec[jkeep] = FGPT;
00717                         break;
00718                     }
00719                 }
00720             }
00721         }
00722     }
00723 }
00724
00725 #endif
00726
00727 fasp_icsr_free(&ST);
00728
00729 if ( LoL_head ) {
00730     list_ptr = LoL_head;
00731     LoL_head->prev_node = NULL;

```

```

00732         LoL_head->next_node = NULL;
00733         LoL_head = list_ptr->next_node;
00734         fasp_mem_free(list_ptr); list_ptr = NULL;
00735     }
00736
00737     FINISHED:
00738         fasp_mem_free(work); work = NULL;
00739
00740     #if DEBUG_MODE > 0
00741         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00742     #endif
00743
00744     return col;
00745 }
00746
00766 static INT cfsplitting_clsp (dCSRmat    *A,
00767                             iCSRmat    *S,
00768                             ivector    *vertices)
00769 {
00770     const INT    row = A->row;
00771
00772     // local variables
00773     INT col = 0;
00774     INT maxmeas, maxnode, num_left = 0;
00775     INT measure, newmeas;
00776     INT i, j, k, l;
00777     INT myid, mybegin, myend;
00778
00779     INT *ia = A->IA, *vec = vertices->val;
00780     INT *work = (INT*)fasp_mem_calloc(3*row, sizeof(INT));
00781     INT *lists = work, *where = lists+row, *lambda = where+row;
00782
00783     LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;
00784
00785     SHORT nthreads = 1, use_openmp = FALSE;
00786
00787     #if DEBUG_MODE > 0
00788         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00789     #endif
00790
00791     #ifdef _OPENMP
00792         if ( row > OPENMP_HOLDS ) {
00793             use_openmp = TRUE;
00794             nthreads = fasp_get_num_threads();
00795         }
00796     #endif
00797
00798     // 0. Compress S and form S_transpose (not complete, just IA and JA)
00799     iCSRmat Stemp;
00800     Stemp.row = S->row; Stemp.col = S->col; Stemp.nnz = S->nnz;
00801     Stemp.IA = (INT *)fasp_mem_calloc(S->row+1, sizeof(INT));
00802     fasp_iarray_cp (S->row+1, S->IA, Stemp.IA);
00803     Stemp.JA = (INT *)fasp_mem_calloc(S->nnz, sizeof(INT));
00804     fasp_iarray_cp (S->nnz, S->JA, Stemp.JA);
00805
00806     if ( compress_S(S) < 0 ) goto FINISHED; // compression failed!!!
00807
00808     iCSRmat ST; fasp_icsr_trans(S, &ST);
00809
00810     // 1. Initialize lambda
00811     if ( use_openmp ) {
00812     #ifdef _OPENMP
00813         #pragma omp parallel for private(myid, mybegin, myend, i)
00814     #endif
00815         for ( myid = 0; myid < nthreads; myid++ ) {
00816             fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
00817             for ( i = mybegin; i < myend; i++ ) lambda[i] = ST.IA[i+1] - ST.IA[i];
00818         }
00819     }
00820     else {
00821         for ( i = 0; i < row; ++i ) lambda[i] = ST.IA[i+1] - ST.IA[i];
00822     }
00823
00824     // 2. Before C/F splitting algorithm starts, filter out the variables which
00825     // have no connections at all and mark them as special F-variables.
00826     if ( use_openmp ) {
00827     #ifdef _OPENMP
00828         #pragma omp parallel for reduction(+:num_left) private(myid, mybegin, myend, i)
00829     #endif
00830     #endif
00831         for ( myid = 0; myid < nthreads; myid++ ) {

```

```

00832         fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
00833         for ( i = mybegin; i < myend; i++ ) {
00834             if ( (ia[i+1]-ia[i]) <= 1 ) {
00835                 vec[i] = ISPT; // set i as an ISOLATED fine node
00836                 lambda[i] = 0;
00837             }
00838             else {
00839                 vec[i] = UNPT; // set i as a undecided node
00840                 num_left++;
00841             }
00842         }
00843     } // end for myid
00844 }
00845 }
00846 else {
00847     for ( i = 0; i < row; ++i ) {
00848         if ( (ia[i+1]-ia[i]) <= 1 ) {
00849             vec[i] = ISPT; // set i as an ISOLATED fine node
00850             lambda[i] = 0;
00851         }
00852         else {
00853             vec[i] = UNPT; // set i as a undecided node
00854             num_left++;
00855         }
00856     } // end for i
00857 }
00858 }
00859 }
00860 // 3. Form linked list for lambda (max to min)
00861 for ( i = 0; i < row; ++i ) {
00862     if ( vec[i] == ISPT ) continue; // skip isolated variables
00863     measure = lambda[i];
00864     if ( measure > 0 ) {
00865         enter_list(&LoL_head, &LoL_tail, lambda[i], i, lists, where);
00866     }
00867     else {
00868         if ( measure < 0 ) printf("### WARNING: Negative lambda[%d]!\n", i);
00869         // Set variables with non-positive measure as F-variables
00870         vec[i] = FGPT; // no strong connections, set i as fine node
00871         --num_left;
00872         // Update lambda and linked list after i->F
00873         for ( k = S->IA[i]; k < S->IA[i+1]; ++k ) {
00874             j = S->JA[k];
00875             if ( vec[j] == ISPT ) continue; // skip isolate variables
00876             if ( j < i ) { // only look at the previous points!!
00877                 newmeas = lambda[j];
00878                 if ( newmeas > 0 ) {
00879                     remove_node(&LoL_head, &LoL_tail, newmeas, j, lists, where);
00880                     newmeas = ++(lambda[j]);
00881                     enter_list(&LoL_head, &LoL_tail, newmeas, j, lists, where);
00882                 }
00883                 else { // will be checked later on
00884                     newmeas = ++(lambda[j]);
00885                 } // end if
00886             } // end for k
00887         } // end if measure
00888     } // end for i
00889 } // end for i
00890 }
00891 // 4. Main loop
00892 while ( num_left > 0 ) {
00893     // pick $i$ in U$ with $\max\lambda_i$: C=C\cup\{i\}, U=U\cup\{i\}$
00894     maxnode = LoL_head->head;
00895     maxmeas = lambda[maxnode];
00896     if ( maxmeas == 0 )
00897         printf("### WARNING: Head of the list has measure 0!\n");
00898     vec[maxnode] = CGPT; // set maxnode as coarse node

```

```

00913     lambda[maxnode] = 0;
00914     --num_left;
00915     remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
00916     col++;
00917
00918     // for all $j$ in  $S_i^T \cap U$ :  $F := F \cup \{j\}$ ,  $U := U \setminus \{j\}$ 
00919     for ( i = ST.IA[maxnode]; i < ST.IA[maxnode+1]; ++i ) {
00920
00921         j = ST.JA[i];
00922
00923         if ( vec[j] != UNPT ) continue; // skip decided variables
00924
00925         vec[j] = FGPT; // set j as fine node
00926         remove_node(&LoL_head, &LoL_tail, lambda[j], j, lists, where);
00927         --num_left;
00928
00929         // Update lambda and linked list after j->F
00930         for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
00931             k = S->JA[l];
00932             if ( vec[k] == UNPT ) { // k is unknown
00933                 remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
00934                 newmeas = ++(lambda[k]);
00935                 enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
00936             }
00937         }
00938     } // end for i
00939
00940     // Update lambda and linked list after maxnode->C
00941     for ( i = S->IA[maxnode]; i < S->IA[maxnode+1]; ++i ) {
00942
00943         j = S->JA[i];
00944
00945         if ( vec[j] != UNPT ) continue; // skip decided variables
00946
00947         measure = lambda[j];
00948         remove_node(&LoL_head, &LoL_tail, measure, j, lists, where);
00949         lambda[j] = --measure;
00950
00951         if ( measure > 0 ) {
00952             enter_list(&LoL_head, &LoL_tail, measure, j, lists, where);
00953         }
00954         else { // j is the only point left, set as fine variable
00955             vec[j] = FGPT;
00956             --num_left;
00957
00958             // Update lambda and linked list after j->F
00959             for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
00960                 k = S->JA[l];
00961                 if ( vec[k] == UNPT ) { // k is unknown
00962                     remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
00963                     newmeas = ++(lambda[k]);
00964                     enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
00965                 }
00966             } // end for l
00967         } // end if
00968     } // end for
00969
00970 } // end for
00971
00972 } // end while
00973
00974 fasp_icsr_free(&ST);
00975
00976 if ( LoL_head ) {
00977     list_ptr = LoL_head;
00978     LoL_head->prev_node = NULL;
00979     LoL_head->next_node = NULL;
00980     LoL_head = list_ptr->next_node;
00981     fasp_mem_free(list_ptr); list_ptr = NULL;
00982 }
00983
00984 // Enforce F-C connections. Adding this step helps for the ExxonMobil test
00985 // problems! Need more tests though --Chensong 06/08/2013
00986 // col = clean_ff_couplings(S, vertices, row, col);
00987
00988 rem_positive_ff(A, &Stemp, vertices);
00989
00990 if ( compress_S(&Stemp) < 0 ) goto FINISHED; // compression failed!!!
00991
00992 S->row = Stemp.row;
00993 S->col = Stemp.col;

```

```

00994     S->nnz = Stemp.nnz;
00995
00996     fasp_mem_free(S->IA); S->IA = Stemp.IA;
00997     fasp_mem_free(S->JA); S->JA = Stemp.JA;
00998
00999     FINISHED:
01000     fasp_mem_free(work); work = NULL;
01001
01002     #if DEBUG_MODE > 0
01003     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01004     #endif
01005
01006     return col;
01007 }
01008
01029 static void strong_couplings_aggl (dCSRmat *A,
01030                                   iCSRmat *S,
01031                                   iCSRmat *Sh,
01032                                   ivector *vertices,
01033                                   ivector *CGPT_index,
01034                                   ivector *CGPT_rindex)
01035 {
01036     const INT row = A->row;
01037
01038     // local variables
01039     INT i, j, k;
01040     INT num_c, count, ci, cj, ck, fj, cck;
01041     INT *cp_index, *cp_rindex, *visited;
01042     INT *vec = vertices->val;
01043
01044     // count the number of coarse grid points
01045     for ( num_c = i = 0; i < row; i++ ) {
01046         if ( vec[i] == CGPT ) num_c++;
01047     }
01048
01049     // for the reverse indexing of coarse grid points
01050     fasp_ivec_alloc(row, CGPT_rindex);
01051     cp_rindex = CGPT_rindex->val;
01052
01053     // generate coarse grid point index
01054     fasp_ivec_alloc(num_c, CGPT_index);
01055     cp_index = CGPT_index->val;
01056     for ( j = i = 0; i < row; i++ ) {
01057         if ( vec[i] == CGPT ) {
01058             cp_index[j] = i;
01059             cp_rindex[i] = j;
01060             j++;
01061         }
01062     }
01063
01064     // allocate space for Sh
01065     Sh->row = Sh->col = num_c;
01066     Sh->val = Sh->JA = NULL;
01067     Sh->IA = (INT*)fasp_mem_calloc(Sh->row+1, sizeof(INT));
01068
01069     // record the number of times some coarse point is visited
01070     visited = (INT*)fasp_mem_calloc(num_c, sizeof(INT));
01071     fasp_iarray_set(num_c, visited, -1);
01072
01073     /*****
01074     /* step 1: Find first the structure IA of Sh */
01075     *****/
01076
01077     Sh->IA[0] = 0;
01078
01079     for ( ci = 0; ci < Sh->row; ci++ ) {
01080
01081         i = cp_index[ci]; // find the index of the ci-th coarse grid point
01082
01083         // number of coarse point that i is strongly connected to w.r.t. S(p,2)
01084         count = 0;
01085
01086         // visit all the fine neighbors that ci is strongly connected to
01087         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01088
01089             fj = S->JA[j];
01090
01091             if ( vec[fj] == CGPT && fj != i ) {
01092                 cj = cp_rindex[fj];
01093                 if ( visited[cj] != ci ) {
01094                     visited[cj] = ci; // mark as strongly connected from ci

```

```

01095         count++;
01096     }
01097 }
01098 }
01099
01100 else if ( vec[fj] == FGPT ) { // fine grid point,
01101
01102     // find all the coarse neighbors that fj is strongly connected to
01103     for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
01104         ck = S->JA[k];
01105         if ( vec[ck] == CGPT && ck != i ) { // it is a coarse grid point
01106             if ( cp_rindex[ck] >= num_c ) {
01107                 printf("### ERROR: ck=%d, num_c=%d, out of bound!\n",
01108                     ck, num_c);
01109                 fasp_chkerr(ERROR_AMG_COARSEING, __FUNCTION__);
01110             }
01111             cck = cp_rindex[ck];
01112
01113             if ( visited[cck] != ci ) {
01114                 visited[cck] = ci; // mark as strongly connected from ci
01115                 count++;
01116             }
01117         } //end if
01118     } //end for k
01119 } //end if
01120 } //end for j
01121
01122 } //end for j
01123
01124 Sh->IA[ci+1] = Sh->IA[ci] + count;
01125 } //end for i
01126
01127
01128 /*****
01129  * step 2: Find JA of Sh */
01130 /*****/
01131
01132 fasp_iarray_set(num_c, visited, -1); // reset visited
01133
01134 Sh->nnz = Sh->IA[Sh->row];
01135 Sh->JA = (INT*)fasp_mem_calloc(Sh->nnz, sizeof(INT));
01136
01137 for ( ci = 0; ci < Sh->row; ci++ ) {
01138
01139     i = cp_index[ci]; // find the index of the i-th coarse grid point
01140     count = Sh->IA[ci]; // count for coarse points
01141
01142     // visit all the fine neighbors that ci is strongly connected to
01143     for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01144         fj = S->JA[j];
01145
01146         if ( vec[fj] == CGPT && fj != i ) {
01147             cj = cp_rindex[fj];
01148             if ( visited[cj] != ci ) { // not visited yet
01149                 visited[cj] = ci;
01150                 Sh->JA[count] = cj;
01151                 count++;
01152             }
01153         }
01154     }
01155     else if ( vec[fj] == FGPT ) { // fine grid point,
01156         //find all the coarse neighbors that fj is strongly connected to
01157         for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
01158             ck = S->JA[k];
01159             if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
01160                 cck = cp_rindex[ck];
01161                 if ( visited[cck] != ci ) { // not visited yet
01162                     visited[cck] = ci;
01163                     Sh->JA[count] = cck;
01164                     count++;
01165                 }
01166             } // end if
01167         } // end for k
01168     } // end if
01169 } // end for j
01170
01171 if ( count != Sh->IA[ci+1] ) {
01172     printf("### WARNING: Inconsistent numbers of nonzeros!\n ");
01173 }
01174
01175

```

```

01176     } // end for ci
01177
01178     fasp_mem_free(visited); visited = NULL;
01179 }
01180
01207 static void strong_couplings_agg2 (dCSRmat *A,
01208                                   iCSRmat *S,
01209                                   iCSRmat *Sh,
01210                                   ivector *vertices,
01211                                   ivector *CGPT_index,
01212                                   ivector *CGPT_rindex)
01213 {
01214     const INT row = A->row;
01215
01216     // local variables
01217     INT i, j, k;
01218     INT num_c, count, ci, cj, ck, fj, cck;
01219     *cp_index, *cp_rindex, *visited;
01220     INT *vec = vertices->val;
01221
01222     // count the number of coarse grid points
01223     for ( num_c = i = 0; i < row; i++ ) {
01224         if ( vec[i] == CGPT ) num_c++;
01225     }
01226
01227     // for the reverse indexing of coarse grid points
01228     fasp_ivec_alloc(row, CGPT_rindex);
01229     cp_rindex = CGPT_rindex->val;
01230
01231     // generate coarse grid point index
01232     fasp_ivec_alloc(num_c, CGPT_index);
01233     cp_index = CGPT_index->val;
01234     for ( j = i = 0; i < row; i++ ) {
01235         if ( vec[i] == CGPT ) {
01236             cp_index[j] = i;
01237             cp_rindex[i] = j;
01238             j++;
01239         }
01240     }
01241
01242     // allocate space for Sh
01243     Sh->row = Sh->col = num_c;
01244     Sh->val = Sh->JA = NULL;
01245     Sh->IA = (INT*)fasp_mem_calloc(Sh->row+1, sizeof(INT));
01246
01247     // record the number of times some coarse point is visited
01248     visited = (INT*)fasp_mem_calloc(num_c, sizeof(INT));
01249     memset(visited, 0, sizeof(INT)*num_c);
01250
01251     /******
01252     /* step 1: Find first the structure IA of Sh */
01253     /******
01254
01255     Sh->IA[0] = 0;
01256
01257     for ( ci = 0; ci < Sh->row; ci++ ) {
01258
01259         i = cp_index[ci]; // find the index of the ci-th coarse grid point
01260
01261         // number of coarse point that i is strongly connected to w.r.t. S(p,2)
01262         count = 0;
01263
01264         // visit all the fine neighbors that ci is strongly connected to
01265         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01266             fj = S->JA[j];
01267
01268             if ( vec[fj] == CGPT && fj != i ) {
01269                 cj = cp_rindex[fj];
01270                 if ( visited[cj] != ci+1 ) { // not visited yet
01271                     visited[cj] = ci+1; // mark as strongly connected from ci
01272                     count++;
01273                 }
01274             }
01275         }
01276
01277         else if ( vec[fj] == FGPT ) { // fine grid point
01278
01279             // find all the coarse neighbors that fj is strongly connected to
01280             for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
01281
01282                 ck = S->JA[k];

```

```

01283
01284         if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
01285             if ( cp_rindex[ck] >= num_c ) {
01286                 printf("### ERROR: ck=%d, num_c=%d, out of bound!\n",
01287                     ck, num_c);
01288                 fasp_chkerr(ERROR_AMG_COARSEING, __FUNCTION__);
01289             }
01290             cck = cp_rindex[ck];
01291
01292             if ( visited[cck] == ci+1 ) {
01293                 // visited already!
01294             }
01295             else if ( visited[cck] == -ci-1 ) {
01296                 visited[cck] = ci+1; // mark as strongly connected from ci
01297                 count++;
01298             }
01299             else {
01300                 visited[cck] = -ci-1; // mark as visited
01301             }
01302         } //end if vec[ck]
01303     } //end for k
01304 } // end for j
01305 } // end if vec[fj]
01306 } // end for j
01307
01308 Sh->IA[ci+1] = Sh->IA[ci] + count;
01309 } //end for i
01310
01311 /*****
01312  /* step 2: Find JA of Sh */
01313  *****/
01314
01315 memset(visited, 0, sizeof(INT)*num_c); // reset visited
01316
01317 Sh->nnz = Sh->IA[Sh->row];
01318 Sh->JA = (INT*)fasp_mem_calloc(Sh->nnz, sizeof(INT));
01319
01320 for ( ci = 0; ci < Sh->row; ci++ ) {
01321     i = cp_index[ci]; // find the index of the i-th coarse grid point
01322     count = Sh->IA[ci]; // count for coarse points
01323
01324     // visit all the fine neighbors that ci is strongly connected to
01325     for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01326         fj = S->JA[j];
01327
01328         if ( vec[fj] == CGPT && fj != i ) {
01329             cj = cp_rindex[fj];
01330             if ( visited[cj] != ci+1 ) { // not visited yet
01331                 visited[cj] = ci+1;
01332                 Sh->JA[count] = cj;
01333                 count++;
01334             }
01335         }
01336     }
01337
01338     else if ( vec[fj] == FGPT ) { // fine grid point
01339         // find all the coarse neighbors that fj is strongly connected to
01340         for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
01341             ck = S->JA[k];
01342
01343             if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
01344                 cck = cp_rindex[ck];
01345                 if ( visited[cck] == ci+1 ) {
01346                     // visited before
01347                 }
01348                 else if ( visited[cck] == -ci-1 ) {
01349                     visited[cck] = ci+1;
01350                     Sh->JA[count] = cck;
01351                     count++;
01352                 }
01353                 else {
01354                     visited[cck] = -ci-1;
01355                 }
01356             }
01357         } // end if vec[ck]
01358     }
01359 }

```



```

01364
01365         } // end for k
01366
01367     } // end if vec[fj]
01368
01369 } // end for j
01370
01371     if ( count != Sh->IA[ci+1] ) {
01372         printf("### WARNING: Inconsistent numbers of nonzeros!\n ");
01373     }
01374
01375 } // end for ci
01376
01377 fasp_mem_free(visited); visited = NULL;
01378 }
01379
01401 static INT cfsplitting_agg (dCSRmat *A,
01402                             iCSRmat *S,
01403                             ivector *vertices,
01404                             INT aggressive_path)
01405 {
01406     const INT row = A->row;
01407     INT col = 0; // initialize col(P): returning output
01408
01409     // local variables
01410     INT *vec = vertices->val, *cp_index;
01411     INT maxmeas, maxnode, num_left = 0;
01412     INT measure, newmeas;
01413     INT i, j, k, l, m, ci, cj, ck, cl, num_c;
01414     SHORT IS_CNEIGH;
01415
01416     INT *work = (INT*)fasp_mem_calloc(3*row, sizeof(INT));
01417     INT *lists = work, *where = lists+row, *lambda = where+row;
01418
01419     ivector CGPT_index, CGPT_rindex;
01420     LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;
01421
01422     // Sh is for the strong coupling matrix between temporary CGPTs
01423     // ShT is the transpose of Sh
01424     // Snew is for combining the information from S and Sh
01425     iCSRmat ST, Sh, ShT;
01426
01427 #if DEBUG_MODE > 0
01428     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
01429 #endif
01430
01431 /* Coarsening Phase ONE: find temporary coarse level points */
01432 /* Coarsening Phase TWO: find real coarse level points */
01433
01434     num_c = cfsplitting_cls(A, S, vertices);
01435     fasp_icsr_trans(S, &ST);
01436
01437 /* Coarsening Phase TWO: find real coarse level points */
01438
01439     // find Sh, the strong coupling between coarse grid points S(path,2)
01440     if ( aggressive_path < 2 )
01441         strong_couplings_agg1(A, S, &Sh, vertices, &CGPT_index, &CGPT_rindex);
01442     else
01443         strong_couplings_agg2(A, S, &Sh, vertices, &CGPT_index, &CGPT_rindex);
01444
01445     fasp_icsr_trans(&Sh, &ShT);
01446
01447     CGPT_index.row = num_c;
01448     CGPT_rindex.row = row;
01449     cp_index = CGPT_index.val;
01450
01451     // 1. Initialize lambda
01452 #ifdef _OPENMP
01453 #pragma omp parallel for if(num_c>OPENMP_HOLDS)
01454 #endif
01455     for ( ci = 0; ci < num_c; ++ci ) lambda[ci] = ShT.IA[ci+1]-ShT.IA[ci];
01456
01457     // 2. Form linked list for lambda (max to min)
01458     for ( ci = 0; ci < num_c; ++ci ) {
01459         i = cp_index[ci];
01460         measure = lambda[ci];
01461     }
01462
01463
01464
01465

```

```

01466         if ( vec[i] == ISPT ) continue; // skip isolated points
01467
01468         if ( measure > 0 ) {
01469             enter_list(&LoL_head, &LoL_tail, lambda[ci], ci, lists, where);
01470             num_left++;
01471         }
01472     else {
01473         if ( measure < 0 ) printf("### WARNING: Negative lambda[%d]!\n", i);
01474
01475         vec[i] = FGPT; // set i as fine node
01476
01477         // update the lambda value in the CGPT neighbor of i
01478         for ( ck = Sh.IA[ci]; ck < Sh.IA[ci+1]; ++ck ) {
01479             cj = Sh.JA[ck];
01480             j = cp_index[cj];
01481
01482             if ( vec[j] == ISPT ) continue;
01483
01484             if ( cj < ci ) {
01485                 newmeas = lambda[cj];
01486                 if ( newmeas > 0 ) {
01487                     remove_node(&LoL_head, &LoL_tail, newmeas, cj, lists, where);
01488                     num_left--;
01489                 }
01490                 newmeas = ++(lambda[cj]);
01491                 enter_list(&LoL_head, &LoL_tail, newmeas, cj, lists, where);
01492                 num_left++;
01493             }
01494             else {
01495                 newmeas = ++(lambda[cj]);
01496             } // end if cj<ci
01497         } // end for ck
01498     } // end if
01499 } // end for ci
01500 } // end if
01501 } // end for ci
01502
01503 // 3. Main loop
01504 while ( num_left > 0 ) {
01505     // pick $i$ in $U$ with $\max \lambda_{i-1}$: $C=C \cup \{i\}$, $U=U \setminus \{i\}$
01506     maxnode = LoL_head->head;
01507     maxmeas = lambda[maxnode];
01508     if ( maxmeas == 0 ) printf("### WARNING: Head of the list has measure 0!\n");
01509
01510     // mark maxnode as real coarse node, labelled as number 3
01511     vec[cp_index[maxnode]] = 3;
01512     --num_left;
01513     remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
01514     lambda[maxnode] = 0;
01515     col++; // count for the real coarse node after aggressive coarsening
01516
01517     // for all $j$ in $S_i \cap U$: $F=F \cup \{j\}$, $U=U \setminus \{j\}$
01518     for ( ci = ShT.IA[maxnode]; ci < ShT.IA[maxnode+1]; ++ci ) {
01519         cj = ShT.JA[ci];
01520         j = cp_index[cj];
01521
01522         if ( vec[j] != CGPT ) continue; // skip if j is not C-point
01523
01524         vec[j] = 4; // set j as 4--fake CGPT
01525         remove_node(&LoL_head, &LoL_tail, lambda[cj], cj, lists, where);
01526         --num_left;
01527
01528         // update the measure for neighboring points
01529         for ( cl = Sh.IA[cj]; cl < Sh.IA[cj+1]; ++cl ) {
01530             ck = Sh.JA[cl];
01531             k = cp_index[ck];
01532             if ( vec[k] == CGPT ) { // k is temporary CGPT
01533                 remove_node(&LoL_head, &LoL_tail, lambda[ck], ck, lists, where);
01534                 newmeas = ++(lambda[ck]);
01535                 enter_list(&LoL_head, &LoL_tail, newmeas, ck, lists, where);
01536             }
01537         }
01538     }
01539 } // end for ci
01540 } // end for ci
01541
01542 // Update lambda and linked list after maxnode->C
01543 for ( ci = Sh.IA[maxnode]; ci < Sh.IA[maxnode+1]; ++ci ) {

```

```

01547
01548     cj = Sh.JA[cj];
01549     j = cp_index[cj];
01550
01551     if ( vec[j] != CGPT ) continue; // skip if j is not C-point
01552
01553     measure = lambda[cj];
01554     remove_node(&LoL_head, &LoL_tail, measure, cj, lists, where);
01555     lambda[cj] = --measure;
01556
01557     if ( measure > 0 ) {
01558         enter_list(&LoL_head, &LoL_tail, measure, cj, lists, where);
01559     }
01560     else {
01561         vec[j] = 4; // set j as fake CGPT variable
01562         --num_left;
01563         for ( cl = Sh.IA[cj]; cl < Sh.IA[cj+1]; cl++ ) {
01564             ck = Sh.JA[cl];
01565             k = cp_index[ck];
01566             if ( vec[k] == CGPT ) { // k is temporary CGPT
01567                 remove_node(&LoL_head, &LoL_tail, lambda[ck], ck, lists, where);
01568                 newmeas = ++(lambda[ck]);
01569                 enter_list(&LoL_head, &LoL_tail, newmeas, ck, lists, where);
01570             }
01571         } // end for l
01572     } // end if
01573
01574 } // end for
01575
01576 } // while
01577
01578 // 4. reorganize the variable type: mark temporary CGPT--1 and fake CGPT--4 as
01579 // FGPT; mark real CGPT--3 to be CGPT
01580 #ifdef _OPENMP
01581 #pragma omp parallel for if(row>OPENMP_HOLDS)
01582 #endif
01583     for ( i = 0; i < row; i++ ) {
01584         if ( vec[i] == CGPT || vec[i] == 4 ) vec[i] = FGPT;
01585     }
01586
01587 #ifdef _OPENMP
01588 #pragma omp parallel for if(row>OPENMP_HOLDS)
01589 #endif
01590     for ( i = 0; i < row; i++ ) {
01591         if ( vec[i] == 3 ) vec[i] = CGPT;
01592     }
01593
01594 /*****/
01595 /* Coarsening Phase THREE: all the FGPTs which have no CGPT */
01596 /* neighbors within distance 2. Change them into CGPT such */
01597 /* that the standard interpolation works! */
01598 /*****/
01599
01600     for ( i = 0; i < row; i++ ) {
01601
01602         if ( vec[i] != FGPT ) continue;
01603
01604         IS_CNEIGH = FALSE; // whether there exist CGPT neighbors within distance of 2
01605
01606         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01607
01608             if ( IS_CNEIGH ) break;
01609
01610             k = S->JA[j];
01611
01612             if ( vec[k] == CGPT ) {
01613                 IS_CNEIGH = TRUE;
01614             }
01615             else if ( vec[k] == FGPT ) {
01616                 for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) {
01617                     m = S->JA[l];
01618                     if ( vec[m] == CGPT ) {
01619                         IS_CNEIGH = TRUE; break;
01620                     }
01621                 } // end for l
01622             }
01623
01624         } // end for j
01625
01626         // no CGPT neighbors in distance <= 2, mark i as CGPT
01627         if ( !IS_CNEIGH ) {

```

```

01628         vec[i] = CGPT; col++;
01629     }
01630
01631 } // end for i
01632
01633 if ( LoL_head ) {
01634     list_ptr = LoL_head;
01635     LoL_head->prev_node = NULL;
01636     LoL_head->next_node = NULL;
01637     LoL_head = list_ptr->next_node;
01638     fasp_mem_free(list_ptr); list_ptr = NULL;
01639 }
01640
01641 fasp_ivec_free(&CGPT_index);
01642 fasp_ivec_free(&CGPT_rindex);
01643 fasp_icsr_free(&Sh);
01644 fasp_icsr_free(&ST);
01645 fasp_icsr_free(&ShT);
01646 fasp_mem_free(work); work = NULL;
01647
01648 #if DEBUG_MODE > 0
01649 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01650 #endif
01651
01652 return col;
01653 }
01654
01655 static INT clean_ff_couplings (iCSRmat *S,
01656                               ivector *vertices,
01657                               INT row,
01658                               INT col)
01659 {
01660     // local variables
01661     INT *vec = vertices->val;
01662     INT *cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));
01663     INT set_empty = TRUE, C_i_nonempty = FALSE;
01664     INT ci_tilde = -1, ci_tilde_mark = -1;
01665     INT ji, jj, i, j, index;
01666
01667     fasp_iarray_set(row, cindex, -1);
01668
01669     for ( i = 0; i < row; ++i ) {
01670
01671         if ( vec[i] != FGPT ) continue; // skip non F-variables
01672
01673         for ( ji = S->IA[i]; ji < S->IA[i+1]; ++ji ) {
01674             j = S->JA[ji];
01675             if ( vec[j] == CGPT ) cindex[j] = i; // mark C-neighbors
01676             else cindex[j] = -1; // reset cindex --Chensong 06/02/2013
01677         }
01678
01679         if ( ci_tilde_mark != i ) ci_tilde = -1;////
01680
01681         for ( ji = S->IA[i]; ji < S->IA[i+1]; ++ji ) {
01682             j = S->JA[ji];
01683
01684             if ( vec[j] != FGPT ) continue; // skip non F-variables
01685
01686             // check whether there is a C-connection
01687             set_empty = TRUE;
01688             for ( jj = S->IA[j]; jj < S->IA[j+1]; ++jj ) {
01689                 index = S->JA[jj];
01690                 if ( cindex[index] == i ) {
01691                     set_empty = FALSE; break;
01692                 }
01693             } // end for jj
01694
01695             // change the point i (if only F-F exists) to C
01696             if ( set_empty ) {
01697                 if ( C_i_nonempty ) {
01698                     vec[i] = CGPT; col++;
01699                     if ( ci_tilde > -1 ) {
01700                         vec[ci_tilde] = FGPT; col--;
01701                         ci_tilde = -1;
01702                     }
01703                     C_i_nonempty = FALSE;
01704                     break;
01705                 }
01706                 else { // temporary set j->C and roll back
01707                     vec[j] = CGPT; col++;
01708                 }
01709             }
01710         }
01711     }
01712 }

```

```

01732         ci_tilde = j;
01733         ci_tilde_mark = i;
01734         C_i_nonempty = TRUE;
01735         i--; // roll back to check i-point again
01736         break;
01737     } // end if C_i_nonempty
01738 } // end if set_empty
01739
01740 } // end for ji
01741
01742 } // end for i
01743
01744 fasp_mem_free(cindex); cindex = NULL;
01745
01746 return col;
01747 }
01748
01767 static void form_P_pattern_dir (dCSRmat *P,
01768                                iCSRmat *S,
01769                                ivector *vertices,
01770                                INT row,
01771                                INT col)
01772 {
01773     // local variables
01774     INT i, j, k, index;
01775     INT *vec = vertices->val;
01776
01777     SHORT nthreads = 1, use_openmp = FALSE;
01778
01779 #ifdef _OPENMP
01780     if ( row > OPENMP_HOLDS ) {
01781         use_openmp = TRUE;
01782         nthreads = fasp_get_num_threads();
01783     }
01784 #endif
01785
01786     // Initialize P matrix
01787     P->row = row; P->col = col;
01788     P->IA = (INT*)fasp_mem_calloc(row+1, sizeof(INT));
01789
01790     // step 1: Find the structure IA of P first: using P as a counter
01791     if ( use_openmp ) {
01792
01793         INT mybegin, myend, myid;
01794 #ifdef _OPENMP
01795 #pragma omp parallel for private(myid, mybegin,myend,i,j,k)
01796 #endif
01797         for ( myid = 0; myid < nthreads; myid++ ) {
01798             fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
01799             for ( i = mybegin; i < myend; ++i ) {
01800                 switch ( vec[i] ) {
01801                     case FGPT: // fine grid points
01802                         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01803                             k = S->JA[j];
01804                             if ( vec[k] == CGPT ) P->IA[i+1]++;
01805                         }
01806                         break;
01807
01808                     case CGPT: // coarse grid points
01809                         P->IA[i+1] = 1; break;
01810
01811                     default: // treat everything else as isolated
01812                         P->IA[i+1] = 0; break;
01813                 }
01814             }
01815         }
01816
01817     }
01818
01819     else {
01820
01821         for ( i = 0; i < row; ++i ) {
01822             switch ( vec[i] ) {
01823                 case FGPT: // fine grid points
01824                     for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01825                         k = S->JA[j];
01826                         if ( vec[k] == CGPT ) P->IA[i+1]++;
01827                     }
01828                     break;
01829
01830                 case CGPT: // coarse grid points

```

```

01831         P->IA[i+1] = 1; break;
01832
01833         default: // treat everything else as isolated
01834             P->IA[i+1] = 0; break;
01835     }
01836 } // end for i
01837
01838 } // end if
01839
01840 // Form P->IA from the counter P
01841 for ( i = 0; i < P->row; ++i ) P->IA[i+1] += P->IA[i];
01842 P->nnz = P->IA[P->row]-P->IA[0];
01843
01844 // step 2: Find the structure JA of P
01845 P->JA = (INT*)fasp_mem_calloc(P->nnz,sizeof(INT));
01846 P->val = (REAL*)fasp_mem_calloc(P->nnz,sizeof(REAL));
01847
01848 for ( index = i = 0; i < row; ++i ) {
01849     if ( vec[i] == FGPT ) { // fine grid point
01850         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01851             k = S->JA[j];
01852             if ( vec[k] == CGPT ) P->JA[index++] = k;
01853         } // end for j
01854     } // end if
01855     else if ( vec[i] == CGPT ) { // coarse grid point -- one entry only
01856         P->JA[index++] = i;
01857     }
01858 }
01859
01860 }
01861
01862 static void form_P_pattern_std (dCSRmat *P,
01863                                iCSRmat *S,
01864                                ivector *vertices,
01865                                INT row,
01866                                INT col)
01867 {
01868     // local variables
01869     INT i, j, k, l, h, index;
01870     INT *vec = vertices->val;
01871
01872     // number of times a C-point is visited
01873     INT *visited = (INT*)fasp_mem_calloc(row,sizeof(INT));
01874
01875     P->row = row; P->col = col;
01876     P->IA = (INT*)fasp_mem_calloc(row+1, sizeof(INT));
01877
01878     fasp_iarray_set(row, visited, -1);
01879
01880     // Step 1: Find the structure IA of P first: use P as a counter
01881     for ( i = 0; i < row; ++i ) {
01882
01883         if ( vec[i] == FGPT ) { // if node i is a F point
01884             for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01885
01886                 k = S->JA[j];
01887
01888                 // if neighbor of i is a C point, good
01889                 if ( (vec[k] == CGPT) && (visited[k] != i) ) {
01890                     visited[k] = i;
01891                     P->IA[i+1]++;
01892                 }
01893
01894                 // if k is a F point and k is not i, look for indirect C neighbors
01895                 else if ( (vec[k] == FGPT) && (k != i) ) {
01896                     for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) { // neighbors of k
01897                         h = S->JA[l];
01898                         if ( (vec[h] == CGPT) && (visited[h] != i) ) {
01899                             visited[h] = i;
01900                             P->IA[i+1]++;
01901                         }
01902                     } // end for(l=S->IA[k];l<S->IA[k+1];l++)
01903                 } // end if (vec[k]==CGPT)
01904             } // end for (j=S->IA[i];j<S->IA[i+1];j++)
01905         }
01906
01907         else if ( vec[i] == CGPT ) { // if node i is a C point
01908             P->IA[i+1] = 1;
01909         }
01910     }
01911 }

```

```

01930         else { // treat everything else as isolated points
01931             P->IA[i+1] = 0;
01932         } // end if (vec[i]==FGPT)
01933     } // end for (i=0;i<row;++i)
01934
01935     // Form P->IA from the counter P
01936     for ( i = 0; i < P->row; ++i ) P->IA[i+1] += P->IA[i];
01937     P->nnz = P->IA[P->row]-P->IA[0];
01938
01939     // Step 2: Find the structure JA of P
01940     P->JA = (INT*)fasp_mem_calloc(P->nnz,sizeof(INT));
01941     P->val = (REAL*)fasp_mem_calloc(P->nnz,sizeof(REAL));
01942
01943     fasp_iarray_set(row, visited, -1); // re-init visited array
01944
01945     for ( i = 0; i < row; ++i ) {
01946
01947         if ( vec[i] == FGPT ) { // if node i is a F point
01948
01949             index = 0;
01950
01951             for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
01952
01953                 k = S->JA[j];
01954
01955                 // if neighbor k of i is a C point
01956                 if ( (vec[k] == CGPT) && (visited[k] != i) ) {
01957                     visited[k] = i;
01958                     P->JA[P->IA[i]+index] = k;
01959                     index++;
01960                 }
01961
01962                 // if neighbor k of i is a F point and k is not i
01963                 else if ( (vec[k] == FGPT) && (k != i) ) {
01964                     for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) { // neighbors of k
01965                         h = S->JA[l];
01966                         if ( (vec[h] == CGPT) && (visited[h] != i) ) {
01967                             visited[h] = i;
01968                             P->JA[P->IA[i]+index] = h;
01969                             index++;
01970                         }
01971                     }
01972                 } // end for (l=S->IA[k];l<S->IA[k+1];l++)
01973             } // end if (vec[k]==CGPT)
01974         } // end if (vec[i]==FGPT)
01975     } // end for (j=S->IA[i];j<S->IA[i+1];j++)
01976
01977     else if ( vec[i] == CGPT ) {
01978         P->JA[P->IA[i]] = i;
01979     }
01980 }
01981
01982 // clean up
01983 fasp_mem_free(visited); visited = NULL;
01984 }
01985
02003 static INT cfsplitting_mis (iCSRmat *S,
02004                             ivector *vertices,
02005                             ivector *order)
02006 {
02007     const INT n = S->row;
02008
02009     INT col = 0;
02010     INT *ord = order->val;
02011     INT *vec = vertices->val;
02012     INT *IS = S->IA;
02013     INT *JS = S->JA;
02014
02015     INT i, j, ind;
02016     INT row_begin, row_end;
02017
02018     fasp_ivec_set (n, vertices, UNPT);
02019
02020     for (i=0; i<n ; i++)
02021     {
02022         ind = ord[i];
02023         if (vec[ind] == UNPT) {
02024             vec[ind] = CGPT;

```

```

02025         row_begin = IS[ind]; row_end = IS[ind+1];
02026         for (j = row_begin; j<row_end; j++)
02027         {
02028             if (vec[JS[j]] == CGPT ) {
02029                 vec[ind] = FGPT;
02030                 break;
02031             }
02032         }
02033         if (vec[ind] == CGPT) {
02034             col++;
02035             for (j = row_begin; j<row_end; j++)
02036             {
02037                 vec[JS[j]] = FGPT;
02038             }
02039         }
02040     }
02041 }
02042 return col;
02043 }
02044
02059 static void ordering1 (iCSRmat *S,
02060                       ivector *order)
02061 {
02062     const INT n = order->row;
02063     INT * IS = S->IA;
02064     INT * ord = order->val;
02065     INT maxind, maxdeg, degree;
02066     INT i;
02067
02068     for (i = 0; i < n; i++) ord[i] = i;
02069
02070     for (maxind = maxdeg = i = 0; i < n; i++)
02071     {
02072         degree = IS[i+1] - IS[i];
02073         if (degree > maxdeg)
02074         {
02075             maxind = i;
02076             maxdeg = degree;
02077         }
02078     }
02079
02080     ord[0] = maxind;
02081     ord[maxind] = 0;
02082
02083     return;
02084 }
02085
02086 /*-----*/
02087 /*--      End of File      --*/
02088 /*-----*/

```

9.139 PreAMGInterp.c File Reference

Direct and standard interpolations for classical AMG.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void `fasp_amg_interp` (dCSRmat *A, ivector *vertices, dCSRmat *P, iCSRmat *S, AMG_param *param)
Generate interpolation operator P.

9.139.1 Detailed Description

Direct and standard interpolations for classical AMG.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), and [PreAMGInterpEM.c](#)

Reference: U. Trottenberg, C. W. Oosterlee, and A. Schuller Multigrid (Appendix A: An Intro to Algebraic Multigrid) Academic Press Inc., San Diego, CA, 2001 With contributions by A. Brandt, P. Oswald and K. Stuben.
Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGInterp.c](#).

9.139.2 Function Documentation**9.139.2.1 fasp_amg_interp()**

```
void fasp_amg_interp (
    dCSRmat * A,
    ivector * vertices,
    dCSRmat * P,
    iCSRmat * S,
    AMG_param * param )
```

Generate interpolation operator P.

Parameters

<i>A</i>	Pointer to dCSRmat coefficient matrix (index starts from 0)
<i>vertices</i>	Indicator vector for the C/F splitting of the variables
<i>P</i>	Prolongation (input: nonzero pattern, output: prolongation)
<i>S</i>	Strong connection matrix
<i>param</i>	AMG parameters

Author

Xuehai Huang, Chensong Zhang

Date

04/04/2010

Modified by Xiaozhe Hu on 05/23/2012: add S as input Modified by Chensong Zhang on 09/12/2012: clean up and debug interp_RS Modified by Chensong Zhang on 05/14/2013: reconstruct the code
Definition at line 63 of file [PreAMGInterp.c](#).

9.140 PreAMGInterp.c

[Go to the documentation of this file.](#)

```
00001
00021 #include <math.h>
00022 #include <time.h>
00023
00024 #ifdef _OPENMP
00025 #include <omp.h>
```

```

00026 #endif
00027
00028 #include "fasp.h"
00029 #include "fasp_functs.h"
00030
00031 /*-----*/
00032 /*--  Declare Private Functions  --*/
00033 /*-----*/
00034
00035 static void interp_DIR (dCSRmat *, ivector *, dCSRmat *, AMG_param *);
00036 static void interp_STD (dCSRmat *, ivector *, dCSRmat *, iCSRmat *, AMG_param *);
00037 static void interp_EXT (dCSRmat *, ivector *, dCSRmat *, iCSRmat *, AMG_param *);
00038 static void amg_interp_trunc (dCSRmat *, AMG_param *);
00039
00040 /*-----*/
00041 /*--      Public Functions      --*/
00042 /*-----*/
00043
00063 void fasp_amg_interp (dCSRmat      *A,
00064                     ivector      *vertices,
00065                     dCSRmat      *P,
00066                     iCSRmat      *S,
00067                     AMG_param    *param)
00068 {
00069     const INT coarsening_type = param->coarsening_type;
00070     INT      interp_type     = param->interpolation_type;
00071
00072     // make sure standard interpolation is used for aggressive coarsening
00073     if ( coarsening_type == COARSE_AC ) interp_type = INTERP_STD;
00074
00075 #if DEBUG_MODE > 0
00076     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00077 #endif
00078
00079     switch ( interp_type ) {
00080
00081         case INTERP_DIR: // Direct interpolation
00082             interp_DIR(A, vertices, P, param); break;
00083
00084         case INTERP_STD: // Standard interpolation
00085             interp_STD(A, vertices, P, S, param); break;
00086
00087         case INTERP_EXT: // Extended interpolation
00088             interp_EXT(A, vertices, P, S, param); break;
00089
00090         case INTERP_ENG: // Energy-min interpolation defined in PreAMGInterpEM.c
00091             fasp_amg_interp_em(A, vertices, P, param); break;
00092
00093         default:
00094             fasp_chkerr(ERROR_AMG_INTERP_TYPE, __FUNCTION__);
00095
00096     }
00097
00098 #if DEBUG_MODE > 0
00099     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00100 #endif
00101 }
00102
00103 /*-----*/
00104 /*--      Private Functions      --*/
00105 /*-----*/
00106
00122 static void amg_interp_trunc (dCSRmat      *P,
00123                             AMG_param    *param)
00124 {
00125     const INT  row      = P->row;
00126     const INT  nnzold   = P->nnz;
00127     const INT  prtlvl   = param->print_level;
00128     const REAL eps_tr   = param->truncation_threshold;
00129
00130     // local variables
00131     INT  num_nonzero = 0; // number of non zeros after truncation
00132     REAL Min_neg, Max_pos; // min negative and max positive entries
00133     REAL Fac_neg, Fac_pos; // factors for negative and positive entries
00134     REAL Sum_neg, TSum_neg; // sum and truncated sum of negative entries
00135     REAL Sum_pos, TSum_pos; // sum and truncated sum of positive entries
00136
00137     INT  index1 = 0, index2 = 0, begin, end;
00138     INT  i, j;
00139
00140 #if DEBUG_MODE > 0

```

```

00141     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00142 #endif
00143
00144     for ( i = 0; i < row; ++i ) {
00145
00146         begin = P->IA[i]; end = P->IA[i+1];
00147
00148         P->IA[i] = num_nonzero;
00149         Min_neg = Max_pos = 0;
00150         Sum_neg = Sum_pos = 0;
00151         TSum_neg = TSum_pos = 0;
00152
00153         // 1. Summations of positive and negative entries
00154         for ( j = begin; j < end; ++j ) {
00155
00156             if ( P->val[j] > 0 ) {
00157                 Sum_pos += P->val[j];
00158                 Max_pos = MAX(Max_pos, P->val[j]);
00159             }
00160
00161             else {
00162                 Sum_neg += P->val[j];
00163                 Min_neg = MIN(Min_neg, P->val[j]);
00164             }
00165
00166         }
00167
00168         // Truncate according to max and min values!!!
00169         Max_pos *= eps_tr; Min_neg *= eps_tr;
00170
00171         // 2. Set JA of truncated P
00172         for ( j = begin; j < end; ++j ) {
00173
00174             if ( P->val[j] >= Max_pos ) {
00175                 num_nonzero++;
00176                 P->JA[index1++] = P->JA[j];
00177                 TSum_pos += P->val[j];
00178             }
00179
00180             else if ( P->val[j] <= Min_neg ) {
00181                 num_nonzero++;
00182                 P->JA[index1++] = P->JA[j];
00183                 TSum_neg += P->val[j];
00184             }
00185
00186         }
00187
00188         // 3. Compute factors and set values of truncated P
00189         if ( TSum_pos > SMALLREAL ) {
00190             Fac_pos = Sum_pos / TSum_pos; // factor for positive entries
00191         }
00192         else {
00193             Fac_pos = 1.0;
00194         }
00195
00196         if ( TSum_neg < -SMALLREAL ) {
00197             Fac_neg = Sum_neg / TSum_neg; // factor for negative entries
00198         }
00199         else {
00200             Fac_neg = 1.0;
00201         }
00202
00203         for ( j = begin; j < end; ++j ) {
00204
00205             if ( P->val[j] >= Max_pos )
00206                 P->val[index2++] = P->val[j] * Fac_pos;
00207
00208             else if ( P->val[j] <= Min_neg )
00209                 P->val[index2++] = P->val[j] * Fac_neg;
00210
00211         }
00212     }
00213
00214     // resize the truncated prolongation P
00215     P->nnz = P->IA[row] = num_nonzero;
00216     P->JA = (INT *)fasp_mem_realloc(P->JA, num_nonzero*sizeof(INT));
00217     P->val = (REAL *)fasp_mem_realloc(P->val, num_nonzero*sizeof(REAL));
00218
00219     if ( prtlvl >= PRINT_MOST ) {
00220         printf("NNZ in prolongator: before truncation = %10d, after = %10d\n",
00221             nnzold, num_nonzero);

```

```

00222     }
00223
00224 #if DEBUG_MODE > 0
00225     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00226 #endif
00227
00228 }
00229
00250 static void interp_DIR (dCSRmat    *A,
00251                        ivector    *vertices,
00252                        dCSRmat    *P,
00253                        AMG_param  *param )
00254 {
00255     INT    row = A->row;
00256     INT    *vec = vertices->val;
00257
00258     // local variables
00259     SHORT  IS_STRONG; // is the variable strong coupled to i?
00260     INT    num_pcouple; // number of positive strong couplings
00261     INT    begin_row, end_row;
00262     INT    i, j, k, l, index = 0, idiag;
00263
00264     // a_minus and a_plus for Neighbors and Prolongation support
00265     REAL    amN, amP, apN, apP;
00266     REAL    alpha, beta, aii = 0.0;
00267
00268     // indices of C-nodes
00269     INT    *cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));
00270
00271     SHORT  use_omp = FALSE;
00272
00273 #ifdef _OPENMP
00274     INT myid, mybegin, myend, stride_i, nthreads;
00275     // row = MIN(P->IA[P->row], row);
00276     if ( MIN(P->IA[P->row], row) > OPENMP_HOLDS ) {
00277         use_omp = TRUE;
00278         nthreads = fasp_get_num_threads();
00279     }
00280 #endif
00281
00282     // Step 1. Fill in values for interpolation operator P
00283     if (use_omp) {
00284 #ifdef _OPENMP
00285         stride_i = row/nthreads;
00286 #pragma omp parallel private(myid,mybegin,myend,i,begin_row,end_row,idiag,aii, \
00287 amN,amP,apN,apP,num_pcouple,j,k,alpha,beta,l) \
00288 num_threads(nthreads)
00289         {
00290             myid = omp_get_thread_num();
00291             mybegin = myid*stride_i;
00292             if (myid < nthreads-1) myend = mybegin+stride_i;
00293             else myend = row;
00294             aii = 0.0;
00295
00296             for (i=mybegin; i<myend; ++i) {
00297                 begin_row=A->IA[i]; end_row=A->IA[i+1]-1;
00298                 for (idiag=begin_row; idiag<=end_row; idiag++) {
00299                     if (A->JA[idiag]==i) {
00300                         aii=A->val[idiag];
00301                         break;
00302                     }
00303                 }
00304                 if (vec[i]==0) { // if node i is on fine grid
00305                     amN=0, amP=0, apN=0, apP=0, num_pcouple=0;
00306                     for (j=begin_row; j<=end_row; ++j) {
00307                         if (j==idiag) continue;
00308                         for (k=P->IA[i]; k<P->IA[i+1]; ++k) {
00309                             if (P->JA[k]==A->JA[j]) break;
00310                         }
00311                         if (A->val[j]>0) {
00312                             apN+=A->val[j];
00313                             if (k<P->IA[i+1]) {
00314                                 apP+=A->val[j];
00315                                 num_pcouple++;
00316                             }
00317                         }
00318                         else {
00319                             amN+=A->val[j];
00320                             if (k<P->IA[i+1]) {
00321                                 amP+=A->val[j];
00322                             }

```

```

00323         }
00324     } // j
00325
00326     alpha=amN/amP;
00327     if (num_pcouple>0) {
00328         beta=apN/apP;
00329     }
00330     else {
00331         beta=0;
00332         aii+=apN;
00333     }
00334     for (j=P->IA[i]; j<P->IA[i+1]; ++j) {
00335         k=P->JA[j];
00336         for (l=A->IA[i]; l<A->IA[i+1]; l++) {
00337             if (A->JA[l]==k) break;
00338         }
00339         if (A->val[l]>0) {
00340             P->val[j]=-beta*A->val[l]/aii;
00341         }
00342         else {
00343             P->val[j]=-alpha*A->val[l]/aii;
00344         }
00345     }
00346 }
00347 else if (vec[i]==2) // if node i is a special fine node
00348 {
00349
00350 }
00351 else { // if node i is on coarse grid
00352     P->val[P->IA[i]]=1;
00353 }
00354 }
00355 }
00356 #endif
00357 }
00358
00359 else {
00360
00361     for ( i = 0; i < row; ++i ) {
00362
00363         begin_row = A->IA[i]; end_row = A->IA[i+1];
00364
00365         // find diagonal entry first!!!
00366         for ( idiag = begin_row; idiag < end_row; idiag++ ) {
00367             if ( A->JA[idiag] == i ) {
00368                 aii = A->val[idiag]; break;
00369             }
00370         }
00371
00372         if ( vec[i] == FGPT ) { // fine grid nodes
00373
00374             amN = amP = apN = apP = 0.0;
00375
00376             num_pcouple = 0;
00377
00378             for ( j = begin_row; j < end_row; ++j ) {
00379
00380                 if ( j == idiag ) continue; // skip diagonal
00381
00382                 // check a point strong-coupled to i or not
00383                 IS_STRONG = FALSE;
00384                 for ( k = P->IA[i]; k < P->IA[i+1]; ++k ) {
00385                     if ( P->JA[k] == A->JA[j] ) { IS_STRONG = TRUE; break; }
00386                 }
00387
00388                 if ( A->val[j] > 0 ) {
00389                     apN += A->val[j]; // sum up positive entries
00390                     if ( IS_STRONG ) { apP += A->val[j]; num_pcouple++; }
00391                 }
00392                 else {
00393                     amN += A->val[j]; // sum up negative entries
00394                     if ( IS_STRONG ) { amP += A->val[j]; }
00395                 }
00396             } // end for j
00397
00398             // set weight factors
00399             alpha = amN / amP;
00400             if ( num_pcouple > 0 ) {
00401                 beta = apN / apP;
00402             }
00403             else {

```

```

00404         beta = 0.0; aii += apN;
00405     }
00406
00407     // keep aii inside the loop to avoid floating pt error! --Chensong
00408     for ( j = P->IA[i]; j < P->IA[i+1]; ++j ) {
00409         k = P->JA[j];
00410         for ( l = A->IA[i]; l < A->IA[i+1]; l++ ) {
00411             if ( A->JA[l] == k ) break;
00412         }
00413         if ( A->val[l] > 0 ) {
00414             P->val[j] = -beta * A->val[l] / aii;
00415         }
00416         else {
00417             P->val[j] = -alpha * A->val[l] / aii;
00418         }
00419     }
00420
00421 } // end if vec
00422
00423 else if ( vec[i] == CGPT ) { // coarse grid nodes
00424     P->val[P->IA[i]] = 1.0;
00425 }
00426 }
00427 }
00428
00429 // Step 2. Generate coarse level indices and set values of P.JA
00430 for ( index = i = 0; i < row; ++i ) {
00431     if ( vec[i] == CGPT ) cindex[i] = index++;
00432 }
00433 P->col = index;
00434
00435 if (use_omp) {
00436 #ifdef _OPENMP
00437     stride_i = P->IA[P->row]/nthreads;
00438 #pragma omp parallel private(myid,mybegin,myend,i,j) num_threads(nthreads)
00439     {
00440         myid = omp_get_thread_num();
00441         mybegin = myid*stride_i;
00442         if ( myid < nthreads-1 ) myend = mybegin+stride_i;
00443         else myend = P->IA[P->row];
00444         for ( i = mybegin; i < myend; ++i ) {
00445             j = P->JA[i];
00446             P->JA[i] = cindex[j];
00447         }
00448     }
00449 #endif
00450 }
00451 else {
00452     for ( i = 0; i < P->nnz; ++i ) {
00453         j = P->JA[i];
00454         P->JA[i] = cindex[j];
00455     }
00456 }
00457
00458 // clean up
00459 fasp_mem_free(cindex); cindex = NULL;
00460
00461 // Step 3. Truncate the prolongation operator to reduce cost
00462 amg_interp_trunc(P, param);
00463 }
00464
00484 static void interp_STD (dCSRmat *A,
00485                        ivector *vertices,
00486                        dCSRmat *P,
00487                        iCSRmat *S,
00488                        AMG_param *param)
00489 {
00490     const INT row = A->row;
00491     INT *vec = vertices->val;
00492
00493     // local variables
00494     INT i, j, k, l, m, index;
00495     REAL alpha = 1.0, factor, alN, alP;
00496     REAL akk, ak1, aik, aki;
00497
00498     // indices for coarse neighbor node for every node
00499     INT *cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));
00500
00501     // indices from column number to index in nonzeros in i-th row
00502     INT *rindi = (INT *)fasp_mem_calloc(2*row, sizeof(INT));
00503 }

```

```

00504 // indices from column number to index in nonzeros in k-th row
00505 INT * rindk = (INT *)fasp_mem_calloc(2*row, sizeof(INT));
00506
00507 // sums of strongly connected C neighbors
00508 REAL * csum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00509
00510 #if RS_C1
00511 // sums of all neighbors except ISPT
00512 REAL * psum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00513 #endif
00514
00515 // sums of all neighbors
00516 REAL * nsum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00517
00518 // diagonal entries
00519 REAL * diag = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00520
00521 // coefficients hat a_ij for relevant CGPT of the i-th node
00522 REAL * Ahat = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00523
00524 // Step 0. Prepare diagonal, Cs-sum, and N-sum
00525 fasp_iarray_set(row, cindex, -1);
00526 fasp_darray_set(row, csum, 0.0);
00527 fasp_darray_set(row, nsum, 0.0);
00528
00529 for ( i = 0; i < row; i++ ) {
00530
00531 // set flags for strong-connected C nodes
00532 for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
00533 k = S->JA[j];
00534 if ( vec[k] == CGPT ) cindex[k] = i;
00535 }
00536
00537 for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
00538 k = A->JA[j];
00539
00540 if ( cindex[k] == i ) csum[i] += A->val[j]; // strong C-couplings
00541
00542 if ( k == i ) diag[i] = A->val[j];
00543 #if RS_C1
00544 else {
00545 nsum[i] += A->val[j];
00546 if ( vec[k] != ISPT ) {
00547 psum[i] += A->val[j];
00548 }
00549 }
00550 #else
00551 else nsum[i] += A->val[j];
00552 #endif
00553 }
00554
00555 }
00556
00557 // Step 1. Fill in values for interpolation operator P
00558 for ( i = 0; i < row; i++ ) {
00559
00560 if ( vec[i] == FGPT ) {
00561 #if RS_C1
00562 alN = psum[i];
00563 #else
00564 alN = nsum[i];
00565 #endif
00566 alP = csum[i];
00567
00568 // form the reverse indices for i-th row
00569 for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) rindi[A->JA[j]] = j;
00570
00571 // clean up Ahat for relevant nodes only
00572 for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) Ahat[P->JA[j]] = 0.0;
00573
00574 // set values of Ahat
00575 Ahat[i] = diag[i];
00576
00577 for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
00578
00579 k = S->JA[j]; aik = A->val[rindi[k]];
00580
00581 if ( vec[k] == CGPT ) Ahat[k] += aik;
00582
00583 else if ( vec[k] == FGPT ) {
00584

```

```

00585         akk = diag[k];
00586
00587         // form the reverse indices for k-th row
00588         for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) rindk[A->JA[m]] = m;
00589
00590         factor = aik / akk;
00591
00592         // visit the strong-connected C neighbors of k, compute
00593         // Ahat in the i-th row, set aki if found
00594         aki = 0.0;
00595         #if 0
00596         // modified by Xiaoqiang Yue 12/25/2013
00597         for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
00598             l = S->JA[m];
00599             ak1 = A->val[rindk[l]];
00600             if ( vec[l] == CGPT ) Ahat[l] -= factor * ak1;
00601             else if ( l == i ) {
00602                 aki = ak1; Ahat[l] -= factor * aki;
00603             }
00604         } // end for m
00605         #else
00606         for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) {
00607             if ( A->JA[m] == i ) {
00608                 aki = A->val[m];
00609                 Ahat[i] -= factor * aki;
00610             }
00611         } // end for m
00612         #endif
00613         for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
00614             l = S->JA[m];
00615             ak1 = A->val[rindk[l]];
00616             if ( vec[l] == CGPT ) Ahat[l] -= factor * ak1;
00617         } // end for m
00618
00619         // compute Cs-sum and N-sum for Ahat
00620         alN -= factor * (nsum[k]-aki+akk);
00621         alP -= factor * csum[k];
00622     } // end if vec[k]
00623
00624 } // end for j
00625
00626 // Originally: alpha = alN/alP, do this only if P is not empty!
00627 if ( P->IA[i] < P->IA[i+1] ) alpha = alN/alP;
00628
00629 // How about positive entries? --Chensong
00630 for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) {
00631     k = P->JA[j];
00632     P->val[j] = -alpha*Ahat[k]/Ahat[i];
00633 }
00634
00635 }
00636
00637 else if ( vec[i] == CGPT ) {
00638     P->val[P->IA[i]] = 1.0;
00639 }
00640
00641 } // end for i
00642
00643 // Step 2. Generate coarse level indices and set values of P.JA
00644 for ( index = i = 0; i < row; ++i ) {
00645     if ( vec[i] == CGPT ) cindex[i] = index++;
00646 }
00647 P->col = index;
00648
00649 #ifdef _OPENMP
00650 #pragma omp parallel for private(i,j) if(P->IA[P->row]>OPENMP_HOLDS)
00651 #endif
00652 for ( i = 0; i < P->IA[P->row]; ++i ) {
00653     j = P->JA[i];
00654     P->JA[i] = cindex[j];
00655 }
00656
00657 // clean up
00658 fasp_mem_free(cindex); cindex = NULL;
00659 fasp_mem_free(rindi); rindi = NULL;
00660 fasp_mem_free(rindk); rindk = NULL;
00661 fasp_mem_free(nsum); nsum = NULL;
00662 fasp_mem_free(csum); csum = NULL;
00663 fasp_mem_free(diag); diag = NULL;
00664 fasp_mem_free(Ahat); Ahat = NULL;
00665

```



```

00666 #if RS_C1
00667     fasp_mem_free(psum);    psum = NULL;
00668 #endif
00669
00670 // Step 3. Truncate the prolongation operator to reduce cost
00671 amg_interp_trunc(P, param);
00672 }
00673
00691 static void interp_EXT (dCSRmat    *A,
00692                        ivector    *vertices,
00693                        dCSRmat    *P,
00694                        iCSRmat    *S,
00695                        AMG_param  *param)
00696 {
00697     const INT    row    = A->row;
00698     INT          *vec    = vertices->val;
00699
00700     // local variables
00701     INT    i, j, k, l, m, index;
00702     REAL    alpha = 1.0, factor, alN, alP;
00703     REAL    akk, akL, aik, aki;
00704
00705     // indices for coarse neighbor node for every node
00706     INT * cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));
00707
00708     // indices from column number to index in nonzeros in i-th row
00709     INT * rindi = (INT *)fasp_mem_calloc(2*row, sizeof(INT));
00710
00711     // indices from column number to index in nonzeros in k-th row
00712     INT * rindk = (INT *)fasp_mem_calloc(2*row, sizeof(INT));
00713
00714     // sums of strongly connected C neighbors
00715     REAL * csum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00716
00717 #if RS_C1
00718     // sums of all neighbors except ISPT
00719     REAL * psum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00720 #endif
00721     // sums of all neighbors
00722     REAL * nsum = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00723
00724     // diagonal entries
00725     REAL * diag = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00726
00727     // coefficients hat a_ij for relevant CGPT of the i-th node
00728     REAL * Ahat = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
00729
00730     // Step 0. Prepare diagonal, Cs-sum, and N-sum
00731     fasp_iarray_set(row, cindex, -1);
00732     fasp_darray_set(row, csum, 0.0);
00733     fasp_darray_set(row, nsum, 0.0);
00734
00735     for ( i = 0; i < row; i++ ) {
00736
00737         // set flags for strong-connected C nodes
00738         for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
00739             k = S->JA[j];
00740             if ( vec[k] == CGPT ) cindex[k] = i;
00741         }
00742
00743         for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
00744             k = A->JA[j];
00745
00746             if ( cindex[k] == i ) csum[i] += A->val[j]; // strong C-couplings
00747
00748             if ( k == i ) diag[i] = A->val[j];
00749 #if RS_C1
00750             else {
00751                 nsum[i] += A->val[j];
00752                 if ( vec[k] != ISPT ) {
00753                     psum[i] += A->val[j];
00754                 }
00755             }
00756 #else
00757             else
00758                 nsum[i] += A->val[j];
00759 #endif
00760         }
00761     }
00762
00763     // Step 1. Fill in values for interpolation operator P

```

```

00764     for ( i = 0; i < row; i++ ) {
00765
00766         if ( vec[i] == FGPT ) {
00767 #if RS_C1
00768             alN = psum[i];
00769 #else
00770             alN = nsum[i];
00771 #endif
00772             alP = csum[i];
00773
00774             // form the reverse indices for i-th row
00775             for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) rindi[A->JA[j]] = j;
00776
00777             // clean up Ahat for relevant nodes only
00778             for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) Ahat[P->JA[j]] = 0.0;
00779
00780             // set values of Ahat
00781             Ahat[i] = diag[i];
00782
00783             for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
00784
00785                 k = S->JA[j]; aik = A->val[rindi[k]];
00786
00787                 if ( vec[k] == CGPT ) Ahat[k] += aik;
00788
00789                 else if ( vec[k] == FGPT ) {
00790
00791                     akk = diag[k];
00792
00793                     // form the reverse indices for k-th row
00794                     for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) rindk[A->JA[m]] = m;
00795
00796                     factor = aik / akk;
00797
00798                     // visit the strong-connected C neighbors of k, compute
00799                     // Ahat in the i-th row, set aki if found
00800                     aki = 0.0;
00801 #if 0
00802                     // modified by Xiaoqiang Yue 12/25/2013
00803                     for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
00804                         l = S->JA[m];
00805                         ak1 = A->val[rindk[l]];
00806                         if ( vec[l] == CGPT ) Ahat[l] -= factor * ak1;
00807                         else if ( l == i ) {
00808                             aki = ak1; Ahat[l] -= factor * aki;
00809                         }
00810                     } // end for m
00811 #else
00812                     for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) {
00813                         if ( A->JA[m] == i ) {
00814                             aki = A->val[m];
00815                             Ahat[i] -= factor * aki;
00816                         }
00817                     } // end for m
00818 #endif
00819                     for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
00820                         l = S->JA[m];
00821                         ak1 = A->val[rindk[l]];
00822                         if ( vec[l] == CGPT ) Ahat[l] -= factor * ak1;
00823                     } // end for m
00824
00825                     // compute Cs-sum and N-sum for Ahat
00826                     alN -= factor * (nsum[k]-aki+akk);
00827                     alP -= factor * csum[k];
00828
00829                 } // end if vec[k]
00830
00831             } // end for j
00832
00833             // Originally: alpha = alN/alP, do this only if P is not empty!
00834             if ( P->IA[i] < P->IA[i+1] ) alpha = alN/alP;
00835
00836             // How about positive entries? --Chensong
00837             for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) {
00838                 k = P->JA[j];
00839                 P->val[j] = -alpha*Ahat[k]/Ahat[i];
00840             }
00841         }
00842
00843         else if ( vec[i] == CGPT ) {
00844             P->val[P->IA[i]] = 1.0;

```

```

00845     }
00846
00847     } // end for i
00848
00849     // Step 2. Generate coarse level indices and set values of P.JA
00850     for ( index = i = 0; i < row; ++i ) {
00851         if ( vec[i] == CGPT ) cindex[i] = index++;
00852     }
00853     P->col = index;
00854
00855 #ifdef _OPENMP
00856 #pragma omp parallel for private(i,j) if(P->IA[P->row]>OPENMP_HOLDS)
00857 #endif
00858     for ( i = 0; i < P->IA[P->row]; ++i ) {
00859         j = P->JA[i];
00860         P->JA[i] = cindex[j];
00861     }
00862
00863     // clean up
00864     fasp_mem_free(cindex); cindex = NULL;
00865     fasp_mem_free(rindi); rindi = NULL;
00866     fasp_mem_free(rindk); rindk = NULL;
00867     fasp_mem_free(nsum); nsum = NULL;
00868     fasp_mem_free(csum); csum = NULL;
00869     fasp_mem_free(diag); diag = NULL;
00870     fasp_mem_free(Ahat); Ahat = NULL;
00871
00872 #if RS_C1
00873     fasp_mem_free(psum); psum = NULL;
00874 #endif
00875
00876     // Step 3. Truncate the prolongation operator to reduce cost
00877     amg_interp_trunc(P, param);
00878 }
00879
00880 /*-----*/
00881 /*--          End of File          --*/
00882 /*-----*/

```

9.141 PreAMGInterpEM.c File Reference

Interpolation operators for AMG based on energy-min.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- void [fasp_amg_interp_em](#) (dCSRmat *A, ivector *vertices, dCSRmat *P, AMG_param *param)
Energy-min interpolation.

9.141.1 Detailed Description

Interpolation operators for AMG based on energy-min.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxThreads.c](#), [AuxVector.c](#), [BlaSmallMatLU.c](#), [BlaSparseCSR.c](#), [KryPcg.c](#), and [PreCSR.c](#)

Reference: J. Xu and L. Zikatanov On An Energy Minimizing Basis in Algebraic Multigrid Methods, Computing and visualization in sciences, 2003

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGInterpEM.c](#).

9.141.2 Function Documentation

9.141.2.1 fasp_amg_interp_em()

```
void fasp_amg_interp_em (
    dCSRmat * A,
    ivector * vertices,
    dCSRmat * P,
    AMG_param * param )
```

Energy-min interpolation.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix (index starts from 0)
<i>vertices</i>	Pointer to the indicator of CF splitting on fine or coarse grid
<i>P</i>	Pointer to the dCSRmat matrix of resulted interpolation
<i>param</i>	Pointer to AMG_param : AMG parameters

Author

Shuo Zhang, Xuehai Huang

Date

04/04/2010

Modified by Chunsheng Feng, Zheng Li on 10/17/2012: add OMP support Modified by Chensong Zhang on 05/14/2013: reconstruct the code

Definition at line 63 of file [PreAMGInterpEM.c](#).

9.142 PreAMGInterpEM.c

[Go to the documentation of this file.](#)

```
00001
00020 #include <math.h>
00021 #include <time.h>
00022
00023 #ifdef _OPENMP
00024 #include <omp.h>
00025 #endif
00026
00027 #include "fasp.h"
00028 #include "fasp_functs.h"
00029
00030 /*-----*/
00031 /*--  Declare Private Functions  --*/
00032 /*-----*/
00033
00034 static SHORT getiteval(dCSRmat *, dCSRmat *);
00035 static SHORT invden(INT, REAL *, REAL *);
00036 static SHORT get_block(dCSRmat *, INT, INT, INT *, INT *, REAL *, INT *);
00037 static SHORT gentisquare_nomass(dCSRmat *, INT, INT *, REAL *, INT *);
00038 static SHORT getinonefull(INT **, REAL **, INT *, INT, INT *, REAL *);
00039 static SHORT orderone(INT **, REAL **, INT *);
00040 static SHORT genintval(dCSRmat *, INT **, REAL **, INT, INT *, INT, INT, INT);
```

```

00041
00042 /*-----*/
00043 /*--      Public Functions      --*/
00044 /*-----*/
00045
00063 void fasp_amg_interp_em (dCSRmat    *A,
00064                        ivector    *vertices,
00065                        dCSRmat    *P,
00066                        AMG_param  *param)
00067 {
00068     INT    *vec = vertices->val;
00069     INT    *CoarseIndex = (INT *)fasp_mem_malloc(vertices->row, sizeof(INT));
00070     INT    i, j, index;
00071
00072     // generate indices for C-points
00073     for ( index = i = 0; i < vertices->row; ++i ) {
00074         if ( vec[i] == 1 ) {
00075             CoarseIndex[i] = index;
00076             index++;
00077         }
00078     }
00079
00080 #ifdef _OPENMP
00081 #pragma omp parallel for private(i,j) if(P->nnz>OPENMP_HOLDS)
00082 #endif
00083     for ( i = 0; i < P->nnz; ++i ) {
00084         j = P->JA[i];
00085         P->JA[i] = CoarseIndex[j];
00086     }
00087
00088     // clean up memory
00089     fasp_mem_free(CoarseIndex); CoarseIndex = NULL;
00090
00091     // main part
00092     getiteval(A, P);
00093 }
00094
00095 /*-----*/
00096 /*--      Private Functions      --*/
00097 /*-----*/
00098
00115 static SHORT invden (INT    nn,
00116                     REAL    *mat,
00117                     REAL    *invmat)
00118 {
00119     INT    i, j;
00120     SHORT    status = FASP_SUCCESS;
00121
00122 #ifdef _OPENMP
00123     // variables for OpenMP
00124     INT myid, mybegin, myend;
00125     INT nthreads = fasp_get_num_threads();
00126 #endif
00127
00128     INT *pivot=(INT *)fasp_mem_malloc(nn,sizeof(INT));
00129     REAL *rhs=(REAL *)fasp_mem_malloc(nn,sizeof(REAL));
00130     REAL *sol=(REAL *)fasp_mem_malloc(nn,sizeof(REAL));
00131
00132     fasp_smat_lu_decomp(mat,pivot,nn);
00133
00134 #ifdef _OPENMP
00135 #pragma omp parallel for private(myid,mybegin,myend,i,j) if(nn>OPENMP_HOLDS)
00136     for (myid=0; myid<nthreads; ++myid) {
00137         fasp_get_start_end(myid, nthreads, nn, &mybegin, &myend);
00138         for (i=mybegin; i<myend; ++i) {
00139             #else
00140                 for (i=0;i<nn;++i) {
00141             #endif
00142                 for (j=0;j<nn;++j) rhs[j]=0.;
00143                 rhs[i]=1.;
00144                 fasp_smat_lu_solve(mat,rhs,pivot,sol,nn);
00145                 for (j=0;j<nn;++j) invmat[i*nn+j]=sol[j];
00146 #ifdef _OPENMP
00147         }
00148     }
00149 #else
00150 }
00151 #endif
00152
00153     fasp_mem_free(pivot); pivot = NULL;
00154     fasp_mem_free(rhs);    rhs = NULL;

```

```

00155     fasp_mem_free(sol);   sol   = NULL;
00156
00157     return status;
00158 }
00159
00181 static SHORT get_block (dCSRmat  *A,
00182                        INT        m,
00183                        INT        n,
00184                        INT        *rows,
00185                        INT        *cols,
00186                        REAL       *Aloc,
00187                        INT        *mask)
00188 {
00189     INT i, j, k, iloc;
00190
00191 #ifdef _OPENMP
00192     // variables for OpenMP
00193     INT myid, mybegin, myend;
00194     INT nthreads = fasp_get_num_threads();
00195 #endif
00196     memset(Aloc, 0x0, sizeof(REAL)*m*n);
00197
00198 #ifdef _OPENMP
00199 #pragma omp parallel for if(n>OPENMP_HOLDS) private(j)
00200 #endif
00201     for ( j=0; j<n; ++j ) {
00202         mask[cols[j]] = j; // initialize mask, mask stores C indices 0,1,...
00203     }
00204
00205 #ifdef _OPENMP
00206 #pragma omp parallel for private(myid,mybegin,myend,i,j,k,iloc) if(m>OPENMP_HOLDS)
00207 #endif
00208     for ( myid=0; myid<nthreads; ++myid ) {
00209         fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00210         for ( i=mybegin; i<myend; ++i ) {
00211 #else
00212         for ( i=0; i<m; ++i ) {
00213 #endif
00214             iloc=rows[i];
00215             for ( k=A->IA[iloc]; k<A->IA[iloc+1]; ++k ) {
00216                 j = A->JA[k];
00217                 if (mask[j]>=0) Aloc[i*n+mask[j]]=A->val[k];
00218             } /* end for k */
00219 #ifdef _OPENMP
00220         }
00221     }
00222 #else
00223     } /* enf for i */
00224 #endif
00225
00226 #ifdef _OPENMP
00227 #pragma omp parallel for if(n>OPENMP_HOLDS) private(j)
00228 #endif
00229     for ( j=0; j<n; ++j ) mask[cols[j]] = -1; // re-initialize mask
00230
00231     return FASP_SUCCESS;
00232 }
00233
00250 static SHORT gentisquare_nomass (dCSRmat  *A,
00251                                INT        mm,
00252                                INT        *Ii,
00253                                REAL       *ima,
00254                                INT        *mask)
00255 {
00256     SHORT status = FASP_SUCCESS;
00257
00258     REAL *ms = (REAL *)fasp_mem_calloc(mm*mm,sizeof(REAL));
00259
00260     get_block(A,mm,mm,Ii,Ii,ms,mask);
00261
00262     status = invden(mm,ms,ima);
00263
00264     fasp_mem_free(ms); ms = NULL;
00265
00266     return status;
00267 }
00268
00288 static SHORT getinonefull (INT    **mat,
00289                           REAL    **matval,
00290                           INT     *lengths,
00291                           INT     mm,
```

```

00292             INT      *Ii,
00293             REAL      *ima)
00294 {
00295     INT tniz,i,j;
00296
00297 #ifdef _OPENMP
00298     // variables for OpenMP
00299     INT myid, mybegin, myend;
00300     INT nthreads = fasp_get_num_threads();
00301 #endif
00302
00303     tniz=lengths[1];
00304
00305 #ifdef _OPENMP
00306 #pragma omp parallel for private(myid,mybegin,myend,i,j) if(mm>OPENMP_HOLDS)
00307     for (myid=0; myid<nthreads; ++myid) {
00308         fasp_get_start_end(myid, nthreads, mm, &mybegin, &myend);
00309         for (i=mybegin; i<myend; ++i) {
00310 #else
00311             for (i=0; i<mm; ++i) {
00312 #endif
00313                 for (j=0; j<mm; ++j) {
00314                     mat[0][tniz+i*mm+j]=Ii[i];
00315                     mat[1][tniz+i*mm+j]=Ii[j];
00316                     matval[0][tniz+i*mm+j]=ima[i*mm+j];
00317                 }
00318 #ifdef _OPENMP
00319             }
00320         }
00321 #else
00322     }
00323 #endif
00324     lengths[1]=tniz+mm*mm;
00325
00326     return FASP_SUCCESS;
00327 }
00328
00345 static SHORT orderone (INT      **mat,
00346                        REAL      **matval,
00347                        INT      *lengths)
00348 //     lengths[0] for the number of rows
00349 //     lengths[1] for the number of cols
00350 //     lengths[2] for the number of nonzeros
00351 {
00352     INT *rows[2],*cols[2],nns[2],tnizs[2];
00353     REAL *vals[2];
00354     SHORT status = FASP_SUCCESS;
00355     INT tniz,i;
00356
00357     nns[0]=lengths[0];
00358     nns[1]=lengths[1];
00359     tnizs[0]=lengths[2];
00360     tniz=lengths[2];
00361
00362     rows[0]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
00363     cols[0]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
00364     vals[0]=(REAL *)fasp_mem_calloc(tniz,sizeof(REAL));
00365
00366 #ifdef _OPENMP
00367 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00368 #endif
00369     for (i=0; i<tniz; ++i) {
00370         rows[0][i]=mat[0][i];
00371         cols[0][i]=mat[1][i];
00372         vals[0][i]=matval[0][i];
00373     }
00374
00375     rows[1]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
00376     cols[1]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
00377     vals[1]=(REAL *)fasp_mem_calloc(tniz,sizeof(REAL));
00378
00379     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00380
00381     // all the nonzeros with same col are gathering together
00382
00383 #ifdef _OPENMP
00384 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00385 #endif
00386     for (i=0; i<tniz; ++i) {
00387         rows[0][i]=rows[1][i];
00388         cols[0][i]=cols[1][i];

```

```

00389         vals[0][i]=vals[1][i];
00390     }
00391     tnizs[1]=nns[0];
00392     nns[0]=nns[1];
00393     nns[1]=tnizs[1];
00394     tnizs[1]=tnizs[0];
00395     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00396
00397     // all the nonzeros with same col and row are gathering together
00398 #ifdef _OPENMP
00399 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00400 #endif
00401     for (i=0;i<tniz;++i) {
00402         rows[0][i]=rows[1][i];
00403         cols[0][i]=cols[1][i];
00404         vals[0][i]=vals[1][i];
00405     }
00406     tnizs[1]=nns[0];
00407     nns[0]=nns[1];
00408     nns[1]=tnizs[1];
00409     tnizs[1]=tnizs[0];
00410
00411     tniz=tnizs[0];
00412     for (i=0;i<tniz-1;++i) {
00413         if (rows[0][i]==rows[0][i+1]&&cols[0][i]==cols[0][i+1]) {
00414             vals[0][i+1]+=vals[0][i];
00415             rows[0][i]=nns[0];
00416             cols[0][i]=nns[1];
00417         }
00418     }
00419     nns[0]=nns[0]+1;
00420     nns[1]=nns[1]+1;
00421
00422     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00423
00424 #ifdef _OPENMP
00425 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00426 #endif
00427     for (i=0;i<tniz;++i) {
00428         rows[0][i]=rows[1][i];
00429         cols[0][i]=cols[1][i];
00430         vals[0][i]=vals[1][i];
00431     }
00432     tnizs[1]=nns[0];
00433     nns[0]=nns[1];
00434     nns[1]=tnizs[1];
00435     tnizs[1]=tnizs[0];
00436
00437     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00438
00439 #ifdef _OPENMP
00440 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00441 #endif
00442     for (i=0;i<tniz;++i) {
00443         rows[0][i]=rows[1][i];
00444         cols[0][i]=cols[1][i];
00445         vals[0][i]=vals[1][i];
00446     }
00447     tnizs[1]=nns[0];
00448     nns[0]=nns[1];
00449     nns[1]=tnizs[1];
00450     tnizs[1]=tnizs[0];
00451
00452     tniz=0;
00453     for (i=0;i<tnizs[0];++i)
00454         if (rows[0][i]<nns[0]-1) tniz++;
00455
00456 #ifdef _OPENMP
00457 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00458 #endif
00459     for (i=0;i<tniz;++i) {
00460         mat[0][i]=rows[0][i];
00461         mat[1][i]=cols[0][i];
00462         matval[0][i]=vals[0][i];
00463     }
00464     nns[0]=nns[0]-1;
00465     nns[1]=nns[1]-1;
00466     lengths[0]=nns[0];
00467     lengths[1]=nns[1];
00468     lengths[2]=tniz;
00469

```



```

00470     fasp_mem_free(rows[0]); rows[0] = NULL;
00471     fasp_mem_free(rows[1]); rows[1] = NULL;
00472     fasp_mem_free(cols[0]); cols[0] = NULL;
00473     fasp_mem_free(cols[1]); cols[1] = NULL;
00474     fasp_mem_free(vals[0]); vals[0] = NULL;
00475     fasp_mem_free(vals[1]); vals[1] = NULL;
00476
00477     return(status);
00478 }
00479
00511 static SHORT genintval (dCSRmat *A,
00512                        INT **itmat,
00513                        REAL **itmatval,
00514                        INT ittniz,
00515                        INT *isol,
00516                        INT numiso,
00517                        INT nf,
00518                        INT nc)
00519 {
00520     INT *Ii=NULL, *mask=NULL;
00521     REAL *ima=NULL, *pex=NULL, **imas=NULL;
00522     INT **mat=NULL;
00523     REAL **matval;
00524     INT lengths[3];
00525     dCSRmat T;
00526     INT tniz;
00527     dvector sol, rhs;
00528
00529     INT mm,sum,i,j,k;
00530     INT *iz,*izs,*izt,*izts;
00531     SHORT status=FASP_SUCCESS;
00532
00533     mask=(INT *) fasp_mem_calloc(nf,sizeof(INT));
00534     iz=(INT *) fasp_mem_calloc(nc,sizeof(INT));
00535     izs=(INT *) fasp_mem_calloc(nc,sizeof(INT));
00536     izt=(INT *) fasp_mem_calloc(nf,sizeof(INT));
00537     izts=(INT *) fasp_mem_calloc(nf,sizeof(INT));
00538
00539     fasp_iarray_set(nf, mask, -1);
00540
00541     memset(iz, 0, sizeof(INT)*nc);
00542
00543 #ifdef _OPENMP
00544 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
00545 #endif
00546     for (i=0;i<ittniz;++i) iz[itmat[0][i]]++;
00547
00548     izs[0]=0;
00549     for (i=1;i<nc;++i) izs[i]=izs[i-1]+iz[i-1];
00550
00551     sum = 0;
00552 #ifdef _OPENMP
00553 #pragma omp parallel for reduction(+:sum) if(nc>OPENMP_HOLDS) private(i)
00554 #endif
00555     for (i=0;i<nc;++i) sum+=iz[i]*iz[i];
00556
00557     imas=(REAL **) fasp_mem_calloc(nc,sizeof(REAL *));
00558
00559     for (i=0;i<nc;++i) {
00560         imas[i]=(REAL *) fasp_mem_calloc(iz[i]*iz[i],sizeof(REAL));
00561     }
00562
00563     mat=(INT **) fasp_mem_calloc(2,sizeof(INT *));
00564     mat[0]=(INT *) fasp_mem_calloc((sum+numiso),sizeof(INT));
00565     mat[1]=(INT *) fasp_mem_calloc((sum+numiso),sizeof(INT));
00566     matval=(REAL **) fasp_mem_calloc(1,sizeof(REAL *));
00567     matval[0]=(REAL *) fasp_mem_calloc(sum+numiso,sizeof(REAL));
00568
00569     lengths[1]=0;
00570
00571     for (i=0;i<nc;++i) {
00572
00573         mm=iz[i];
00574         Ii=(INT *) fasp_mem_realloc(Ii,mm*sizeof(INT));
00575
00576 #ifdef _OPENMP
00577 #pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
00578 #endif
00579         for (j=0;j<mm;++j) Ii[j]=itmat[1][izs[i]+j];
00580
00581         ima=(REAL *) fasp_mem_realloc(ima,mm*mm*sizeof(REAL));

```

```

00582
00583     gentisquare_nomass(A,mm,Ii,ima,mask);
00584
00585     getinonefull(mat,matval,lengths,mm,Ii,ima);
00586
00587 #ifdef _OPENMP
00588 #pragma omp parallel for if(mm*mm>OPENMP_HOLDS) private(j)
00589 #endif
00590     for (j=0;j<mm*mm;++j) imas[i][j]=ima[j];
00591 }
00592
00593 #ifdef _OPENMP
00594 #pragma omp parallel for if(numiso>OPENMP_HOLDS) private(i)
00595 #endif
00596     for (i=0;i<numiso;++i) {
00597         mat[0][sum+i]=isol[i];
00598         mat[1][sum+i]=isol[i];
00599         matval[0][sum+i]=1.0;
00600     }
00601
00602     lengths[0]=nf;
00603     lengths[2]=lengths[1]+numiso;
00604     lengths[1]=nf;
00605     orderone(mat,matval,lengths);
00606     tniz=lengths[2];
00607
00608     sol.row=nf;
00609     sol.val=(REAL*) fasp_mem_calloc(nf,sizeof(REAL));
00610
00611     memset(izt, 0, sizeof(INT)*nf);
00612
00613 #ifdef _OPENMP
00614 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
00615 #endif
00616     for (i=0;i<tniz;++i) izt[mat[0][i]]++;
00617
00618     T.IA=(INT*) fasp_mem_calloc((nf+1),sizeof(INT));
00619
00620     T.row=nf;
00621     T.col=nf;
00622     T.nnz=tniz;
00623     T.IA[0]=0;
00624     for (i=1;i<nf+1;++i) T.IA[i]=T.IA[i-1]+izt[i-1];
00625
00626     T.JA=(INT*) fasp_mem_calloc(tniz,sizeof(INT));
00627
00628 #ifdef _OPENMP
00629 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(j)
00630 #endif
00631     for (j=0;j<tniz;++j) T.JA[j]=mat[1][j];
00632
00633     T.val=(REAL*) fasp_mem_calloc(tniz,sizeof(REAL));
00634
00635 #ifdef _OPENMP
00636 #pragma omp parallel for if(tniz>OPENMP_HOLDS) private(j)
00637 #endif
00638     for (j=0;j<tniz;++j) T.val[j]=matval[0][j];
00639
00640     rhs.val=(REAL*) fasp_mem_calloc(nf,sizeof(REAL));
00641
00642 #ifdef _OPENMP
00643 #pragma omp parallel for if(nf>OPENMP_HOLDS) private(i)
00644 #endif
00645     for (i=0;i<nf;++i) rhs.val[i]=1.0;
00646     rhs.row=nf;
00647
00648     // setup preconditioner
00649     dvector diag; fasp_dcsr_getdiag(0,&T,&diag);
00650
00651     precondition pc;
00652     pc.data = &diag;
00653     pc.fct = fasp_precond_diag;
00654
00655     status = fasp_solver_dcsr_pcg(&T,&rhs,&sol,&pc,1e-3,100,STOP_REL_RES,PRINT_NONE);
00656
00657     for (i=0;i<nc;++i) {
00658         mm=iz[i];
00659
00660         ima=(REAL *) fasp_mem_realloc(ima,mm*mm*sizeof(REAL));
00661
00662         pex=(REAL *) fasp_mem_realloc(pex,mm*sizeof(REAL));

```

```

00663
00664     Ii=(INT *) fasp_mem_realloc(Ii,mm*sizeof(INT));
00665
00666 #ifdef _OPENMP
00667 #pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
00668 #endif
00669     for (j=0;j<mm;++j) Ii[j]=itmat[1][izs[i]+j];
00670
00671 #ifdef _OPENMP
00672 #pragma omp parallel for if(mm*mm>OPENMP_HOLDS) private(j)
00673 #endif
00674     for (j=0;j<mm*mm;++j) ima[j]=imas[i][j];
00675
00676 #ifdef _OPENMP
00677 #pragma omp parallel for if(mm>OPENMP_HOLDS) private(k,j)
00678 #endif
00679     for (k=0;k<mm;++k) {
00680         for (pex[k]=j=0;j<mm;++j) pex[k]+=ima[k*mm+j]*sol.val[Ii[j]];
00681     }
00682 #ifdef _OPENMP
00683 #pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
00684 #endif
00685     for (j=0;j<mm;++j) itmatval[0][izs[i]+j]=pex[j];
00686
00687 }
00688
00689 fasp_mem_free(ima); ima = NULL;
00690 fasp_mem_free(pex); pex = NULL;
00691 fasp_mem_free(Ii); Ii = NULL;
00692 fasp_mem_free(mask); mask = NULL;
00693 fasp_mem_free(iz); iz = NULL;
00694 fasp_mem_free(izs); izs = NULL;
00695 fasp_mem_free(izt); izt = NULL;
00696 fasp_mem_free(izts); izts = NULL;
00697 fasp_mem_free(mat[0]); mat[0] = NULL;
00698 fasp_mem_free(mat[1]); mat[1] = NULL;
00699 fasp_mem_free(mat); mat = NULL;
00700 fasp_mem_free(matval[0]); matval[0] = NULL;
00701 fasp_mem_free(matval); matval = NULL;
00702 for ( i=0; i<nc; ++i ) {fasp_mem_free(imas[i]); imas[i] = NULL;}
00703 fasp_mem_free(imas); imas = NULL;
00704
00705 fasp_dcsr_free(&T);
00706 fasp_dvec_free(&rhs);
00707 fasp_dvec_free(&sol);
00708 fasp_dvec_free(&diag);
00709
00710 return status;
00711 }
00712
00727 static SHORT getiteval (dCSRmat *A,
00728                        dCSRmat *it)
00729 {
00730     INT nf,nc,ittniz;
00731     INT *itmat[2];
00732     REAL **itmatval;
00733     INT *rows[2],*cols[2];
00734     REAL *vals[2];
00735     INT nns[2],tnizs[2];
00736     INT i,j,numiso;
00737     INT *isol;
00738     SHORT status = FASP_SUCCESS;
00739
00740     nf=A->row;
00741     nc=it->col;
00742     ittniz=it->IA[nf];
00743
00744     itmat[0]=(INT *) fasp_mem_calloc(ittniz,sizeof(INT));
00745     itmat[1]=(INT *) fasp_mem_calloc(ittniz,sizeof(INT));
00746     itmatval=(REAL **) fasp_mem_calloc(1,sizeof(REAL *));
00747     itmatval[0]=(REAL *) fasp_mem_calloc(ittniz,sizeof(REAL));
00748     isol=(INT *) fasp_mem_calloc(nf,sizeof(INT));
00749
00750     numiso=0;
00751     for (i=0;i<nf;++i) {
00752         if (it->IA[i]==it->IA[i+1]) {
00753             isol[numiso]=i;
00754             numiso++;
00755         }
00756     }
00757

```

```

00758 #ifdef _OPENMP
00759 #pragma omp parallel for if(nf>OPENMP_HOLDS) private(i,j)
00760 #endif
00761     for (i=0;i<nf;++i) {
00762         for (j=it->IA[i];j<it->IA[i+1];++j) itmat[0][j]=i;
00763     }
00764
00765 #ifdef _OPENMP
00766 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(j)
00767 #endif
00768     for (j=0;j<ittniz;++j) {
00769         itmat[1][j]=it->JA[j];
00770         itmatval[0][j]=it->val[j];
00771     }
00772
00773     rows[0]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
00774     cols[0]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
00775     vals[0]=(REAL *)fasp_mem_calloc(ittniz,sizeof(REAL));
00776
00777 #ifdef _OPENMP
00778 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
00779 #endif
00780     for (i=0;i<ittniz;++i) {
00781         rows[0][i]=itmat[0][i];
00782         cols[0][i]=itmat[1][i];
00783         vals[0][i]=itmat[0][i];
00784     }
00785
00786     nns[0]=nf;
00787     nns[1]=nc;
00788     tnizs[0]=ittniz;
00789
00790     rows[1]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
00791     cols[1]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
00792     vals[1]=(REAL *)fasp_mem_calloc(ittniz,sizeof(REAL));
00793
00794     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00795
00796 #ifdef _OPENMP
00797 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
00798 #endif
00799     for (i=0;i<ittniz;++i) {
00800         itmat[0][i]=rows[1][i];
00801         itmat[1][i]=cols[1][i];
00802         itmatval[0][i]=vals[1][i];
00803     }
00804     genintval(A,itmat,itmatval,ittniz,isol,numiso,nf,nc);
00805
00806 #ifdef _OPENMP
00807 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
00808 #endif
00809     for (i=0;i<ittniz;++i) {
00810         rows[0][i]=itmat[0][i];
00811         cols[0][i]=itmat[1][i];
00812         vals[0][i]=itmatval[0][i];
00813     }
00814     nns[0]=nc;
00815     nns[1]=nf;
00816     tnizs[0]=ittniz;
00817
00818     fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);
00819
00820 #ifdef _OPENMP
00821 #pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
00822 #endif
00823     for (i=0;i<ittniz;++i) it->val[i]=vals[1][i];
00824
00825     fasp_mem_free(isol); isol = NULL;
00826     fasp_mem_free(itmat[0]); itmat[0] = NULL;
00827     fasp_mem_free(itmat[1]); itmat[1] = NULL;
00828     fasp_mem_free(itmatval[0]); itmatval[0] = NULL;
00829     fasp_mem_free(itmatval); itmatval = NULL;
00830     fasp_mem_free(rows[0]); rows[0] = NULL;
00831     fasp_mem_free(rows[1]); rows[1] = NULL;
00832     fasp_mem_free(cols[0]); cols[0] = NULL;
00833     fasp_mem_free(cols[1]); cols[1] = NULL;
00834     fasp_mem_free(vals[0]); vals[0] = NULL;
00835     fasp_mem_free(vals[1]); vals[1] = NULL;
00836
00837     return status;
00838 }

```

```

00839
00840 /*-----*/
00841 /*--      End of File      --*/
00842 /*-----*/

```

9.143 PreAMGSetupCR.c File Reference

Brannick-Falgout compatible relaxation based AMG: SETUP phase.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_amg_setup_cr](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)

Set up phase of Brannick Falgout CR coarsening for classic AMG.

9.143.1 Detailed Description

Brannick-Falgout compatible relaxation based AMG: SETUP phase.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), and [PreAMGCoarsenCR.c](#)

Setup A, P, R and levels using the Compatible Relaxation coarsening for classic AMG interpolation

Reference: J. Brannick and R. Falgout Compatible relaxation and coarsening in AMG
Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Not working. Need to be fixed. –Chensong

Definition in file [PreAMGSetupCR.c](#).

9.143.2 Function Documentation

9.143.2.1 fasp_amg_setup_cr()

```

SHORT fasp_amg_setup_cr (
    AMG\_data * mgl,
    AMG\_param * param )

```

Set up phase of Brannick Falgout CR coarsening for classic AMG.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

James Brannick

Date

04/21/2010

Modified by Chensong Zhang on 05/10/2013: adjust the structure.

Definition at line 48 of file [PreAMGSetupCR.c](#).

9.144 PreAMGSetupCR.c

[Go to the documentation of this file.](#)

```

00001
00023 #include <math.h>
00024 #include <time.h>
00025
00026 #include "fasp.h"
00027 #include "fasp_funcs.h"
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00048 SHORT fasp_amg_setup_cr (AMG_data  *mgl,
00049                          AMG_param *param)
00050 {
00051     const SHORT prtlvl = param->print_level;
00052     const SHORT min_cdof = MAX(param->coarse_dof, 50);
00053     const INT    m      = mgl[0].A.row;
00054
00055     // local variables
00056     INT    i_0 = 0, i_n;
00057     SHORT  level = 0, status = FASP_SUCCESS;
00058     SHORT  max_levels = param->max_levels;
00059     REAL    setup_start, setup_end;
00060
00061     // The variable vertices stores level info (fine: 0; coarse: 1)
00062     ivector vertices = fasp_ivec_create(m);
00063
00064     fasp_gettime(&setup_start);
00065
00066 #if DEBUG_MODE > 0
00067     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00068     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
00069           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00070 #endif
00071
00072 #if DIAGONAL_PREF
00073     fasp_dcsr_diagpref(&mgl[0].A); // reorder each row to make diagonal appear first
00074 #endif
00075
00076     // Main AMG setup loop
00077     while ( (mgl[level].A.row > min_cdof) && (level < max_levels-1) ) {
00078
00079         /*-- Coarsen and form the structure of interpolation --*/
00080         i_n = mgl[level].A.row-1;
00081
00082         fasp_amg_coarsening_cr(i_0, i_n, &mgl[level].A, &vertices, param);
00083
00084         /*-- Form interpolation --*/
00085         /* 1. SPARSITY -- Form ip and jp */
00086         /* First a symbolic one
00087 then gather the list */
00088         /* 2. COEFFICIENTS -- Form P */
00089         // energymin(mgl[level].A, &vertices[level], mgl[level].P, param);
00090         // fasp_mem_free(vertices[level].val); vertices[level].val = NULL;
00091
00092         /*-- Form coarse level stiffness matrix --*/

```

```

00093         // fasp_dcsr_trans(mgl[level].P, mgl[level].R);
00094
00095         /*-- Form coarse level stiffness matrix --*/
00096         //fasp_blas_dcsr_rap(mgl[level].R, mgl[level].A, mgl[level].P, mgl[level+1].A);
00097
00098         ++level;
00099
00100 #if DIAGONAL_PREF
00101         fasp_dcsr_diagpref(&mgl[level].A); // reorder each row to make diagonal appear first
00102 #endif
00103     }
00104
00105     // setup total level number and current level
00106     mgl[0].num_levels = max_levels = level+1;
00107     mgl[0].w = fasp_dvec_create(m);
00108
00109     for ( level = 1; level < max_levels; ++level ) {
00110         INT mm = mgl[level].A.row;
00111         mgl[level].num_levels = max_levels;
00112         mgl[level].b = fasp_dvec_create(mm);
00113         mgl[level].x = fasp_dvec_create(mm);
00114         mgl[level].w = fasp_dvec_create(mm);
00115     }
00116
00117     if ( prtlvl > PRINT_NONE ) {
00118         fasp_gettime(&setup_end);
00119         fasp_amgcomplexity(mgl,prtlvl);
00120         fasp_cputime("Compatible relaxation setup", setup_end - setup_start);
00121     }
00122
00123     fasp_ivec_free(&vertices);
00124
00125 #if DEBUG_MODE > 0
00126     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00127 #endif
00128
00129     return status;
00130 }
00131
00132 /*-----*/
00133 /*--      End of File      --*/
00134 /*-----*/

```

9.145 PreAMGSetupRS.c File Reference

Ruge-Stuben AMG: SETUP phase.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"

```

Functions

- [SHORT fasp_amg_setup_rs](#) (AMG_data *mgl, AMG_param *param)
Setup phase of Ruge and Stuben's classic AMG.

9.145.1 Detailed Description

Ruge-Stuben AMG: SETUP phase.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaLUSetupCSR.c](#), [BlaSchwarzSetup.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), [PreAMGCoarsenRS.c](#), [PreAMGInterp.c](#), and [PreMGRecurAMLI.c](#)

Reference: Multigrid by U. Trottenberg, C. W. Oosterlee and A. Schuller Appendix P475 A.7 (by A. Brandt, P. Oswald and K. Stuben) Academic Press Inc., San Diego, CA, 2001.
Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGSetupRS.c](#).

9.145.2 Function Documentation

9.145.2.1 fasp_amg_setup_rs()

```
SHORT fasp_amg_setup_rs (
    AMG_data * mgl,
    AMG_param * param )
```

Setup phase of Ruge and Stuben's classic AMG.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Chensong Zhang

Date

05/09/2010

Modified by Xiaozhe Hu on 01/23/2011: add AMLI cycle. Modified by Xiaozhe Hu on 04/24/2013: aggressive coarsening. Modified by Chensong Zhang on 09/23/2014: check coarse spaces. Modified by Chensong Zhang on 08/28/2022: min↵_cdof from SHORT to INT.

Definition at line 52 of file [PreAMGSetupRS.c](#).

9.146 PreAMGSetupRS.c

[Go to the documentation of this file.](#)

```
00001
00021 #include <time.h>
00022
00023 #ifdef _OPENMP
00024 #include <omp.h>
00025 #endif
00026
00027 #include "fasp.h"
00028 #include "fasp_funcs.h"
00029
00030 /*-----*/
00031 /*--      Public Functions      --*/
00032 /*-----*/
00033
00052 SHORT fasp_amg_setup_rs (AMG_data  *mgl,
00053                          AMG_param *param)
00054 {
00055     const SHORT prtlvl   = param->print_level;
00056     const SHORT cycle_type = param->cycle_type;
00057     const SHORT csolver   = param->coarse_solver;
00058     const INT   min_cdof  = MAX(param->coarse_dof, MIN_CDOF);
```



```

00059     const INT    m          = mgl[0].A.row;
00060
00061     // local variables
00062     SHORT        status = FASP_SUCCESS;
00063     INT          lvl = 0, max_lvls = param->max_levels;
00064     REAL         setup_start, setup_end;
00065     ILU_param    iluparam;
00066     SWZ_param    swzparam;
00067     iCSRmat      Scouple; // strong n-couplings
00068
00069     // level info (fine: 0; coarse: 1)
00070     ivector      vertices = fasp_ivec_create(m);
00071
00072     // Output some info for debugging
00073     if ( prtlvl > PRINT_NONE ) printf("\nSetting up Classical AMG ...\n");
00074
00075     #if DEBUG_MODE > 0
00076     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00077     printf("### DEBUG: n = %d, nnz = %d\n", mgl[0].A.row, mgl[0].A.nnz);
00078     #endif
00079
00080     fasp_gettime(&setup_start);
00081
00082     // Make sure classical AMG will not call fasp_blas_dcsr_mvx_agg!
00083     param->tentative_smooth = 1.0;
00084
00085     // If user want to use aggressive coarsening but did not specify number of
00086     // levels use aggressive coarsening, make sure apply aggressive coarsening
00087     // on the finest level only !!!
00088     if ( param->coarsening_type == COARSE_AC ) {
00089         param->aggressive_level = MAX(param->aggressive_level, 1);
00090     }
00091
00092     // Initialize AMLI coefficients
00093     if ( cycle_type == AMLI_CYCLE ) {
00094         const INT amlideg = param->amli_degree;
00095         param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1, sizeof(REAL));
00096         fasp_amg_amli_coef(2.0, 0.5, amlideg, param->amli_coef);
00097     }
00098
00099     // Initialize ILU parameters
00100     mgl->ILU_levels = param->ILU_levels;
00101     if ( param->ILU_levels > 0 ) {
00102         iluparam.print_level = param->print_level;
00103         iluparam.ILU_lfil    = param->ILU_lfil;
00104         iluparam.ILU_droptol = param->ILU_droptol;
00105         iluparam.ILU_relax   = param->ILU_relax;
00106         iluparam.ILU_type    = param->ILU_type;
00107     }
00108
00109     // Initialize Schwarz parameters
00110     mgl->SWZ_levels = param->SWZ_levels;
00111     if ( param->SWZ_levels > 0 ) {
00112         swzparam.SWZ_mmsize = param->SWZ_mmsize;
00113         swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
00114         swzparam.SWZ_type   = param->SWZ_type;
00115         swzparam.SWZ_blksolver = param->SWZ_blksolver;
00116     }
00117
00118     #if DIAGONAL_PREF
00119     // Reorder each row to keep the diagonal entries appear first !!!
00120     fasp_dcsr_diagpref(&mgl[0].A);
00121     #endif
00122
00123     // Main AMG setup loop
00124     while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_lvls-1) ) {
00125
00126     #if DEBUG_MODE > 1
00127         printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
00128             lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);
00129     #endif
00130
00131     /*-- Setup ILU decomposition if needed --*/
00132     if ( lvl < param->ILU_levels ) {
00133         status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00134         if ( status < 0 ) {
00135             if ( prtlvl > PRINT_MIN ) {
00136                 printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00137                 printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00138             }
00139             param->ILU_levels = lvl;

```

```

00140     }
00141 }
00142
00143 /*-- Setup Schwarz smoother if needed --*/
00144 if ( lvl < param->SWZ_levels ) {
00145     mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
00146     fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
00147     status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
00148     if ( status < 0 ) {
00149         if ( prtlvl > PRINT_MIN ) {
00150             printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
00151             printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
00152         }
00153         param->SWZ_levels = lvl;
00154     }
00155 }
00156
00157 /*-- Coarsening and form the structure of interpolation --*/
00158 status = fasp_amg_coarsening_rs(&mgl[lvl].A, &vertices, &mgl[lvl].P,
00159                                &Scouple, param);
00160
00161 // Check 1: Did coarsening step succeeded?
00162 if ( status < 0 ) {
00163     /*-- Clean up Scouple generated in coarsening --*/
00164     fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
00165     fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
00166
00167     // When error happens, stop at the current multigrid level!
00168     if ( prtlvl > PRINT_MIN ) {
00169         printf("### WARNING: Could not find any C-variables!\n");
00170         printf("### WARNING: Stop coarsening on level=%d!\n", lvl);
00171     }
00172     status = FASP_SUCCESS; break;
00173 }
00174
00175 // Check 2: Is coarse sparse too small?
00176 if ( mgl[lvl].P.col < MIN_CDOF ) {
00177     /*-- Clean up Scouple generated in coarsening --*/
00178     fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
00179     fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
00180     break;
00181 }
00182
00183 // Check 3: Does this coarsening step too aggressive?
00184 if ( mgl[lvl].P.row > mgl[lvl].P.col * 10.0 ) {
00185     if ( prtlvl > PRINT_MIN ) {
00186         printf("### WARNING: Coarsening might be too aggressive!\n");
00187         printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00188             mgl[lvl].P.row, mgl[lvl].P.col);
00189     }
00190
00191     /*-- Clean up Scouple generated in coarsening --*/
00192     fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
00193     fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
00194     break;
00195 }
00196
00197 /*-- Perform aggressive coarsening only up to the specified level --*/
00198 if ( mgl[lvl].P.col*1.5 > mgl[lvl].A.row ) param->coarsening_type = COARSE_RS;
00199 if ( lvl == param->aggressive_level ) param->coarsening_type = COARSE_RS;
00200
00201 /*-- Store the C/F marker --*/
00202 {
00203     INT size = mgl[lvl].A.row;
00204     mgl[lvl].cfmark = fasp_ivec_create(size);
00205     memcpy(mgl[lvl].cfmark.val, vertices.val, size*sizeof(INT));
00206 }
00207
00208 /*-- Form interpolation --*/
00209 fasp_amg_interp(&mgl[lvl].A, &vertices, &mgl[lvl].P, &Scouple, param);
00210
00211 /*-- Form coarse level matrix: two RAP routines available! --*/
00212 fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00213
00214 fasp_blas_dcsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);
00215
00216 /*-- Clean up Scouple generated in coarsening --*/
00217 fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
00218 fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
00219
00220 ++lvl;

```

```

00221
00222 #if DIAGONAL_PREF
00223     // reorder each row to make diagonal appear first
00224     fasp_dcsr_diagpref(&mgl[lvl].A);
00225 #endif
00226
00227     // Check 4: Is the coarse matrix too dense?
00228     if ( mgl[lvl].A.nnz / mgl[lvl].A.row > mgl[lvl].A.col * 0.2 ) {
00229         if ( prtlvl > PRINT_MIN ) {
00230             printf("### WARNING: Coarse matrix is too dense!\n");
00231             printf("### WARNING: m = n = %d, nnz = %d!\n",
00232                 mgl[lvl].A.col, mgl[lvl].A.nnz);
00233         }
00234         break;
00235     }
00236 } // end of the main while loop
00237
00238 } // end of the main while loop
00239
00240 // Setup coarse level systems for direct solvers
00241 switch (csolver) {
00242
00243 #if WITH_MUMPS
00244     case SOLVER_MUMPS: {
00245         // Setup MUMPS direct solver on the coarsest level
00246         mgl[lvl].mumps.job = 1;
00247         fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00248         break;
00249     }
00250 #endif
00251
00252 #if WITH_UMFPACK
00253     case SOLVER_UMFPACK: {
00254         // Need to sort the matrix A for UMFPACK to work
00255         dCSRmat Ac_tran;
00256         Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
00257         fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
00258         // It is equivalent to do transpose and then sort
00259         // fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
00260         // fasp_dcsr_sort(&Ac_tran);
00261         fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
00262         fasp_dcsr_free(&Ac_tran);
00263         mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
00264         break;
00265     }
00266 #endif
00267
00268 #if WITH_PARDISO
00269     case SOLVER_PARDISO: {
00270         fasp_dcsr_sort(&mgl[lvl].A);
00271         fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
00272         break;
00273     }
00274 #endif
00275
00276     default:
00277         // Do nothing!
00278         break;
00279 }
00280
00281 // setup total level number and current level
00282 mgl[0].num_levels = max_lvls = lvl+1;
00283 mgl[0].w = fasp_dvec_create(m);
00284
00285 for ( lvl = 1; lvl < max_lvls; ++lvl ) {
00286     const INT mm = mgl[lvl].A.row;
00287
00288     mgl[lvl].num_levels = max_lvls;
00289     mgl[lvl].b = fasp_dvec_create(mm);
00290     mgl[lvl].x = fasp_dvec_create(mm);
00291
00292     mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
00293     mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00294     mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!
00295
00296     // allocate work arrays for the solve phase
00297     if ( cycle_type == NL_AMLI_CYCLE )
00298         mgl[lvl].w = fasp_dvec_create(3*mm);
00299     else
00300         mgl[lvl].w = fasp_dvec_create(2*mm);
00301 }

```

```

00302
00303     fasp_ivec_free(&vertices);
00304
00305 #if MULTI_COLOR_ORDER
00306     INT Colors,rowmax;
00307 #ifdef _OPENMP
00308     int threads = fasp_get_num_threads();
00309     if (threads > max_lvls-1 ) threads = max_lvls-1;
00310 #pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
00311 #endif
00312     for (lvl=0; lvl<max_lvls-1; lvl++){
00313
00314 #if 1
00315         dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
00316 #else
00317         dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
00318 #endif
00319         if ( prtlvl > 1 )
00320             printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %1e.\n",
00321                 lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
00322                 mgl[lvl].A.color, mgl[lvl].GS_Theta);
00323     }
00324 #endif
00325
00326     if ( prtlvl > PRINT_NONE ) {
00327         fasp_gettime(&setup_end);
00328         fasp_amgcomplexity(mgl, prtlvl);
00329         fasp_cputime("Classical AMG setup", setup_end - setup_start);
00330     }
00331
00332 #if DEBUG_MODE > 0
00333     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00334 #endif
00335
00336     return status;
00337 }
00338
00339 /*-----*/
00340 /*--          End of File          --*/
00341 /*-----*/
00342

```

9.147 PreAMGSetupSA.c File Reference

Smoothed aggregation AMG: SETUP phase.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreAMGAggregation.inl"
#include "PreAMGAggregationCSR.inl"

```

Functions

- [SHORT fasp_amg_setup_sa](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
Set up phase of smoothed aggregation AMG.

9.147.1 Detailed Description

Smoothed aggregation AMG: SETUP phase.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlalLUSetupCSR.c](#), [BlaSchwarzSetup.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), and [PreMGRecurAMLI.c](#)

Setup A, P, PT and levels using the unsmoothed aggregation algorithm

Reference: P. Vanek, J. Madel and M. Brezina Algebraic Multigrid on Unstructured Meshes, 1994
 Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGSetupSA.c](#).

9.147.2 Function Documentation

9.147.2.1 fasp_amg_setup_sa()

```
SHORT fasp_amg_setup_sa (
    AMG_data * mgl,
    AMG_param * param )
```

Set up phase of smoothed aggregation AMG.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xiaozhe Hu

Date

09/29/2009

Modified by Xiaozhe Hu on 01/23/2011: add AMLI cycle. Modified by Chensong Zhang on 05/10/2013: adjust the structure.

Definition at line 63 of file [PreAMGSetupSA.c](#).

9.148 PreAMGSetupSA.c

[Go to the documentation of this file.](#)

```
00001
00022 #include <math.h>
00023 #include <time.h>
00024
00025 #ifdef _OPENMP
00026 #include <omp.h>
00027 #endif
00028
00029 #include "fasp.h"
00030 #include "fasp_funcs.h"
00031
00032 /*-----*/
00033 /*--  Declare Private Functions  --*/
00034 /*-----*/
00035
00036 #include "PreAMGAggregation.inl"
00037 #include "PreAMGAggregationCSR.inl"
00038
00039 static SHORT amg_setup_smoothP_smoothR (AMG_data *, AMG_param *);
```

```

00040 static SHORT amg_setup_smoothP_unsmoothR (AMG_data *, AMG_param *);
00041 static void smooth_agg (dCSRmat *, dCSRmat *, dCSRmat *, AMG_param *, dCSRmat *);
00042
00043 /*-----*/
00044 /*--      Public Functions      --*/
00045 /*-----*/
00046
00063 SHORT fasp_amg_setup_sa (AMG_data *mgl,
00064                          AMG_param *param)
00065 {
00066     const SHORT prtlvl      = param->print_level;
00067     const SHORT smoothR     = param->smooth_restriction;
00068     SHORT status            = FASP_SUCCESS;
00069
00070     // Output some info for debugging
00071     if ( prtlvl > PRINT_NONE ) printf("\nSetting up SA AMG ...\n");
00072
00073     #if DEBUG_MODE > 0
00074         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00075         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
00076               mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00077     #endif
00078
00079     if ( smoothR ) { // Default: smoothed P, smoothed R
00080         status = amg_setup_smoothP_smoothR(mgl, param);
00081     }
00082     else { // smoothed P, unsmoothed R
00083         status = amg_setup_smoothP_unsmoothR(mgl, param);
00084     }
00085
00086     #if DEBUG_MODE > 0
00087         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00088     #endif
00089
00090     return status;
00091 }
00092
00093 /*-----*/
00094 /*--      Private Functions      --*/
00095 /*-----*/
00096
00115 static void smooth_agg (dCSRmat *A,
00116                        dCSRmat *tentp,
00117                        dCSRmat *P,
00118                        AMG_param *param,
00119                        dCSRmat *N)
00120 {
00121     const SHORT filter = param->smooth_filter;
00122     const INT row = A->row, col = A->col;
00123     const REAL smooth_factor = param->tentative_smooth;
00124
00125     dCSRmat S;
00126     dvector diag; // diagonal entries
00127
00128     REAL row_sum_A, row_sum_N;
00129     INT i, j;
00130
00131     /* Step 1. Form smoother */
00132
00133     /* Without filter: Using A for damped Jacobian smoother */
00134     if ( filter != ON ) {
00135
00136         // copy structure from A
00137         S = fasp_dcsr_create(row, col, A->IA[row]);
00138
00139         #ifdef _OPENMP
00140         #pragma omp parallel for if(row>OPENMP_HOLDS)
00141         #endif
00142         for ( i=0; i<=row; ++i ) S.IA[i] = A->IA[i];
00143         for ( i=0; i<S.IA[S.row]; ++i ) S.JA[i] = A->JA[i];
00144
00145         fasp_dcsr_getdiag(0, A, &diag); // get the diagonal entries of A
00146
00147         // check the diagonal entries.
00148         // if it is too small, use Richardson smoother for the corresponding row
00149         #ifdef _OPENMP
00150         #pragma omp parallel for if(row>OPENMP_HOLDS)
00151         #endif
00152         for ( i=0; i<row; ++i ) {
00153             if ( ABS(diag.val[i]) < 1e-6 ) diag.val[i] = 1.0;
00154         }
00155     }

```

```

00155
00156 #ifdef _OPENMP
00157 #pragma omp parallel for if(row>OPENMP_HOLDS) private(j)
00158 #endif
00159     for (i=0; i<row; ++i) {
00160         for (j=S.IA[i]; j<S.IA[i+1]; ++j) {
00161             if (S.JA[j] == i) {
00162                 S.val[j] = 1 - smooth_factor * A->val[j] / diag.val[i];
00163             }
00164             else {
00165                 S.val[j] = - smooth_factor * A->val[j] / diag.val[i];
00166             }
00167         }
00168     }
00169 }
00170
00171 /* Using filtered A for damped Jacobian smoother */
00172 else {
00173     /* Form filtered A and store in N */
00174 #ifdef _OPENMP
00175 #pragma omp parallel for private(j, row_sum_A, row_sum_N) if (row>OPENMP_HOLDS)
00176 #endif
00177     for (i=0; i<row; ++i) {
00178         for (row_sum_A = 0.0, j=A->IA[i]; j<A->IA[i+1]; ++j) {
00179             if (A->JA[j] != i) row_sum_A += A->val[j];
00180         }
00181
00182         for (row_sum_N = 0.0, j=N->IA[i]; j<N->IA[i+1]; ++j) {
00183             if (N->JA[j] != i) row_sum_N += N->val[j];
00184         }
00185
00186         for (j=N->IA[i]; j<N->IA[i+1]; ++j) {
00187             if (N->JA[j] == i) {
00188                 // The original paper has a wrong sign!!! --Chensong
00189                 N->val[j] += row_sum_A - row_sum_N;
00190             }
00191         }
00192     }
00193
00194     // copy structure from N (filtered A)
00195     S = fasp_dcsr_create(row, col, N->IA[row]);
00196
00197 #ifdef _OPENMP
00198 #pragma omp parallel for if(row>OPENMP_HOLDS)
00199 #endif
00200     for (i=0; i<=row; ++i) S.IA[i] = N->IA[i];
00201
00202     for (i=0; i<S.IA[S.row]; ++i) S.JA[i] = N->JA[i];
00203
00204     fasp_dcsr_getdiag(0, N, &diag); // get the diagonal entries of N (filtered A)
00205
00206     // check the diagonal entries.
00207     // if it is too small, use Richardson smoother for the corresponding row
00208 #ifdef _OPENMP
00209 #pragma omp parallel for if(row>OPENMP_HOLDS)
00210 #endif
00211     for (i=0; i<row; ++i) {
00212         if (ABS(diag.val[i]) < 1e-6) diag.val[i] = 1.0;
00213     }
00214
00215 #ifdef _OPENMP
00216 #pragma omp parallel for if(row>OPENMP_HOLDS) private(i, j)
00217 #endif
00218     for (i=0; i<row; ++i) {
00219         for (j=S.IA[i]; j<S.IA[i+1]; ++j) {
00220             if (S.JA[j] == i) {
00221                 S.val[j] = 1 - smooth_factor * N->val[j] / diag.val[i];
00222             }
00223             else {
00224                 S.val[j] = - smooth_factor * N->val[j] / diag.val[i];
00225             }
00226         }
00227     }
00228 }
00229 }
00230
00231 fasp_dvec_free(&diag);
00232
00233 /* Step 2. Smooth the tentative prolongation P = S*tenp */
00234 fasp_blas_dcsr_mxm(&S, tenp, P); // Note: think twice about this.
00235 P->nnz = P->IA[P->row];

```

```

00236     fasp_dcsr_free(&S);
00237 }
00238
00254 static SHORT amg_setup_smoothP_smoothR (AMG_data *mgl,
00255                                         AMG_param *param)
00256 {
00257     const SHORT prtlvl      = param->print_level;
00258     const SHORT cycle_type  = param->cycle_type;
00259     const SHORT csolver     = param->coarse_solver;
00260     const SHORT min_cdof    = MAX(param->coarse_dof,50);
00261     const INT    m          = mgl[0].A.row;
00262
00263     // local variables
00264     SHORT    max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
00265     INT      i, j;
00266     REAL     setup_start, setup_end;
00267     ILU_param iluparam;
00268     SWZ_param swzparam;
00269
00270 #if DEBUG_MODE > 0
00271     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00272 #endif
00273
00274     fasp_gettime(&setup_start);
00275
00276     // level info (fine: 0; coarse: 1)
00277     ivector *vertices = (ivector *)fasp_mem_calloc(max_levels, sizeof(ivector));
00278
00279     // each elvel stores the information of the number of aggregations
00280     INT *num_aggs = (INT *)fasp_mem_calloc(max_levels, sizeof(INT));
00281
00282     // each level stores the information of the strongly coupled neighbourhood
00283     dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00284
00285     // each level stores the information of the tentative prolongations
00286     dCSRmat *tentp = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00287
00288     // Initialize level information
00289     for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;
00290
00291     mgl[0].near_kernel_dim = 1;
00292     mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));
00293
00294     for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
00295         mgl[0].near_kernel_basis[i] = (REAL *)fasp_mem_calloc(m, sizeof(REAL));
00296         for ( j = 0; j < m; ++j ) mgl[0].near_kernel_basis[i][j] = 1.0;
00297     }
00298
00299     // Initialize ILU parameters
00300     mgl->ILU_levels = param->ILU_levels;
00301     if ( param->ILU_levels > 0 ) {
00302         iluparam.print_level = param->print_level;
00303         iluparam.ILU_lfil    = param->ILU_lfil;
00304         iluparam.ILU_droptol = param->ILU_droptol;
00305         iluparam.ILU_relax   = param->ILU_relax;
00306         iluparam.ILU_type    = param->ILU_type;
00307     }
00308
00309     // Initialize Schwarz parameters
00310     mgl->SWZ_levels = param->SWZ_levels;
00311     if ( param->SWZ_levels > 0 ) {
00312         swzparam.SWZ_mmsize = param->SWZ_mmsize;
00313         swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
00314         swzparam.SWZ_type   = param->SWZ_type;
00315         swzparam.SWZ_blksolver = param->SWZ_blksolver;
00316     }
00317
00318     // Initialize AMLI coefficients
00319     if ( cycle_type == AMLI_CYCLE ) {
00320         const INT amlideg = param->amli_degree;
00321         param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1, sizeof(REAL));
00322         REAL lambda_max = 2.0, lambda_min = lambda_max/4;
00323         fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
00324     }
00325
00326 #if DIAGONAL_PREF
00327     fasp_dcsr_diagpref(&mgl[0].A); // reorder each row to make diagonal appear first
00328 #endif
00329
00330     /*-----*/
00331     /*--- checking aggregation ---*/

```



```

00332      /*-----*/
00333      if ( param->aggregation_type == PAIRWISE )
00334          param->pair_number = MIN(param->pair_number, max_levels);
00335
00336      // Main AMG setup loop
00337      while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_levels-1) ) {
00338
00339      #if DEBUG_MODE > 2
00340          printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
00341              lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);
00342      #endif
00343
00344      /*-- setup ILU decomposition if necessary */
00345      if ( lvl < param->ILU_levels ) {
00346          status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00347          if ( status < 0 ) {
00348              if ( prtlvl > PRINT_MIN ) {
00349                  printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00350                  printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00351              }
00352              param->ILU_levels = lvl;
00353          }
00354      }
00355
00356      /* -- setup Schwarz smoother if necessary */
00357      if ( lvl < param->SWZ_levels ) {
00358          mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
00359          fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
00360          status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
00361          if ( status < 0 ) {
00362              if ( prtlvl > PRINT_MIN ) {
00363                  printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
00364                  printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
00365              }
00366              param->SWZ_levels = lvl;
00367          }
00368      }
00369
00370      /*-- Aggregation --*/
00371      status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl+1,
00372          &Neighbor[lvl], &num_aggs[lvl]);
00373
00374      // Check 1: Did coarsening step succeed?
00375      if ( status < 0 ) {
00376          // When error happens, stop at the current multigrid level!
00377          if ( prtlvl > PRINT_MIN ) {
00378              printf("### WARNING: Forming aggregates on level-%d failed!\n", lvl);
00379          }
00380          status = FASP_SUCCESS;
00381          fasp_ivec_free(&vertices[lvl]);
00382          fasp_dcsr_free(&Neighbor[lvl]);
00383          break;
00384      }
00385
00386      /* -- Form Tentative prolongation --*/
00387      form_tentative_p(&vertices[lvl], &tentp[lvl], mgl[0].near_kernel_basis,
00388          num_aggs[lvl]);
00389
00390      /* -- Form smoothed prolongation -- */
00391      smooth_agg(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);
00392
00393      // Check 2: Is coarse sparse too small?
00394      if ( mgl[lvl].P.col < MIN_CDof ) {
00395          fasp_ivec_free(&vertices[lvl]);
00396          fasp_dcsr_free(&Neighbor[lvl]);
00397          fasp_dcsr_free(&tentp[lvl]);
00398          break;
00399      }
00400
00401      // Check 3: Does this coarsening step too aggressive?
00402      if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
00403          if ( prtlvl > PRINT_MIN ) {
00404              printf("### WARNING: Coarsening might be too aggressive!\n");
00405              printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00406                  mgl[lvl].P.row, mgl[lvl].P.col);
00407          }
00408          fasp_ivec_free(&vertices[lvl]);
00409          fasp_dcsr_free(&Neighbor[lvl]);
00410          fasp_dcsr_free(&tentp[lvl]);
00411          break;
00412      }

```

```

00413
00414     /*--- Form restriction ---*/
00415     fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00416
00417     /*--- Form coarse level stiffness matrix ---*/
00418     fasp_blas_dcsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);
00419
00420     fasp_dcsr_free(&Neighbor[lvl]);
00421     fasp_dcsr_free(&tentp[lvl]);
00422     fasp_ivec_free(&vertices[lvl]);
00423
00424     ++lvl;
00425
00426 #if DIAGONAL_PREF
00427     // reorder each row to make diagonal appear first
00428     fasp_dcsr_diagpref(&mgl[lvl].A);
00429 #endif
00430
00431     // Check 4: Is this coarsening ratio too small?
00432     if ( (REAL)mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
00433         if ( prtlvl > PRINT_MIN ) {
00434             printf("### WARNING: Coarsening rate is too small!\n");
00435             printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00436                 mgl[lvl].P.row, mgl[lvl].P.col);
00437         }
00438
00439         break;
00440     }
00441
00442 } // end of the main while loop
00443
00444 // Setup coarse level systems for direct solvers
00445 switch (csolver) {
00446
00447 #if WITH_MUMPS
00448     case SOLVER_MUMPS: {
00449         // Setup MUMPS direct solver on the coarsest level
00450         mgl[lvl].mumps.job = 1;
00451         fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00452         break;
00453     }
00454 #endif
00455
00456 #if WITH_UMFPACK
00457     case SOLVER_UMFPACK: {
00458         // Need to sort the matrix A for UMFPACK to work
00459         dCSRmat Ac_tran;
00460         Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
00461         fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
00462         // It is equivalent to do transpose and then sort
00463         // fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
00464         // fasp_dcsr_sort(&Ac_tran);
00465         fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
00466         fasp_dcsr_free(&Ac_tran);
00467         mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
00468         break;
00469     }
00470 #endif
00471
00472 #if WITH_PARDISO
00473     case SOLVER_PARDISO: {
00474         fasp_dcsr_sort(&mgl[lvl].A);
00475         fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
00476         break;
00477     }
00478 #endif
00479
00480     default:
00481         // Do nothing!
00482         break;
00483 }
00484
00485 // setup total level number and current level
00486 mgl[0].num_levels = max_levels = lvl+1;
00487 mgl[0].w          = fasp_dvec_create(m);
00488
00489 for ( lvl = 1; lvl < max_levels; ++lvl ) {
00490     INT mm = mgl[lvl].A.row;
00491     mgl[lvl].num_levels = max_levels;
00492     mgl[lvl].b          = fasp_dvec_create(mm);
00493     mgl[lvl].x          = fasp_dvec_create(mm);

```

```

00494
00495     mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
00496     mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00497     mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!
00498
00499     if ( cycle_type == NL_AMLI_CYCLE )
00500         mgl[lvl].w = fasp_dvec_create(3*mm);
00501     else
00502         mgl[lvl].w = fasp_dvec_create(2*mm);
00503 }
00504
00505 #if MULTI_COLOR_ORDER
00506     INT Colors,rowmax;
00507 #ifdef _OPENMP
00508     int threads = fasp_get_num_threads();
00509     if (threads > max_levels-1 ) threads = max_levels-1;
00510 #pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
00511 #endif
00512     for (lvl=0; lvl<max_levels-1; lvl++){
00513
00514 #if 1
00515         dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
00516 #else
00517         dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
00518 #endif
00519         if ( prtlvl > 1 )
00520             printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %1e.\n",
00521                 lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
00522                 mgl[lvl].A.color, mgl[lvl].GS_Theta);
00523     }
00524 #endif
00525
00526     if ( prtlvl > PRINT_NONE ) {
00527         fasp_gettime(&setup_end);
00528         fasp_amgcomplexity(mgl,prtlvl);
00529         fasp_cputime("Smoothed aggregation setup", setup_end - setup_start);
00530     }
00531
00532     fasp_mem_free(vertices); vertices = NULL;
00533     fasp_mem_free(num_aggs); num_aggs = NULL;
00534     fasp_mem_free(Neighbor); Neighbor = NULL;
00535     fasp_mem_free(tentp); tentp = NULL;
00536
00537 #if DEBUG_MODE > 0
00538     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00539 #endif
00540
00541     return status;
00542 }
00543
00544 static SHORT amg_setup_smoothP_unsmoothR (AMG_data *mgl,
00545                                           AMG_param *param)
00546 {
00547     const SHORT prtlvl = param->print_level;
00548     const SHORT cycle_type = param->cycle_type;
00549     const SHORT csolver = param->coarse_solver;
00550     const SHORT min_cdof = MAX(param->coarse_dof,50);
00551     const INT m = mgl[0].A.row;
00552
00553     // local variables
00554     SHORT max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
00555     INT i, j;
00556     REAL setup_start, setup_end;
00557     ILU_param iluparam;
00558     SWZ_param swzparam;
00559
00560 #if DEBUG_MODE > 0
00561     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00562 #endif
00563
00564     fasp_gettime(&setup_start);
00565
00566     // level info (fine: 0; coarse: 1)
00567     ivector *vertices = (ivector *)fasp_mem_calloc(max_levels,sizeof(ivector));
00568
00569     // each level stores the information of the number of aggregations
00570     INT *num_aggs = (INT *)fasp_mem_calloc(max_levels,sizeof(INT));
00571
00572     // each level stores the information of the strongly coupled neighbourhood
00573     dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));
00574
00575

```

```

00590 // each level stores the information of the tentative prolongations
00591 dCSRmat *tentp = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00592 dCSRmat *tentr = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00593
00594 for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;
00595
00596 mgl[0].near_kernel_dim = 1;
00597
00598 mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));
00599
00600 for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
00601     mgl[0].near_kernel_basis[i] = (REAL *)fasp_mem_calloc(m, sizeof(REAL));
00602     for ( j = 0; j < m; ++j ) mgl[0].near_kernel_basis[i][j] = 1.0;
00603 }
00604
00605 // Initialize ILU parameters
00606 if ( param->ILU_levels > 0 ) {
00607     iluparam.print_level = param->print_level;
00608     iluparam.ILU_lfil = param->ILU_lfil;
00609     iluparam.ILU_droptol = param->ILU_droptol;
00610     iluparam.ILU_relax = param->ILU_relax;
00611     iluparam.ILU_type = param->ILU_type;
00612 }
00613
00614 // Initialize Schwarz parameters
00615 mgl->SWZ_levels = param->SWZ_levels;
00616 if ( param->SWZ_levels > 0 ) {
00617     swzparam.SWZ_mmsize = param->SWZ_mmsize;
00618     swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
00619     swzparam.SWZ_type = param->SWZ_type;
00620     swzparam.SWZ_blksolver = param->SWZ_blksolver;
00621 }
00622
00623 // Initialize AMLI coefficients
00624 if ( cycle_type == AMLI_CYCLE ) {
00625     const INT amlideg = param->amli_degree;
00626     param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1, sizeof(REAL));
00627     REAL lambda_max = 2.0, lambda_min = lambda_max/4;
00628     fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
00629 }
00630
00631 // Main AMG setup loop
00632 while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_levels-1) ) {
00633
00634     /*-- setup ILU decomposition if necessary */
00635     if ( lvl < param->ILU_levels ) {
00636         status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00637         if ( status < 0 ) {
00638             if ( prtlvl > PRINT_MIN ) {
00639                 printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00640                 printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00641             }
00642             param->ILU_levels = lvl;
00643         }
00644     }
00645
00646     /* -- setup Schwarz smoother if necessary */
00647     if ( lvl < param->SWZ_levels ) {
00648         mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
00649         fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
00650         status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
00651         if ( status < 0 ) {
00652             if ( prtlvl > PRINT_MIN ) {
00653                 printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
00654                 printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
00655             }
00656             param->SWZ_levels = lvl;
00657         }
00658     }
00659
00660     /*-- Aggregation --*/
00661     status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl+1,
00662                             &Neighbor[lvl], &num_aggs[lvl]);
00663
00664     // Check 1: Did coarsening step succeeded?
00665     if ( status < 0 ) {
00666         // When error happens, stop at the current multigrid level!
00667         if ( prtlvl > PRINT_MIN ) {
00668             printf("### WARNING: Stop coarsening on level=%d!\n", lvl);
00669         }
00670         status = FASP_SUCCESS; break;
00671     }

```

```

00671     }
00672
00673     /* -- Form Tentative prolongation --*/
00674     form_tentative_p(&vertices[lvl], &tentp[lvl], mgl[0].near_kernel_basis,
00675                     num_aggs[lvl]);
00676
00677     /* -- Form smoothed prolongation -- */
00678     smooth_agg(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);
00679
00680     // Check 2: Is coarse sparse too small?
00681     if ( mgl[lvl].P.col < MIN_CDOF ) break;
00682
00683     // Check 3: Does this coarsening step too aggressive?
00684     if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
00685         if ( prtlvl > PRINT_MIN ) {
00686             printf("### WARNING: Coarsening might be too aggressive!\n");
00687             printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00688                   mgl[lvl].P.row, mgl[lvl].P.col);
00689         }
00690         break;
00691     }
00692
00693     // Check 4: Is this coarsening ratio too small?
00694     if ( (REAL)mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
00695         if ( prtlvl > PRINT_MIN ) {
00696             printf("### WARNING: Coarsening rate is too small!\n");
00697             printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00698                   mgl[lvl].P.row, mgl[lvl].P.col);
00699         }
00700         break;
00701     }
00702
00703     /*-- Form restriction --*/
00704     fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00705     fasp_dcsr_trans(&tentp[lvl], &tentr[lvl]);
00706
00707     /*-- Form coarse level stiffness matrix --*/
00708     fasp_blas_dcsr_rap_agg(&tentr[lvl], &mgl[lvl].A, &tentp[lvl], &mgl[lvl+1].A);
00709
00710     fasp_dcsr_free(&Neighbor[lvl]);
00711     fasp_dcsr_free(&tentp[lvl]);
00712     fasp_ivec_free(&vertices[lvl]);
00713
00714     ++lvl;
00715 }
00716
00717 // Setup coarse level systems for direct solvers
00718 switch (csolver) {
00719 #if WITH_MUMPS
00720     case SOLVER_MUMPS: {
00721         // Setup MUMPS direct solver on the coarsest level
00722         mgl[lvl].mumps.job = 1;
00723         fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00724         break;
00725     }
00726 #endif
00727 #if WITH_UMFPACK
00728     case SOLVER_UMFPACK: {
00729         // Need to sort the matrix A for UMFPACK to work
00730         dCSRmat Ac_tran;
00731         Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
00732         fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
00733         // It is equivalent to do transpose and then sort
00734         fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
00735         fasp_dcsr_sort(&Ac_tran);
00736         fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
00737         fasp_dcsr_free(&Ac_tran);
00738         mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
00739         break;
00740     }
00741 #endif
00742 #if WITH_PARDISO
00743     case SOLVER_PARDISO: {
00744         fasp_dcsr_sort(&mgl[lvl].A);
00745         fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
00746         break;
00747     }
00748 #endif
00749 }
00750 #endif

```

```

00752
00753     default:
00754         // Do nothing!
00755         break;
00756     }
00757
00758     // setup total level number and current level
00759     mgl[0].num_levels = max_levels = lvl+1;
00760     mgl[0].w         = fasp_dvec_create(m);
00761
00762     for ( lvl = 1; lvl < max_levels; ++lvl ) {
00763         INT mm = mgl[lvl].A.row;
00764         mgl[lvl].num_levels = max_levels;
00765         mgl[lvl].b         = fasp_dvec_create(mm);
00766         mgl[lvl].x         = fasp_dvec_create(mm);
00767
00768         mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
00769         mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00770         mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!
00771
00772         if ( cycle_type == NL_AMLI_CYCLE )
00773             mgl[lvl].w = fasp_dvec_create(3*mm);
00774         else
00775             mgl[lvl].w = fasp_dvec_create(2*mm);
00776     }
00777
00778     #if MULTI_COLOR_ORDER
00779     INT Colors,rowmax;
00780     #ifdef _OPENMP
00781     int threads = fasp_get_num_threads();
00782     if (threads > max_levels-1 ) threads = max_levels-1;
00783     #pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
00784     #endif
00785     for (lvl=0; lvl<max_levels-1; lvl++){
00786
00787     #if 1
00788         dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
00789     #else
00790         dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
00791     #endif
00792     if ( prtlvl > 1 )
00793         printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %1e.\n",
00794             lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
00795             mgl[lvl].A.color, mgl[lvl].GS_Theta);
00796     }
00797     #endif
00798
00799     if ( prtlvl > PRINT_NONE ) {
00800         fasp_gettime(&setup_end);
00801         fasp_amgcomplexity(mgl,prtlvl);
00802         fasp_cputime("Smoothed aggregation 1/2 setup", setup_end - setup_start);
00803     }
00804
00805     fasp_mem_free(vertices); vertices = NULL;
00806     fasp_mem_free(num_aggs); num_aggs = NULL;
00807     fasp_mem_free(Neighbor); Neighbor = NULL;
00808     fasp_mem_free(tentp); tentp = NULL;
00809     fasp_mem_free(tentr); tendr = NULL;
00810
00811     #if DEBUG_MODE > 0
00812     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00813     #endif
00814
00815     return status;
00816 }
00817
00818 /*-----*/
00819 /*--          End of File          --*/
00820 /*-----*/

```

9.149 PreAMGSetupSABSR.c File Reference

Smoothed aggregation AMG: SETUP phase (for BSR matrices)

```

#include <math.h>
#include <time.h>
#include "fasp.h"

```

```
#include "fasp_functs.h"
#include "PreAMGAggregation.inl"
#include "PreAMGAggregationBSR.inl"
#include "PreAMGAggregationUA.inl"
```

Functions

- [SHORT fasp_amg_setup_sa_bsr](#) ([AMG_data_bsr](#) *mgl, [AMG_param](#) *param)
Set up phase of smoothed aggregation AMG (BSR format)

9.149.1 Detailed Description

Smoothed aggregation AMG: SETUP phase (for BSR matrices)

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaFormat.c](#), [BlaILUSetupBSR.c](#), [BlaSmallMat.c](#), [BlaSparseBLC.c](#), [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), [BlaSpmvBSR.c](#), and [BlaSpmvCSR.c](#)

Setup A, P, PT and levels using the unsmoothed aggregation algorithm

Reference: P. Vanek, J. Madel and M. Brezina Algebraic Multigrid on Unstructured Meshes, 1994
Copyright (C) 2014–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGSetupSABSR.c](#).

9.149.2 Function Documentation

9.149.2.1 fasp_amg_setup_sa_bsr()

```
INT fasp_amg_setup_sa_bsr (
    AMG\_data\_bsr * mgl,
    AMG\_param * param )
```

Set up phase of smoothed aggregation AMG (BSR format)

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data_bsr
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 61 of file [PreAMGSetupSABSR.c](#).

9.150 PreAMGSetupSABSR.c

[Go to the documentation of this file.](#)

```

00001
00022 #include <math.h>
00023 #include <time.h>
00024
00025 #ifdef _OPENMP
00026 #include <omp.h>
00027 #endif
00028
00029 #include "fasp.h"
00030 #include "fasp_functs.h"
00031
00032 /*-----*/
00033 /*--  Declare Private Functions  --*/
00034 /*-----*/
00035
00036 #include "PreAMGAggregation.inl"
00037 #include "PreAMGAggregationBSR.inl"
00038 #include "PreAMGAggregationUA.inl"
00039
00040 static SHORT amg_setup_smoothP_smoothR_bsr (AMG_data_bsr *, AMG_param *);
00041 static void smooth_agg_bsr (const dBSRmat *, dBSRmat *, dBSRmat *, const AMG_param *,
00042                             const dCSRmat *);
00043
00044 /*-----*/
00045 /*--      Public Functions      --*/
00046 /*-----*/
00047
00061 SHORT fasp_amg_setup_sa_bsr (AMG_data_bsr *mgl,
00062                             AMG_param *param)
00063 {
00064     #if DEBUG_MODE > 0
00065         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00066     #endif
00067
00068     SHORT status = amg_setup_smoothP_smoothR_bsr(mgl, param);
00069
00070     #if DEBUG_MODE > 0
00071         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00072     #endif
00073
00074     return status;
00075 }
00076
00077 /*-----*/
00078 /*--      Private Functions      --*/
00079 /*-----*/
00080
00096 static void smooth_agg_bsr (const dBSRmat *A,
00097                             dBSRmat *tentp,
00098                             dBSRmat *P,
00099                             const AMG_param *param,
00100                             const dCSRmat *N)
00101 {
00102     const INT row = A->ROW, col = A->COL, nnz = A->NNZ;
00103     const INT nb = A->nb, nb2 = nb*nb;
00104     const REAL smooth_factor = param->tentative_smooth;
00105
00106     // local variables
00107     dBSRmat S;
00108     dvector diagin; // diagonal block inv
00109
00110     INT i, j;
00111
00112     REAL *Id = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));
00113     REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));
00114
00115     fasp_smat_identity(Id, nb, nb2);
00116
00117     /* Step 1. Form smoother */
00118
00119     // copy structure from A
00120     S = fasp_dbsr_create(row, col, nnz, nb, 0);
00121
00122     for (i=0; i<=row; ++i) S.IA[i] = A->IA[i];
00123     for (i=0; i<nnz; ++i) S.JA[i] = A->JA[i];
00124
00125     diagin = fasp_dbsr_getdiagin(A);

```



```

00126
00127 // for S
00128 for (i=0; i<row; ++i) {
00129     for (j=S.IA[i]; j<S.IA[i+1]; ++j) {
00131         if (S.JA[j] == i) {
00132             fasp_blas_smat_mul(diaginv.val+(i*nb2), A->val+(j*nb2), temp, nb);
00133             fasp_blas_smat_add(Id, temp, nb, 1.0, (-1.0)*smooth_factor, S.val+(j*nb2));
00134         }
00135     }
00136     else {
00137         fasp_blas_smat_mul(diaginv.val+(i*nb2), A->val+(j*nb2), S.val+(j*nb2), nb);
00138         fasp_blas_smat_axm(S.val+(j*nb2), nb, (-1.0)*smooth_factor);
00139     }
00140 }
00141
00142 fasp_dvec_free(&diaginv);
00143 fasp_mem_free(Id); Id = NULL;
00144 fasp_mem_free(temp); temp = NULL;
00145
00146 /* Step 2. Smooth the tentative prolongation P = S*temp */
00147 fasp_blas_dbsr_mxm(&S, temp, P); // Note: think twice about this.
00148
00149 P->NNZ = P->IA[P->ROW];
00150 fasp_dbsr_free(&S);
00151 }
00152
00153 static SHORT amg_setup_smoothP_smoothR_bsr (AMG_data_bsr *mgl,
00154                                             AMG_param *param)
00155 {
00156     const SHORT CondType = 1; // Condensation method used for AMG
00157
00158     const SHORT prtlvl = param->print_level;
00159     const SHORT csolver = param->coarse_solver;
00160     const SHORT min_cdof = MAX(param->coarse_dof, 50);
00161     const INT m = mgl[0].A.ROW;
00162     const INT nb = mgl[0].A.nb;
00163
00164     ILU_param iluparam;
00165     SHORT max_levels=param->max_levels;
00166     SHORT i, lvl=0, status=FASP_SUCCESS;
00167     REAL setup_start, setup_end;
00168
00169     AMG_data *mgl_csr = fasp_amg_data_create(max_levels);
00170
00171     dCSRmat temp1, temp2;
00172
00173     #if DEBUG_MODE > 0
00174     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00175     printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
00176           mgl[0].A.ROW, mgl[0].A.COL, mgl[0].A.NNZ);
00177     #endif
00178
00179     fasp_gettime(&setup_start);
00180
00181     /*-----*/
00182     /*--local working array--*/
00183     /*-----*/
00184
00185     // level info (fine: 0; coarse: 1)
00186     ivector *vertices = (ivector *)fasp_mem_calloc(max_levels, sizeof(ivector));
00187
00188     // each level stores the information of the number of aggregations
00189     INT *num_aggs = (INT *)fasp_mem_calloc(max_levels, sizeof(INT));
00190
00191     // each level stores the information of the strongly coupled neighbourhood
00192     dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00193
00194     // each level stores the information of the tentative prolongations
00195     dBSRmat *tentp = (dBSRmat *)fasp_mem_calloc(max_levels, sizeof(dBSRmat));
00196
00197     for (i=0; i<max_levels; ++i) num_aggs[i] = 0;
00198 }

```

```

00223  /*-----*/
00224  /*-- setup null spaces --*/
00225  /*-----*/
00226
00227  // null space for whole Jacobian
00228  //mgl[0].near_kernel_dim = 1;
00229  //mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));
00230
00231  //for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = NULL;
00232
00233  /*-----*/
00234  /*-- setup ILU param --*/
00235  /*-----*/
00236
00237  // initialize ILU parameters
00238  mgl->ILU_levels = param->ILU_levels;
00239  if ( param->ILU_levels > 0 ) {
00240      iluparam.print_level = param->print_level;
00241      iluparam.ILU_lfil = param->ILU_lfil;
00242      iluparam.ILU_droptol = param->ILU_droptol;
00243      iluparam.ILU_relax = param->ILU_relax;
00244      iluparam.ILU_type = param->ILU_type;
00245  }
00246
00247  /*-----*/
00248  /*-- checking aggregation --*/
00249  /*-----*/
00250
00251  if (param->aggregation_type == PAIRWISE)
00252      param->pair_number = MIN(param->pair_number, max_levels);
00253
00254  // Main AMG setup loop
00255  while ( (mgl[lvl].A.ROW > min_cdoof) && (lvl < max_levels-1) ) {
00256
00257      /*-- setup ILU decomposition if necessary */
00258      if ( lvl < param->ILU_levels ) {
00259          status = fasp_ilu_dbsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00260          if ( status < 0 ) {
00261              if ( prtlvl > PRINT_MIN ) {
00262                  printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00263                  printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00264              }
00265              param->ILU_levels = lvl;
00266          }
00267      }
00268
00269      /*-- get the diagonal inverse --*/
00270      mgl[lvl].diagin = fasp_dbsr_getdiagin(&mgl[lvl].A);
00271
00272      switch ( CondType ) {
00273          case 2:
00274              mgl[lvl].PP = condenseBSR(&mgl[lvl].A); break;
00275          default:
00276              mgl[lvl].PP = condenseBSRLinf(&mgl[lvl].A); break;
00277      }
00278
00279      /*-- Aggregation --*/
00280      switch ( param->aggregation_type ) {
00281
00282          case NPAIR: // unsymmetric pairwise matching aggregation
00283
00284              mgl_csr[lvl].A = mgl[lvl].PP;
00285              status = aggregation_nsympair (mgl_csr, param, lvl, vertices,
00286                                          &num_aggs[lvl]);
00287
00288              break;
00289
00290          default: // symmetric pairwise matching aggregation
00291
00292              mgl_csr[lvl].A = mgl[lvl].PP;
00293              status = aggregation_sympair (mgl_csr, param, lvl, vertices,
00294                                          &num_aggs[lvl]);
00295
00296              // TODO: Need to design better algorithm for pairwise BSR -- Xiaozhe
00297              // TODO: Check why this fails for BSR --Chensong
00298
00299              break;
00300      }
00301
00302      if ( status < 0 ) {
00303          // When error happens, force solver to use the current multigrid levels!

```

```

00304         if ( prtlvl > PRINT_MIN ) {
00305             printf("### WARNING: Aggregation on level-%d failed!\n", lvl);
00306         }
00307         status = FASP_SUCCESS; break;
00308     }
00309
00310     /* -- Form Tentative prolongation --*/
00311     if (lvl == 0 && mgl[0].near_kernel_dim > 0 ) {
00312         form_tentative_p_bsr1(&vertices[lvl], &tentp[lvl], &mgl[0],
00313                             num_aggs[lvl], mgl[0].near_kernel_dim,
00314                             mgl[0].near_kernel_basis);
00315     }
00316     else {
00317         form_boolean_p_bsr(&vertices[lvl], &tentp[lvl], &mgl[0], num_aggs[lvl]);
00318     }
00319
00320     /* -- Smoothing -- */
00321     smooth_agg_bsr(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);
00322
00323     /*-- Form restriction --*/
00324     fasp_dbsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00325
00326     /*-- Form coarse level stiffness matrix --*/
00327     fasp_blas_dbsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);
00328
00329     /*-- Form extra near kernel space if needed --*/
00330     if (mgl[lvl].A_nk != NULL) {
00331
00332         mgl[lvl+1].A_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00333         mgl[lvl+1].P_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00334         mgl[lvl+1].R_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00335
00336         temp1 = fasp_format_dbsr_dcsr(&mgl[lvl].R);
00337         fasp_blas_dcsr_mxm(&temp1, mgl[lvl].P_nk, mgl[lvl+1].P_nk);
00338         fasp_dcsr_trans(mgl[lvl+1].P_nk, mgl[lvl+1].R_nk);
00339         temp2 = fasp_format_dbsr_dcsr(&mgl[lvl+1].A);
00340         fasp_blas_dcsr_rap(mgl[lvl+1].R_nk, &temp2, mgl[lvl+1].P_nk, mgl[lvl+1].A_nk);
00341         fasp_dcsr_free(&temp1);
00342         fasp_dcsr_free(&temp2);
00343     }
00344
00345     fasp_dcsr_free(&Neighbor[lvl]);
00346     fasp_ivec_free(&vertices[lvl]);
00347     fasp_dbsr_free(&tentp[lvl]);
00348
00349     ++lvl;
00350 }
00351
00352 // Setup coarse level systems for direct solvers (BSR version)
00353 switch (csolver) {
00354 #if WITH_MUMPS
00355     case SOLVER_MUMPS: {
00356         // Setup MUMPS direct solver on the coarsest level
00357         mgl[lvl].mumps.job = 1;
00358         mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00359         fasp_solver_mumps_steps(&mgl[lvl].Ac, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00360         break;
00361     }
00362 #endif
00363 #if WITH_UMFPACK
00364     case SOLVER_UMFPACK: {
00365         // Need to sort the matrix A for UMFPACK to work
00366         mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00367         dCSRmat Ac_tran;
00368         fasp_dcsr_trans(&mgl[lvl].Ac, &Ac_tran);
00369         fasp_dcsr_sort(&Ac_tran);
00370         fasp_dcsr_cp(&Ac_tran, &mgl[lvl].Ac);
00371         fasp_dcsr_free(&Ac_tran);
00372         mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].Ac, 0);
00373         break;
00374     }
00375 #endif
00376 #if WITH_SuperLU
00377     case SOLVER_SUPERLU: {
00378         /* Setup SuperLU direct solver on the coarsest level */
00379         mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00380     }
00381 #endif

```

```

00385 #endif
00386
00387 #if WITH_PARDISO
00388     case SOLVER_PARDISO: {
00389         mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00390         fasp_dcsr_sort(&mgl[lvl].Ac);
00391         fasp_pardiso_factorize(&mgl[lvl].Ac, &mgl[lvl].pdata, prtlvl);
00392         break;
00393     }
00394 #endif
00395
00396     default:
00397         // Do nothing!
00398         break;
00399 }
00400
00401 // setup total level number and current level
00402 mgl[0].num_levels = max_levels = lvl+1;
00403 mgl[0].w = fasp_dvec_create(3*m*nb);
00404
00405 if (mgl[0].A_nk != NULL){
00406
00407 #if WITH_UMFPACK
00408     // Need to sort the matrix A_nk for UMFPACK
00409     fasp_dcsr_trans(mgl[0].A_nk, &templ);
00410     fasp_dcsr_sort(&templ);
00411     fasp_dcsr_cp(&templ, mgl[0].A_nk);
00412     fasp_dcsr_free(&templ);
00413     mgl[0].Numeric = fasp_umfpack_factorize(mgl[0].A_nk, 0);
00414 #endif
00415 }
00416
00417 for ( lvl = 1; lvl < max_levels; lvl++ ) {
00418     const INT mm = mgl[lvl].A_ROW*nb;
00419     mgl[lvl].num_levels = max_levels;
00420     mgl[lvl].b = fasp_dvec_create(mm);
00421     mgl[lvl].x = fasp_dvec_create(mm);
00422     mgl[lvl].w = fasp_dvec_create(3*mm);
00423     mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00424
00425     if (mgl[lvl].A_nk != NULL){
00426
00427 #if WITH_UMFPACK
00428     // Need to sort the matrix A_nk for UMFPACK
00429     fasp_dcsr_trans(mgl[lvl].A_nk, &templ);
00430     fasp_dcsr_sort(&templ);
00431     fasp_dcsr_cp(&templ, mgl[lvl].A_nk);
00432     fasp_dcsr_free(&templ);
00433     mgl[lvl].Numeric = fasp_umfpack_factorize(mgl[lvl].A_nk, 0);
00434 #endif
00435 }
00436 }
00437
00438 }
00439
00440 if ( prtlvl > PRINT_NONE ) {
00441     fasp_gettime(&setup_end);
00442     fasp_amgcomplexity_bsr(mgl,prtlvl);
00443     fasp_cputime("Smoothed aggregation (BSR) setup", setup_end - setup_start);
00444 }
00445
00446 fasp_mem_free(vertices); vertices = NULL;
00447 fasp_mem_free(num_aggs); num_aggs = NULL;
00448 fasp_mem_free(Neighbor); Neighbor = NULL;
00449
00450 #if DEBUG_MODE > 0
00451 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00452 #endif
00453
00454 return status;
00455 }
00456
00457
00458 /*-----*/
00459 /*--      End of File      --*/
00460 /*-----*/

```

9.151 PreAMGSetupUA.c File Reference

Unsmoothed aggregation AMG: SETUP phase.

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreAMGAggregation.inl"
#include "PreAMGAggregationCSR.inl"
#include "PreAMGAggregationUA.inl"
```

Functions

- [SHORT fasp_amg_setup_ua](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
Set up phase of unsmoothed aggregation AMG.

9.151.1 Detailed Description

Unsmoothed aggregation AMG: SETUP phase.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlalLUSetupCSR.c](#), [BlaSchwarzSetup.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), and [PreMGRecurAMLI.c](#)

Setup A, P, PT and levels using the unsmoothed aggregation algorithm

Reference: A. Napov and Y. Notay An Algebraic Multigrid Method with Guaranteed Convergence Rate, 2012
Copyright (C) 2011–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGSetupUA.c](#).

9.151.2 Function Documentation

9.151.2.1 fasp_amg_setup_ua()

```
SHORT fasp_amg_setup_ua (
    AMG_data * mgl,
    AMG_param * param )
```

Set up phase of unsmoothed aggregation AMG.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xiaozhe Hu

Date

12/28/2011

Definition at line 55 of file [PreAMGSetupUA.c](#).**9.152 PreAMGSetupUA.c**[Go to the documentation of this file.](#)

```

00001
00022 #include <math.h>
00023 #include <time.h>
00024
00025 #include "fasp.h"
00026 #include "fasp_funcs.h"
00027
00028 /*-----*/
00029 /*--  Declare Private Functions  --*/
00030 /*-----*/
00031
00032 #include "PreAMGAggregation.inl"
00033 #include "PreAMGAggregationCSR.inl"
00034 #include "PreAMGAggregationUA.inl"
00035
00036 static SHORT amg_setup_unsmoothP_unsmoothR(AMG_data *, AMG_param *);
00037
00038 /*-----*/
00039 /*--      Public Functions      --*/
00040 /*-----*/
00041
00055 SHORT fasp_amg_setup_ua (AMG_data *mgl,
00056                          AMG_param *param)
00057 {
00058     const SHORT prtlvl = param->print_level;
00059
00060     // Output some info for debugging
00061     if ( prtlvl > PRINT_NONE ) printf("\nSetting up UA AMG ...\n");
00062
00063     #if DEBUG_MODE > 0
00064         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00065         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
00066             mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00067     #endif
00068
00069     SHORT status = amg_setup_unsmoothP_unsmoothR(mgl, param);
00070
00071     #if DEBUG_MODE > 0
00072         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00073     #endif
00074
00075     return status;
00076 }
00077
00078 /*-----*/
00079 /*--      Private Functions      --*/
00080 /*-----*/
00081
00101 static SHORT amg_setup_unsmoothP_unsmoothR(AMG_data *mgl,
00102                                             AMG_param *param) {
00103     const SHORT prtlvl = param->print_level;
00104     const SHORT cycle_type = param->cycle_type;
00105     const SHORT csolver = param->coarse_solver;
00106     const SHORT min_cdof = MAX(param->coarse_dof, 50);
00107     const INT m = mgl[0].A.row;
00108
00109     // empiric value
00110     const REAL cplxmax = 3.0;
00111     const REAL xsi = 0.6;
00112     INT icum = 1;
00113     REAL eta, fracratio;
00114
00115     // local variables

```

```

00116     SHORT max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
00117     INT i;
00118     REAL setup_start, setup_end;
00119     ILU_param iluparam;
00120     SWZ_param swzparam;
00121
00122     #if DEBUG_MODE > 0
00123         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00124         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
00125             mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00126     #endif
00127
00128     fasp_gettime(&setup_start);
00129
00130     // level info (fine: 0; coarse: 1)
00131     ivector *vertices = (ivector *) fasp_mem_calloc(max_levels, sizeof(ivector));
00132
00133     // each level stores the information of the number of aggregations
00134     INT *num_aggs = (INT *) fasp_mem_calloc(max_levels, sizeof(INT));
00135
00136     // each level stores the information of the strongly coupled neighborhoods
00137     dCSRmat *Neighbor = (dCSRmat *) fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00138
00139     // Initialize level information
00140     for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;
00141
00142     mgl[0].near_kernel_dim = 1;
00143     mgl[0].near_kernel_basis = (REAL **) fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL * ));
00144
00145     for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
00146         mgl[0].near_kernel_basis[i] = (REAL *) fasp_mem_calloc(m, sizeof(REAL));
00147         fasp_darray_set(m, mgl[0].near_kernel_basis[i], 1.0);
00148     }
00149
00150     // Initialize ILU parameters
00151     mgl->ILU_levels = param->ILU_levels;
00152     if ( param->ILU_levels > 0 ) {
00153         iluparam.print_level = param->print_level;
00154         iluparam.ILU_lfil = param->ILU_lfil;
00155         iluparam.ILU_droptol = param->ILU_droptol;
00156         iluparam.ILU_relax = param->ILU_relax;
00157         iluparam.ILU_type = param->ILU_type;
00158     }
00159
00160     // Initialize Schwarz parameters
00161     mgl->SWZ_levels = param->SWZ_levels;
00162     if ( param->SWZ_levels > 0 ) {
00163         swzparam.SWZ_mmsize = param->SWZ_mmsize;
00164         swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
00165         swzparam.SWZ_type = param->SWZ_type;
00166         swzparam.SWZ_blksolver = param->SWZ_blksolver;
00167     }
00168
00169     // Initialize AMLI coefficients
00170     if ( cycle_type == AMLI_CYCLE ) {
00171         const INT amlideg = param->amli_degree;
00172         param->amli_coef = (REAL *) fasp_mem_calloc(amlideg + 1, sizeof(REAL));
00173         REAL lambda_max = 2.0, lambda_min = lambda_max / 4;
00174         fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
00175     }
00176
00177     #if DIAGONAL_PREF
00178         fasp_dcsr_diagpref(&mgl[0].A); // reorder each row to make diagonal appear first
00179     #endif
00180
00181     /*-----*/
00182     /*--- checking aggregation ---*/
00183     /*-----*/
00184
00185     // Pairwise matching algorithm requires diagonal preference ordering
00186     if ( param->aggregation_type == PAIRWISE ) {
00187         param->pair_number = MIN(param->pair_number, max_levels);
00188         fasp_dcsr_diagpref(&mgl[0].A);
00189     }
00190
00191     // Main AMG setup loop
00192     while ((mgl[lvl].A.row > min_cdof) && (lvl < max_levels - 1)) {
00193
00194         #if DEBUG_MODE > 1
00195             printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
00196                 lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);

```

```

00197 #endif
00198
00199     /*--- Setup ILU decomposition if necessary */
00200     if ( lvl < param->ILU_levels ) {
00201         status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00202         if ( status < 0 ) {
00203             if ( prtlvl > PRINT_MIN ) {
00204                 printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00205                 printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00206             }
00207             param->ILU_levels = lvl;
00208         }
00209     }
00210
00211     /*--- Setup Schwarz smoother if necessary */
00212     if ( lvl < param->SWZ_levels ) {
00213         mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
00214         fasp_dcsr_shift(&mgl[lvl].Schwarz.A, 1);
00215         status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
00216         if ( status < 0 ) {
00217             if ( prtlvl > PRINT_MIN ) {
00218                 printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
00219                 printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
00220             }
00221             param->SWZ_levels = lvl;
00222         }
00223     }
00224
00225     /*--- Aggregation ---*/
00226     switch ( param->aggregation_type ) {
00227
00228         case VMB: // VMB aggregation
00229             status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl + 1,
00230                                     &Neighbor[lvl], &num_aggs[lvl]);
00231
00232             /*--- Choose strength threshold adaptively ---*/
00233             if ( num_aggs[lvl] * 4.0 > mgl[lvl].A.row )
00234                 param->strong_coupled /= 2.0;
00235             else if ( num_aggs[lvl] * 1.25 < mgl[lvl].A.row )
00236                 param->strong_coupled *= 2.0;
00237
00238             break;
00239
00240         case NPAIR: // non-symmetric pairwise matching aggregation
00241             status = aggregation_nsypair(mgl, param, lvl, vertices, &num_aggs[lvl]);
00242             /*--- Modified by Chunsheng Feng on 10/17/2020, ZCS on 01/15/2021:
00243 if NPAIR fail, switch aggregation type to VMB. ---*/
00244             if ( status != FASP_SUCCESS || num_aggs[lvl] * 2.0 > mgl[lvl].A.row ) {
00245                 if ( prtlvl > PRINT_MORE ) {
00246                     printf("### WARNING: Non-symmetric pairwise matching failed on level %d!\n", lvl);
00247                     printf("### WARNING: Switch to VMB aggregation on level %d!\n", lvl);
00248                 }
00249                 param->aggregation_type = VMB;
00250                 status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl + 1,
00251                                         &Neighbor[lvl], &num_aggs[lvl]);
00252             }
00253
00254             break;
00255
00256         default: // symmetric pairwise matching aggregation
00257             status = aggregation_sympair(mgl, param, lvl, vertices, &num_aggs[lvl]);
00258         }
00259
00260     // Check 1: Did coarsening step succeed?
00261     if ( status < 0 ) {
00262         // When error happens, stop at the current multigrid level!
00263         if ( prtlvl > PRINT_MIN ) {
00264             printf("### WARNING: Stop coarsening on level %d!\n", lvl);
00265         }
00266         status = FASP_SUCCESS;
00267         fasp_ivec_free(&vertices[lvl]);
00268         fasp_dcsr_free(&Neighbor[lvl]);
00269         break;
00270     }
00271
00272     /*--- Form Prolongation ---*/
00273     form_tentative_p(&vertices[lvl], &mgl[lvl].P, mgl[0].near_kernel_basis,
00274                     num_aggs[lvl]);
00275
00276     // Check 2: Is coarse sparse too small?
00277

```



```

00278         if ( mgl[lvl].P.col < MIN_CDOF ) {
00279             fasp_ivec_free(&vertices[lvl]);
00280             fasp_dcsr_free(&Neighbor[lvl]);
00281             break;
00282         }
00283
00284         // Check 3: Does this coarsening step too aggressive?
00285         if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
00286             if ( prtlvl > PRINT_MIN ) {
00287                 printf("### WARNING: Coarsening might be too aggressive!\n");
00288                 printf("### WARNING: Fine level = %d, coarse level = %d. Discard!\n",
00289                     mgl[lvl].P.row, mgl[lvl].P.col);
00290             }
00291             fasp_ivec_free(&vertices[lvl]);
00292             fasp_dcsr_free(&Neighbor[lvl]);
00293             break;
00294         }
00295
00296         /*-- Form restriction --*/
00297         fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00298
00299         /*-- Form coarse level stiffness matrix --*/
00300         fasp_blas_dcsr_rap_agg(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P,
00301             &mgl[lvl + 1].A);
00302
00303         fasp_dcsr_free(&Neighbor[lvl]);
00304         fasp_ivec_free(&vertices[lvl]);
00305
00306         ++lvl;
00307
00308         #if DIAGONAL_PREF
00309             fasp_dcsr_diagpref(&mgl[lvl].A); // reorder each row to make diagonal appear first
00310         #endif
00311
00312         // Check 4: Is this coarsening ratio too small?
00313         if ((REAL) mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
00314             param->quality_bound *= 2.0;
00315         }
00316
00317     } // end of the main while loop
00318
00319     // Setup coarse level systems for direct solvers
00320     switch ( csolver ) {
00321
00322     #if WITH_MUMPS
00323         case SOLVER_MUMPS: {
00324             // Setup MUMPS direct solver on the coarsest level
00325             mgl[lvl].mumps.job = 1;
00326             fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00327             break;
00328         }
00329     #endif
00330
00331     #if WITH_UMFPACK
00332         case SOLVER_UMFPACK: {
00333             // Need to sort the matrix A for UMFPACK to work
00334             dCSRmat Ac_tran;
00335             Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
00336             fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
00337             // It is equivalent to do transpose and then sort
00338             // fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
00339             // fasp_dcsr_sort(&Ac_tran);
00340             fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
00341             fasp_dcsr_free(&Ac_tran);
00342             mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
00343             break;
00344         }
00345     #endif
00346
00347     #if WITH_PARDISO
00348         case SOLVER_PARDISO: {
00349             fasp_dcsr_sort(&mgl[lvl].A);
00350             fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
00351             break;
00352         }
00353     #endif
00354
00355         default: // Do nothing!
00356             break;
00357     }
00358

```

```

00359 // setup total level number and current level
00360 mgl[0].num_levels = max_levels = lvl + 1;
00361 mgl[0].w = fasp_dvec_create(m);
00362
00363 for ( lvl = 1; lvl < max_levels; ++lvl ) {
00364     INT mm = mgl[lvl].A.row;
00365     mgl[lvl].num_levels = max_levels;
00366     mgl[lvl].b = fasp_dvec_create(mm);
00367     mgl[lvl].x = fasp_dvec_create(mm);
00368
00369     mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
00370     mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00371     mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!
00372
00373     if ( cycle_type == NL_AMLI_CYCLE )
00374         mgl[lvl].w = fasp_dvec_create(3 * mm);
00375     else
00376         mgl[lvl].w = fasp_dvec_create(2 * mm);
00377 }
00378
00379 // setup for cycle type of unsmoothed aggregation
00380 eta = xsi / ((1 - xsi) * (cplxmax - 1));
00381 mgl[0].cycle_type = 1;
00382 mgl[max_levels - 1].cycle_type = 0;
00383
00384 for ( lvl = 1; lvl < max_levels - 1; ++lvl ) {
00385     fracratio = (REAL) mgl[lvl].A.nnz / mgl[0].A.nnz;
00386     mgl[lvl].cycle_type = (INT) (pow((REAL) xsi, (REAL) lvl) / (eta * fracratio * icum));
00387     // safe-guard step: make cycle type >= 1 and <= 2
00388     mgl[lvl].cycle_type = MAX(1, MIN(2, mgl[lvl].cycle_type));
00389     icum = icum * mgl[lvl].cycle_type;
00390 }
00391
00392 #if MULTI_COLOR_ORDER
00393     INT Colors, rowmax;
00394 #ifdef _OPENMP
00395     int threads = fasp_get_num_threads();
00396     if (threads > max_levels-1 ) threads = max_levels-1;
00397 #pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
00398 #endif
00399     for (lvl=0; lvl<max_levels-1; lvl++){
00400
00401 #if 1
00402         dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
00403 #else
00404         dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
00405 #endif
00406         if ( prtlvl > 1 )
00407             printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.21f, colors = %5d, Theta = %1e.\n",
00408                 lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
00409                 mgl[lvl].A.color, mgl[lvl].GS_Theta);
00410     }
00411 #endif
00412
00413     if ( prtlvl > PRINT_NONE ) {
00414         fasp_gettime(&setup_end);
00415         fasp_amgcomplexity(mgl, prtlvl);
00416         fasp_cputime("Unsmoothed aggregation setup", setup_end - setup_start);
00417     }
00418
00419     fasp_mem_free(Neighbor);
00420     Neighbor = NULL;
00421     fasp_mem_free(vertices);
00422     vertices = NULL;
00423     fasp_mem_free(num_aggs);
00424     num_aggs = NULL;
00425
00426 #if DEBUG_MODE > 0
00427     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00428 #endif
00429
00430     return status;
00431 }
00432
00433 /*-----*/
00434 /*--          End of File          --*/
00435 /*-----*/

```

9.153 PreAMGSetupUABSR.c File Reference

Unsmoothed aggregation AMG: SETUP phase (for BSR matrices)

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreAMGAggregation.inl"
#include "PreAMGAggregationBSR.inl"
#include "PreAMGAggregationUA.inl"
```

Functions

- [SHORT fasp_amg_setup_ua_bsr](#) ([AMG_data_bsr](#) *mgl, [AMG_param](#) *param)
Set up phase of unsmoothed aggregation AMG (BSR format)

9.153.1 Detailed Description

Unsmoothed aggregation AMG: SETUP phase (for BSR matrices)

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaFormat.c](#), [BlaILUSetupBSR.c](#), [BlaSparseBLC.c](#), [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), and [PreDataInit.c](#)

Setup A, P, PT and levels using the unsmoothed aggregation algorithm

Reference: P. Vanek, J. Madel and M. Brezina Algebraic Multigrid on Unstructured Meshes, 1994
Copyright (C) 2012–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreAMGSetupUABSR.c](#).

9.153.2 Function Documentation

9.153.2.1 fasp_amg_setup_ua_bsr()

```
INT fasp_amg_setup_ua_bsr (
    AMG\_data\_bsr * mgl,
    AMG\_param * param )
```

Set up phase of unsmoothed aggregation AMG (BSR format)

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data_bsr
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if succeeded; otherwise, error information.

Author

Xiaozhe Hu

Date

03/16/2012

Definition at line 55 of file [PreAMGSetupUABSR.c](#).**9.154 PreAMGSetupUABSR.c**[Go to the documentation of this file.](#)

```

00001
00022 #include <math.h>
00023 #include <time.h>
00024
00025 #include "fasp.h"
00026 #include "fasp_functs.h"
00027
00028 /*-----*/
00029 /*--  Declare Private Functions  --*/
00030 /*-----*/
00031
00032 #include "PreAMGAggregation.inl"
00033 #include "PreAMGAggregationBSR.inl"
00034 #include "PreAMGAggregationUA.inl"
00035
00036 static SHORT amg_setup_unsmoothP_unsmoothR_bsr (AMG_data_bsr *, AMG_param *);
00037
00038 /*-----*/
00039 /*--      Public Functions      --*/
00040 /*-----*/
00041
00055 SHORT fasp_amg_setup_ua_bsr (AMG_data_bsr *mgl,
00056                             AMG_param *param)
00057 {
00058     #if DEBUG_MODE > 0
00059         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00060     #endif
00061
00062     SHORT status = amg_setup_unsmoothP_unsmoothR_bsr(mgl, param);
00063
00064     #if DEBUG_MODE > 0
00065         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00066     #endif
00067
00068     return status;
00069 }
00070
00071 /*-----*/
00072 /*--      Private Functions      --*/
00073 /*-----*/
00074
00092 static SHORT amg_setup_unsmoothP_unsmoothR_bsr (AMG_data_bsr *mgl,
00093                                                 AMG_param *param)
00094 {
00095     const SHORT CondType = 1; // Condensation method used for AMG
00096
00097     const SHORT prtlvl = param->print_level;
00098     const SHORT csolver = param->coarse_solver;
00099     const SHORT min_cdof = MAX(param->coarse_dof, 50);
00100     const INT m = mgl[0].A.ROW;
00101     const INT nb = mgl[0].A.nb;
00102
00103     SHORT max_levels = param->max_levels;
00104     SHORT i, lvl = 0, status = FASP_SUCCESS;
00105     REAL setup_start, setup_end;
00106
00107     AMG_data *mgl_csr = fasp_amg_data_create(max_levels);
00108
00109     dCSRmat temp1, temp2;
00110
00111     #if DEBUG_MODE > 0
00112         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00113         printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",

```

```

00114         mgl[0].A.ROW, mgl[0].A.COL, mgl[0].A.NNZ);
00115 #endif
00116
00117     fasp_gettime(&setup_start);
00118
00119     /*-----*/
00120     /*--local working array--*/
00121     /*-----*/
00122
00123     // level info (fine: 0; coarse: 1)
00124     ivector *vertices = (ivector *)fasp_mem_calloc(max_levels, sizeof(ivector));
00125
00126     //each elvel stores the information of the number of aggregations
00127     INT *num_aggs = (INT *)fasp_mem_calloc(max_levels, sizeof(INT));
00128
00129     // each level stores the information of the strongly coupled neighborhoods
00130     dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));
00131
00132     for ( i=0; i<max_levels; ++i ) num_aggs[i] = 0;
00133
00134     /*-----*/
00135     /*-- setup null spaces for whole Jacobian --*/
00136     /*-----*/
00137
00138     /*
00139     mgl[0].near_kernel_dim = 1;
00140     mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));
00141
00142     for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = NULL;
00143     */
00144
00145     /*-----*/
00146     /*-- setup ILU param --*/
00147     /*-----*/
00148
00149     // initialize ILU parameters
00150     mgl->ILU_levels = param->ILU_levels;
00151     ILU_param iluparam;
00152
00153     if ( param->ILU_levels > 0 ) {
00154         iluparam.print_level = param->print_level;
00155         iluparam.ILU_lfil = param->ILU_lfil;
00156         iluparam.ILU_droptol = param->ILU_droptol;
00157         iluparam.ILU_relax = param->ILU_relax;
00158         iluparam.ILU_type = param->ILU_type;
00159     }
00160
00161     /*-----*/
00162     /*--- checking aggregation ---*/
00163     /*-----*/
00164     if (param->aggregation_type == PAIRWISE)
00165         param->pair_number = MIN(param->pair_number, max_levels);
00166
00167     // Main AMG setup loop
00168     while ( (mgl[lvl].A.ROW > min_cdof) && (lvl < max_levels-1) ) {
00169
00170         /*-- setup ILU decomposition if necessary */
00171         if ( lvl < param->ILU_levels ) {
00172             status = fasp_ilu_dbsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
00173             if ( status < 0 ) {
00174                 if ( prtlvl > PRINT_MIN ) {
00175                     printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
00176                     printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
00177                 }
00178                 param->ILU_levels = lvl;
00179             }
00180         }
00181
00182         /*-- get the diagonal inverse --*/
00183         mgl[lvl].diaginv = fasp_dbsr_getdiaginv(&mgl[lvl].A);
00184
00185         switch ( CondType ) {
00186             case 2:
00187                 mgl[lvl].PP = condenseBSR(&mgl[lvl].A); break;
00188             default:
00189                 mgl[lvl].PP = condenseBSRLinf(&mgl[lvl].A); break;
00190         }
00191
00192         /*-- Aggregation --*/
00193         switch ( param->aggregation_type ) {
00194

```

```

00195         case VMB: // VMB aggregation
00196
00197             status = aggregation_vmb (&mgl[lvl].PP, &vertices[lvl], param,
00198                                     lvl+1, &Neighbor[lvl], &num_aggs[lvl]);
00199
00200             /*-- Choose strength threshold adaptively --*/
00201             if ( num_aggs[lvl]*4 > mgl[lvl].PP.row )
00202                 param->strong_coupled /= 4;
00203             else if ( num_aggs[lvl]*1.25 < mgl[lvl].PP.row )
00204                 param->strong_coupled *= 1.5;
00205
00206             break;
00207
00208         case NPAIR: // non-symmetric pairwise matching aggregation
00209
00210             mgl_csr[lvl].A = mgl[lvl].PP;
00211             status = aggregation_nsympair (mgl_csr, param, lvl, vertices,
00212                                           &num_aggs[lvl]);
00213
00214             break;
00215
00216         default: // symmetric pairwise matching aggregation
00217
00218             mgl_csr[lvl].A = mgl[lvl].PP;
00219             status = aggregation_sympair (mgl_csr, param, lvl, vertices,
00220                                         &num_aggs[lvl]);
00221
00222             // TODO: Need to design better algorithm for pairwise BSR -- Xiaozhe
00223             // TODO: Unsymmetric pairwise aggregation not finished -- Chensong
00224             // TODO: Check why this fails for BSR --Chensong
00225
00226             break;
00227     }
00228
00229     if ( status < 0 ) {
00230         // When error happens, force solver to use the current multigrid levels!
00231         if ( prtlvl > PRINT_MIN ) {
00232             printf("### WARNING: Forming aggregates on level-%d failed!\n", lvl);
00233         }
00234         status = FASP_SUCCESS; break;
00235     }
00236
00237     /* -- Form Prolongation --*/
00238     if ( lvl == 0 && mgl[0].near_kernel_dim > 0 ) {
00239         form_tentative_p_bsr1(&vertices[lvl], &mgl[lvl].P, &mgl[0],
00240                             num_aggs[lvl], mgl[0].near_kernel_dim,
00241                             mgl[0].near_kernel_basis);
00242     }
00243     else {
00244         form_boolean_p_bsr(&vertices[lvl], &mgl[lvl].P, &mgl[0], num_aggs[lvl]);
00245     }
00246
00247     /*-- Form restriction --*/
00248     fasp_dbsr_trans(&mgl[lvl].P, &mgl[lvl].R);
00249
00250     /*-- Form coarse level stiffness matrix --*/
00251     fasp_blas_dbsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);
00252
00253     /* -- Form extra near kernel space if needed --*/
00254     if (mgl[lvl].A_nk != NULL){
00255
00256         mgl[lvl+1].A_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00257         mgl[lvl+1].P_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00258         mgl[lvl+1].R_nk = (dCSRmat *)fasp_mem_malloc(1, sizeof(dCSRmat));
00259
00260         temp1 = fasp_format_dbsr_dcsr(&mgl[lvl].R);
00261         fasp_blas_dcsr_mxm(&temp1, mgl[lvl].P_nk, mgl[lvl+1].P_nk);
00262         fasp_dcsr_trans(mgl[lvl+1].P_nk, mgl[lvl+1].R_nk);
00263         temp2 = fasp_format_dbsr_dcsr(&mgl[lvl+1].A);
00264         fasp_blas_dcsr_rap(mgl[lvl+1].R_nk, &temp2, mgl[lvl+1].P_nk, mgl[lvl+1].A_nk);
00265         fasp_dcsr_free(&temp1);
00266         fasp_dcsr_free(&temp2);
00267     }
00268
00269     fasp_dcsr_free(&Neighbor[lvl]);
00270     fasp_ivec_free(&vertices[lvl]);
00271
00272     ++lvl;
00273 }
00274
00275

```

```

00276     // Setup coarse level systems for direct solvers (BSR version)
00277     switch (csolver) {
00278
00279     #if WITH_MUMPS
00280         case SOLVER_MUMPS: {
00281             // Setup MUMPS direct solver on the coarsest level
00282             mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00283             mgl[lvl].mumps.job = 1;
00284             fasp_solver_mumps_steps(&mgl[lvl].Ac, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
00285             break;
00286         }
00287     #endif
00288
00289     #if WITH_UMFPACK
00290         case SOLVER_UMFPACK: {
00291             // Need to sort the matrix A for UMFPACK to work
00292             mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00293             dCSRmat Ac_tran;
00294             fasp_dcsr_trans(&mgl[lvl].Ac, &Ac_tran);
00295             fasp_dcsr_sort(&Ac_tran);
00296             fasp_dcsr_cp(&Ac_tran, &mgl[lvl].Ac);
00297             fasp_dcsr_free(&Ac_tran);
00298             mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].Ac, 0);
00299             break;
00300         }
00301     #endif
00302
00303     #if WITH_SuperLU
00304         case SOLVER_SUPERLU: {
00305             /* Setup SuperLU direct solver on the coarsest level */
00306             mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00307         }
00308     #endif
00309
00310     #if WITH_PARDISO
00311         case SOLVER_PARDISO: {
00312             mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
00313             fasp_dcsr_sort(&mgl[lvl].Ac);
00314             fasp_pardiso_factorize(&mgl[lvl].Ac, &mgl[lvl].pdata, prtlvl);
00315             break;
00316         }
00317     #endif
00318
00319     default:
00320         // Do nothing!
00321         break;
00322     }
00323
00324
00325     // setup total level number and current level
00326     mgl[0].num_levels = max_levels = lvl+1;
00327     mgl[0].w = fasp_dvec_create(3*m*nb);
00328
00329     if (mgl[0].A_nk != NULL){
00330
00331     #if WITH_UMFPACK
00332         // Need to sort the matrix A_nk for UMFPACK
00333         fasp_dcsr_trans(mgl[0].A_nk, &templ);
00334         fasp_dcsr_sort(&templ);
00335         fasp_dcsr_cp(&templ, mgl[0].A_nk);
00336         fasp_dcsr_free(&templ);
00337     #endif
00338
00339     }
00340
00341     for ( lvl = 1; lvl < max_levels; lvl++ ) {
00342         const INT mm = mgl[lvl].A.ROW*nb;
00343         mgl[lvl].num_levels = max_levels;
00344         mgl[lvl].b = fasp_dvec_create(mm);
00345         mgl[lvl].x = fasp_dvec_create(mm);
00346         mgl[lvl].w = fasp_dvec_create(3*mm);
00347         mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
00348
00349         if (mgl[lvl].A_nk != NULL){
00350
00351         #if WITH_UMFPACK
00352             // Need to sort the matrix A_nk for UMFPACK
00353             fasp_dcsr_trans(mgl[lvl].A_nk, &templ);
00354             fasp_dcsr_sort(&templ);
00355             fasp_dcsr_cp(&templ, mgl[lvl].A_nk);
00356             fasp_dcsr_free(&templ);

```

```

00357 #endif
00358
00359     }
00360
00361     }
00362
00363     if ( prtlvl > PRINT_NONE ) {
00364         fasp_gettime(&setup_end);
00365         fasp_amgcomplexity_bsr(mgl,prtlvl);
00366         fasp_cputime("Unsmoothed aggregation (BSR) setup", setup_end - setup_start);
00367     }
00368
00369     fasp_mem_free(vertices); vertices = NULL;
00370     fasp_mem_free(num_aggs); num_aggs = NULL;
00371     fasp_mem_free(Neighbor); Neighbor = NULL;
00372
00373 #if DEBUG_MODE > 0
00374     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00375 #endif
00376
00377     return status;
00378 }
00379
00380 /*-----*/
00381 /*--          End of File          --*/
00382 /*-----*/

```

9.155 PreBLC.c File Reference

Preconditioners for **dBLCmat** matrices.

```

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_funcs.h"

```

Functions

- void **fasp_precond_dbkc_diag_3** (REAL *r, REAL *z, void *data)
Block diagonal preconditioner (3x3 blocks)
- void **fasp_precond_dbkc_diag_3_amg** (REAL *r, REAL *z, void *data)
Block diagonal preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_diag_4** (REAL *r, REAL *z, void *data)
Block diagonal preconditioning (4x4 blocks)
- void **fasp_precond_dbkc_lower_3** (REAL *r, REAL *z, void *data)
block lower triangular preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_lower_3_amg** (REAL *r, REAL *z, void *data)
block lower triangular preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_lower_4** (REAL *r, REAL *z, void *data)
block lower triangular preconditioning (4x4 blocks)
- void **fasp_precond_dbkc_upper_3** (REAL *r, REAL *z, void *data)
block upper triangular preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_upper_3_amg** (REAL *r, REAL *z, void *data)
block upper triangular preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_SGS_3** (REAL *r, REAL *z, void *data)
Block symmetric GS preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_SGS_3_amg** (REAL *r, REAL *z, void *data)
Block symmetric GS preconditioning (3x3 blocks)
- void **fasp_precond_dbkc_sweeping** (REAL *r, REAL *z, void *data)
Sweeping preconditioner for Maxwell equations.

9.155.1 Detailed Description

Preconditioners for [dBLCmat](#) matrices.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxVector.c](#), [BlaSpmvCSR.c](#), and [PreMGCycle.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

TODO: Separate solve and setup phases for direct solvers!!! –Chensong

Definition in file [PreBLC.c](#).

9.155.2 Function Documentation

9.155.2.1 fasp_precond_dblc_diag_3()

```
void fasp_precond_dblc_diag_3 (
    REAL * r,
    REAL * z,
    void * data )
```

Block diagonal preconditioner (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved exactly

Definition at line [38](#) of file [PreBLC.c](#).

9.155.2.2 fasp_precond_dblc_diag_3_amg()

```
void fasp_precond_dblc_diag_3_amg (
    REAL * r,
    REAL * z,
    void * data )
```

Block diagonal preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved by AMG

Definition at line 126 of file [PreBLC.c](#).**9.155.2.3 fasp_precond_dblc_diag_4()**

```
void fasp_precond_dblc_diag_4 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Block diagonal preconditioning (4x4 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved exactly

Definition at line 191 of file [PreBLC.c](#).**9.155.2.4 fasp_precond_dblc_lower_3()**

```
void fasp_precond_dblc_lower_3 (  
    REAL * r,
```

```
    REAL * z,  
    void * data )
```

block lower triangular preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved exactly

Definition at line 291 of file [PreBLC.c](#).**9.155.2.5 fasp_precond_dblc_lower_3_amg()**

```
void fasp_precond_dblc_lower_3_amg (  
    REAL * r,  
    REAL * z,  
    void * data )
```

block lower triangular preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved by AMG

Definition at line 379 of file [PreBLC.c](#).**9.155.2.6 fasp_precond_dblc_lower_4()**

```
void fasp_precond_dblc_lower_4 (  
    REAL * r,
```

```
    REAL * z,  
    void * data )
```

block lower triangular preconditioning (4x4 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

07/10/2014

Note

Each diagonal block is solved exactly

Definition at line 453 of file [PreBLC.c](#).**9.155.2.7 fasp_precond_dblc_SGS_3()**

```
void fasp_precond_dblc_SGS_3 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Block symmetric GS preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/19/2015

Note

Each diagonal block is solved exactly

Definition at line 725 of file [PreBLC.c](#).**9.155.2.8 fasp_precond_dblc_SGS_3_amg()**

```
void fasp_precond_dblc_SGS_3_amg (  
    REAL * r,
```

```
    REAL * z,  
    void * data )
```

Block symmetric GS preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/19/2015

Note

Each diagonal block is solved by AMG

Definition at line 838 of file [PreBLC.c](#).**9.155.2.9 fasp_precond_dblc_sweeping()**

```
void fasp_precond_dblc_sweeping (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Sweeping preconditioner for Maxwell equations.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

05/01/2014

Note

Each diagonal block is solved exactly

Definition at line 939 of file [PreBLC.c](#).**9.155.2.10 fasp_precond_dblc_upper_3()**

```
void fasp_precond_dblc_upper_3 (  
    REAL * r,
```



```
    REAL * z,  
    void * data )
```

block upper triangular preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/18/2015

Note

Each diagonal block is solved exactly

Definition at line 557 of file [PreBLC.c](#).**9.155.2.11 fasp_precond_dblc_upper_3_amg()**

```
void fasp_precond_dblc_upper_3_amg (  
    REAL * r,  
    REAL * z,  
    void * data )
```

block upper triangular preconditioning (3x3 blocks)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/19/2015

Note

Each diagonal block is solved by AMG

Definition at line 645 of file [PreBLC.c](#).**9.156 PreBLC.c**[Go to the documentation of this file.](#)

```
00001  
00016 #include "fasp.h"  
00017 #include "fasp_block.h"
```

```

00018 #include "fasp_functs.h"
00019
00020 /*-----*/
00021 /*--      Public Functions      --*/
00022 /*-----*/
00023
00038 void fasp_precond_dblc_diag_3 (REAL *r,
00039                               REAL *z,
00040                               void *data)
00041 {
00042     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00043
00044         precondition_data_blc *prepdata = (precondition_data_blc *)data;
00045         dCSRmat *A_diag = prepdata->A_diag;
00046         dvector *tempr = &(prepdata->r);
00047
00048         const INT N0 = A_diag[0].row;
00049         const INT N1 = A_diag[1].row;
00050         const INT N2 = A_diag[2].row;
00051         const INT N = N0 + N1 + N2;
00052
00053         // back up r, setup z;
00054         fasp_darray_cp(N, r, tempr->val);
00055         fasp_darray_set(N, z, 0.0);
00056
00057         // prepare
00058         #if WITH_UMFPACK
00059             void **LU_diag = prepdata->LU_diag;
00060             dvector r0, r1, r2, z0, z1, z2;
00061
00062             r0.row = N0; z0.row = N0;
00063             r1.row = N1; z1.row = N1;
00064             r2.row = N2; z2.row = N2;
00065
00066             r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00067             z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00068         #elif WITH_SuperLU
00069             dvector r0, r1, r2, z0, z1, z2;
00070
00071             r0.row = N0; z0.row = N0;
00072             r1.row = N1; z1.row = N1;
00073             r2.row = N2; z2.row = N2;
00074
00075             r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00076             z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00077         #endif
00078
00079         // Preconditioning A00 block
00080         #if WITH_UMFPACK
00081             /* use UMFPACK direct solver */
00082             fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00083         #elif WITH_SuperLU
00084             /* use SuperLU direct solver */
00085             fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00086         #endif
00087
00088         // Preconditioning A11 block
00089         #if WITH_UMFPACK
00090             /* use UMFPACK direct solver */
00091             fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00092         #elif WITH_SuperLU
00093             /* use SuperLU direct solver */
00094             fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00095         #endif
00096
00097         // Preconditioning A22 block
00098         #if WITH_UMFPACK
00099             /* use UMFPACK direct solver */
00100             fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00101         #elif WITH_SuperLU
00102             /* use SuperLU direct solver */
00103             fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00104         #endif
00105
00106         // restore r
00107         fasp_darray_cp(N, tempr->val, r);
00108     #endif
00109 }
00110
00126 void fasp_precond_dblc_diag_3_amg (REAL *r,

```

```

00127                                     REAL *z,
00128                                     void *data)
00129 {
00130     precondition_data_blc *prepdata = (precondition_data_blc *)data;
00131     dBLCmat *A      = prepdata->Ablc;
00132     dvector *tempr = &(prepdata->r);
00133
00134     AMG_param *amgparam = prepdata->amgparam;
00135     AMG_data **mgl      = prepdata->mgl;
00136
00137     const INT N0 = A->blocks[0]->row;
00138     const INT N1 = A->blocks[4]->row;
00139     const INT N2 = A->blocks[8]->row;
00140     const INT N  = N0 + N1 + N2;
00141
00142     // back up r, setup z;
00143     fasp_darray_cp(N, r, tempr->val);
00144     fasp_darray_set(N, z, 0.0);
00145
00146     // prepare
00147     dvector r0, r1, r2, z0, z1, z2;
00148     r0.row = N0; z0.row = N0; r1.row = N1; z1.row = N1; r2.row = N2; z2.row = N2;
00149     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); z0.val = z;
00150     z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00151
00152     // Preconditioning A00 block
00153     mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
00154     mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);
00155
00156     fasp_solver_mgcycle(mgl[0], amgparam);
00157     fasp_darray_cp(N0, mgl[0]->x.val, z0.val);
00158
00159     // Preconditioning A11 block
00160     mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
00161     mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x, 0.0);
00162
00163     fasp_solver_mgcycle(mgl[1], amgparam);
00164     fasp_darray_cp(N1, mgl[1]->x.val, z1.val);
00165
00166     // Preconditioning A22 block
00167     mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
00168     mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x, 0.0);
00169
00170     fasp_solver_mgcycle(mgl[2], amgparam);
00171     fasp_darray_cp(N2, mgl[2]->x.val, z2.val);
00172
00173     // restore r
00174     fasp_darray_cp(N, tempr->val, r);
00175 }
00176
00191 void fasp_precond_dbldiag_4 (REAL *r,
00192                             REAL *z,
00193                             void *data)
00194 {
00195     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00196
00197     precondition_data_blc *prepdata=(precondition_data_blc *)data;
00198     dCSRmat *A_diag = prepdata->A_diag;
00199     dvector *tempr = &(prepdata->r);
00200
00201     const INT N0 = A_diag[0].row;
00202     const INT N1 = A_diag[1].row;
00203     const INT N2 = A_diag[2].row;
00204     const INT N3 = A_diag[3].row;
00205     const INT N  = N0 + N1 + N2 + N3;
00206
00207     // back up r, setup z;
00208     fasp_darray_cp(N, r, tempr->val);
00209     fasp_darray_set(N, z, 0.0);
00210
00211     // prepare
00212     #if WITH_UMFPACK
00213     void **LU_diag = prepdata->LU_diag;
00214     dvector r0, r1, r2, r3, z0, z1, z2, z3;
00215
00216     r0.row = N0; z0.row = N0;
00217     r1.row = N1; z1.row = N1;
00218     r2.row = N2; z2.row = N2;
00219     r3.row = N3; z3.row = N3;
00220
00221     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);

```

```

00222     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);
00223 #elif WITH_SuperLU
00224     dvector r0, r1, r2, r3, z0, z1, z2, z3;
00225
00226     r0.row = N0; z0.row = N0;
00227     r1.row = N1; z1.row = N1;
00228     r2.row = N2; z2.row = N2;
00229     r3.row = N3; z3.row = N3;
00230
00231     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);
00232     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);
00233 #endif
00234 // Preconditioning A00 block
00235 #if WITH_UMFPACK
00236 /* use UMFPACK direct solver */
00237 fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00238 #elif WITH_SuperLU
00239 /* use SuperLU direct solver */
00240 fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00241 #endif
00242 // Preconditioning A11 block
00243 #if WITH_UMFPACK
00244 /* use UMFPACK direct solver */
00245 fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00246 #elif WITH_SuperLU
00247 /* use SuperLU direct solver */
00248 fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00249 #endif
00250 // Preconditioning A22 block
00251 #if WITH_UMFPACK
00252 /* use UMFPACK direct solver */
00253 fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00254 #elif WITH_SuperLU
00255 /* use SuperLU direct solver */
00256 fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00257 #endif
00258 // Preconditioning A33 block
00259 #if WITH_UMFPACK
00260 /* use UMFPACK direct solver */
00261 fasp_umfpack_solve(&A_diag[3], &r3, &z3, LU_diag[3], 0);
00262 #elif WITH_SuperLU
00263 /* use SuperLU direct solver */
00264 fasp_solver_superlu(&A_diag[3], &r3, &z3, 0);
00265 #endif
00266 // restore r
00267 fasp_darray_cp(N, tempr->val, r);
00268 #endif
00269 }
00270
00271 void fasp_precond_dblc_lower_3 (REAL *r,
00272                                REAL *z,
00273                                void *data)
00274 {
00275     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00276     precondition_data_blc *precd_data = (precondition_data_blc *)data;
00277     dBLCMat *A = precd_data->AbLC;
00278     dCSRmat *A_diag = precd_data->A_diag;
00279     dvector *tempr = &(precd_data->r);
00280
00281     #if WITH_UMFPACK
00282     void **LU_diag = precd_data->LU_diag;
00283     #endif
00284
00285     const INT N0 = A_diag[0].row;
00286     const INT N1 = A_diag[1].row;
00287     const INT N2 = A_diag[2].row;
00288     const INT N = N0 + N1 + N2;
00289
00290     // back up r, setup z;
00291     fasp_darray_cp(N, r, tempr->val);
00292     fasp_darray_set(N, z, 0.0);
00293
00294     // prepare
00295     dvector r0, r1, r2, z0, z1, z2;

```

```

00317
00318     r0.row = N0; z0.row = N0;
00319     r1.row = N1; z1.row = N1;
00320     r2.row = N2; z2.row = N2;
00321
00322     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00323     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00324
00325     // Preconditioning A00 block
00326 #if WITH_UMFPACK
00327     /* use UMFPACK direct solver */
00328     fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00329 #elif WITH_SuperLU
00330     /* use SuperLU direct solver */
00331     fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00332 #endif
00333
00334     // r1 = r1 - A3*z0
00335     fasp_blas_dcsr_aAxy(-1.0, A->blocks[3], z0.val, r1.val);
00336
00337     // Preconditioning A11 block
00338 #if WITH_UMFPACK
00339     /* use UMFPACK direct solver */
00340     fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00341 #elif WITH_SuperLU
00342     /* use SuperLU direct solver */
00343     fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00344 #endif
00345
00346     // r2 = r2 - A6*z0 - A7*z1
00347     fasp_blas_dcsr_aAxy(-1.0, A->blocks[6], z0.val, r2.val);
00348     fasp_blas_dcsr_aAxy(-1.0, A->blocks[7], z1.val, r2.val);
00349
00350     // Preconditioning A22 block
00351 #if WITH_UMFPACK
00352     /* use UMFPACK direct solver */
00353     fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00354 #elif WITH_SuperLU
00355     /* use SuperLU direct solver */
00356     fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00357 #endif
00358
00359     // restore r
00360     fasp_darray_cp(N, tempr->val, r);
00361
00362 #endif
00363 }
00364
00379 void fasp_precond_dblc_lower_3_amg (REAL *r,
00380                                   REAL *z,
00381                                   void *data)
00382 {
00383     precondition_data_blc *precd_data = (precondition_data_blc *)data;
00384     dBLCMat *A = precd_data->AbLC;
00385     dvector *tempr = &(precd_data->r);
00386
00387     AMG_param *amgparam = precd_data->amgparam;
00388     AMG_data **mgl = precd_data->mgl;
00389
00390     const INT N0 = A->blocks[0]->row;
00391     const INT N1 = A->blocks[4]->row;
00392     const INT N2 = A->blocks[8]->row;
00393     const INT N = N0 + N1 + N2;
00394
00395     INT i;
00396
00397     // back up r, setup z;
00398     fasp_darray_cp(N, r, tempr->val);
00399     fasp_darray_set(N, z, 0.0);
00400
00401     // prepare
00402     dvector r0, r1, r2, z0, z1, z2;
00403     r0.row = N0; z0.row = N0; r1.row = N1; z1.row = N1; r2.row = N2; z2.row = N2;
00404     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); z0.val = z;
00405     z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00406
00407     // Preconditioning A00 block
00408     mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
00409     mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);
00410
00411     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);

```

```

00412     fasp_darray_cp(N0, mgl[0]->x.val, z0.val);
00413
00414     // r1 = r1 - A10*z0
00415     fasp_blas_dcsr_aAxy(-1.0, A->blocks[3], z0.val, r1.val);
00416
00417     // Preconditioning A11 block
00418     mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
00419     mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);
00420
00421     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
00422     fasp_darray_cp(N1, mgl[1]->x.val, z1.val);
00423
00424     // r2 = r2 - A20*z0 - A21*z1
00425     fasp_blas_dcsr_aAxy(-1.0, A->blocks[6], z0.val, r2.val);
00426     fasp_blas_dcsr_aAxy(-1.0, A->blocks[7], z1.val, r2.val);
00427
00428     // Preconditioning A22 block
00429     mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
00430     mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);
00431
00432     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[2], amgparam);
00433     fasp_darray_cp(N2, mgl[2]->x.val, z2.val);
00434
00435     // restore r
00436     fasp_darray_cp(N, tempr->val, r);
00437 }
00438
00453 void fasp_precond_dbic_lower_4 (REAL *r,
00454                                REAL *z,
00455                                void *data)
00456 {
00457     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00458
00459     precondition_data_bic *precd_data = (precondition_data_bic *)data;
00460     dBLCMat *A = precd_data->Ablc;
00461     dCSRmat *A_diag = precd_data->A_diag;
00462     dvector *tempr = &(precd_data->r);
00463
00464     #if WITH_UMFPACK
00465         void **LU_diag = precd_data->LU_diag;
00466     #endif
00467
00468     const INT N0 = A_diag[0].row;
00469     const INT N1 = A_diag[1].row;
00470     const INT N2 = A_diag[2].row;
00471     const INT N3 = A_diag[3].row;
00472     const INT N = N0 + N1 + N2 + N3;
00473
00474     // back up r, setup z;
00475     fasp_darray_cp(N, r, tempr->val);
00476     fasp_darray_set(N, z, 0.0);
00477
00478     // prepare
00479     dvector r0, r1, r2, r3, z0, z1, z2, z3;
00480
00481     r0.row = N0; z0.row = N0;
00482     r1.row = N1; z1.row = N1;
00483     r2.row = N2; z2.row = N2;
00484     r3.row = N3; z3.row = N3;
00485
00486     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);
00487     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);
00488
00489     // Preconditioning A00 block
00490     #if WITH_UMFPACK
00491         /* use UMFPACK direct solver */
00492         fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00493     #elif WITH_SuperLU
00494         /* use SuperLU direct solver */
00495         fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00496     #endif
00497
00498     // r1 = r1 - A4*z0
00499     fasp_blas_dcsr_aAxy(-1.0, A->blocks[4], z0.val, r1.val);
00500
00501     // Preconditioning A11 block
00502     #if WITH_UMFPACK
00503         /* use UMFPACK direct solver */
00504         fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00505     #elif WITH_SuperLU
00506         /* use SuperLU direct solver */

```

```

00507     fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00508 #endif
00509
00510     // r2 = r2 - A8*z0 - A9*z1
00511     fasp_blas_dcsr_aApy(-1.0, A->blocks[8], z0.val, r2.val);
00512     fasp_blas_dcsr_aApy(-1.0, A->blocks[9], z1.val, r2.val);
00513
00514     // Preconditioning A22 block
00515     #if WITH_UMFPACK
00516     /* use UMFPACK direct solver */
00517     fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00518     #elif WITH_SuperLU
00519     /* use SuperLU direct solver */
00520     fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00521 #endif
00522
00523     // r3 = r3 - A12*z0 - A13*z1-A14*z2
00524     fasp_blas_dcsr_aApy(-1.0, A->blocks[12], z0.val, r3.val);
00525     fasp_blas_dcsr_aApy(-1.0, A->blocks[13], z1.val, r3.val);
00526     fasp_blas_dcsr_aApy(-1.0, A->blocks[14], z2.val, r3.val);
00527
00528     // Preconditioning A33 block
00529     #if WITH_UMFPACK
00530     /* use UMFPACK direct solver */
00531     fasp_umfpack_solve(&A_diag[3], &r3, &z3, LU_diag[3], 0);
00532     #elif WITH_SuperLU
00533     /* use SuperLU direct solver */
00534     fasp_solver_superlu(&A_diag[3], &r3, &z3, 0);
00535 #endif
00536
00537     // restore r
00538     fasp_darray_cp(N, tempr->val, r);
00539
00540 #endif
00541 }
00542
00557 void fasp_precond_dblc_upper_3 (REAL *r,
00558                                REAL *z,
00559                                void *data)
00560 {
00561     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00562
00563     precondition_data_blc *prepdata = (precondition_data_blc *)data;
00564     dBLCMat *A = prepdata->Abld;
00565     dCSRmat *A_diag = prepdata->A_diag;
00566     dvector *tempr = &(prepdata->r);
00567
00568     #if WITH_UMFPACK
00569     void **LU_diag = prepdata->LU_diag;
00570 #endif
00571
00572     const INT N0 = A_diag[0].row;
00573     const INT N1 = A_diag[1].row;
00574     const INT N2 = A_diag[2].row;
00575     const INT N = N0 + N1 + N2;
00576
00577     // back up r, setup z;
00578     fasp_darray_cp(N, r, tempr->val);
00579     fasp_darray_set(N, z, 0.0);
00580
00581     // prepare
00582     dvector r0, r1, r2, z0, z1, z2;
00583
00584     r0.row = N0; z0.row = N0;
00585     r1.row = N1; z1.row = N1;
00586     r2.row = N2; z2.row = N2;
00587
00588     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00589     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00590
00591     // Preconditioning A22 block
00592     #if WITH_UMFPACK
00593     /* use UMFPACK direct solver */
00594     fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00595     #elif WITH_SuperLU
00596     /* use SuperLU direct solver */
00597     fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00598 #endif
00599
00600     // r1 = r1 - A5*z2
00601     fasp_blas_dcsr_aApy(-1.0, A->blocks[5], z2.val, r1.val);

```



```

00602
00603 // Preconditioning A11 block
00604 #if WITH_UMFPACK
00605 /* use UMFPACK direct solver */
00606 fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00607 #elif WITH_SuperLU
00608 /* use SuperLU direct solver */
00609 fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00610 #endif
00611
00612 // r0 = r0 - A1*z1 - A2*z2
00613 fasp_blas_dcsr_aAxy(-1.0, A->blocks[1], z1.val, r0.val);
00614 fasp_blas_dcsr_aAxy(-1.0, A->blocks[2], z2.val, r0.val);
00615
00616 // Preconditioning A00 block
00617 #if WITH_UMFPACK
00618 /* use UMFPACK direct solver */
00619 fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00620 #elif WITH_SuperLU
00621 /* use SuperLU direct solver */
00622 fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00623 #endif
00624
00625 // restore r
00626 fasp_darray_cp(N, tempr->val, r);
00627
00628 #endif
00629 }
00630
00645 void fasp_precond_dblc_upper_3_amg (REAL *r,
00646                                     REAL *z,
00647                                     void *data)
00648 {
00649     precondition_data_blc *prepdata = (precondition_data_blc *)data;
00650     dBLCMat *A = prepdata->Abld;
00651     dCSRmat *A_diag = prepdata->A_diag;
00652
00653     AMG_param *amgparam = prepdata->amgparam;
00654     AMG_data **mgl = prepdata->mgl;
00655
00656     dvector *tempr = &(prepdata->r);
00657
00658     const INT N0 = A_diag[0].row;
00659     const INT N1 = A_diag[1].row;
00660     const INT N2 = A_diag[2].row;
00661     const INT N = N0 + N1 + N2;
00662
00663     INT i;
00664
00665     // back up r, setup z;
00666     fasp_darray_cp(N, r, tempr->val);
00667     fasp_darray_set(N, z, 0.0);
00668
00669     // prepare
00670     dvector r0, r1, r2, z0, z1, z2;
00671
00672     r0.row = N0; z0.row = N0;
00673     r1.row = N1; z1.row = N1;
00674     r2.row = N2; z2.row = N2;
00675
00676     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00677     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00678
00679     // Preconditioning A22 block
00680     mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
00681     mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);
00682
00683     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[2], amgparam);
00684     fasp_darray_cp(N2, mgl[2]->x.val, z2.val);
00685
00686     // r1 = r1 - A5*z2
00687     fasp_blas_dcsr_aAxy(-1.0, A->blocks[5], z2.val, r1.val);
00688
00689     // Preconditioning A11 block
00690     mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
00691     mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);
00692
00693     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
00694     fasp_darray_cp(N1, mgl[1]->x.val, z1.val);
00695
00696     // r0 = r0 - A1*z1 - A2*z2

```

```

00697     fasp_blas_dcsr_aAxy(-1.0, A->blocks[1], z1.val, r0.val);
00698     fasp_blas_dcsr_aAxy(-1.0, A->blocks[2], z2.val, r0.val);
00699
00700     // Preconditioning A00 block
00701     mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
00702     mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);
00703
00704     for(i=0;i<l;++i) fasp_solver_mgcycle(mgl[0], amgparam);
00705     fasp_darray_cp(N0, mgl[0]->x.val, z0.val);
00706
00707     // restore r
00708     fasp_darray_cp(N, tempr->val, r);
00709 }
00710
00725 void fasp_precond_dblc_SGS_3 (REAL *r,
00726                               REAL *z,
00727                               void *data)
00728 {
00729     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00730
00731     precondition_data_blc *precd_data = (precondition_data_blc *)data;
00732     dBLCMat *A = precd_data->Ablc;
00733     dCSRmat *A_diag = precd_data->A_diag;
00734     dvector *tempr = &(precd_data->r);
00735
00736     #if WITH_UMFPACK
00737         void **LU_diag = precd_data->LU_diag;
00738     #endif
00739
00740     const INT N0 = A_diag[0].row;
00741     const INT N1 = A_diag[1].row;
00742     const INT N2 = A_diag[2].row;
00743     const INT N = N0 + N1 + N2;
00744
00745     // back up r, setup z;
00746     fasp_darray_cp(N, r, tempr->val);
00747     fasp_darray_set(N, z, 0.0);
00748
00749     // prepare
00750     dvector r0, r1, r2, z0, z1, z2;
00751
00752     r0.row = N0; z0.row = N0;
00753     r1.row = N1; z1.row = N1;
00754     r2.row = N2; z2.row = N2;
00755
00756     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00757     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00758
00759     // Preconditioning A00 block
00760     #if WITH_UMFPACK
00761         /* use UMFPACK direct solver */
00762         fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00763     #elif WITH_SuperLU
00764         /* use SuperLU direct solver */
00765         fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00766     #endif
00767
00768     // r1 = r1 - A3*z0
00769     fasp_blas_dcsr_aAxy(-1.0, A->blocks[3], z0.val, r1.val);
00770
00771     // Preconditioning A11 block
00772     #if WITH_UMFPACK
00773         /* use UMFPACK direct solver */
00774         fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00775     #elif WITH_SuperLU
00776         /* use SuperLU direct solver */
00777         fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00778     #endif
00779
00780     // r2 = r2 - A6*z0 - A7*z1
00781     fasp_blas_dcsr_aAxy(-1.0, A->blocks[6], z0.val, r2.val);
00782     fasp_blas_dcsr_aAxy(-1.0, A->blocks[7], z1.val, r2.val);
00783
00784     // Preconditioning A22 block
00785     #if WITH_UMFPACK
00786         /* use UMFPACK direct solver */
00787         fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
00788     #elif WITH_SuperLU
00789         /* use SuperLU direct solver */
00790         fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
00791     #endif

```

```

00792
00793     // r1 = r1 - A5*z2
00794     fasp_blas_dcsr_aAxy(-1.0, A->blocks[5], z2.val, r1.val);
00795
00796     // Preconditioning All block
00797 #if WITH_UMFPACK
00798     /* use UMFPACK direct solver */
00799     fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
00800 #elif WITH_SuperLU
00801     /* use SuperLU direct solver */
00802     fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
00803 #endif
00804
00805     // r0 = r0 - A1*z1 - A2*z2
00806     fasp_blas_dcsr_aAxy(-1.0, A->blocks[1], z1.val, r0.val);
00807     fasp_blas_dcsr_aAxy(-1.0, A->blocks[2], z2.val, r0.val);
00808
00809     // Preconditioning A00 block
00810 #if WITH_UMFPACK
00811     /* use UMFPACK direct solver */
00812     fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
00813 #elif WITH_SuperLU
00814     /* use SuperLU direct solver */
00815     fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
00816 #endif
00817
00818     // restore r
00819     fasp_darray_cp(N, tempr->val, r);
00820
00821 #endif
00822 }
00823
00838 void fasp_precond_dblc_SGS_3_amg (REAL *r,
00839                                   REAL *z,
00840                                   void *data)
00841 {
00842     precondition_data_blc *precd_data = (precondition_data_blc *)data;
00843     dBLCMat *A = precd_data->Ablc;
00844     dCSRmat *A_diag = precd_data->A_diag;
00845
00846     AMG_param *amgparam = precd_data->amgparam;
00847     AMG_data **mgl = precd_data->mgl;
00848
00849     INT i;
00850
00851     dvector *tempr = &(precd_data->r);
00852
00853     const INT N0 = A_diag[0].row;
00854     const INT N1 = A_diag[1].row;
00855     const INT N2 = A_diag[2].row;
00856     const INT N = N0 + N1 + N2;
00857
00858     // back up r, setup z;
00859     fasp_darray_cp(N, r, tempr->val);
00860     fasp_darray_set(N, z, 0.0);
00861
00862     // prepare
00863     dvector r0, r1, r2, z0, z1, z2;
00864
00865     r0.row = N0; z0.row = N0;
00866     r1.row = N1; z1.row = N1;
00867     r2.row = N2; z2.row = N2;
00868
00869     r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
00870     z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
00871
00872     // Preconditioning A00 block
00873     mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
00874     mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);
00875
00876     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);
00877     fasp_darray_cp(N0, mgl[0]->x.val, z0.val);
00878
00879     // r1 = r1 - A3*z0
00880     fasp_blas_dcsr_aAxy(-1.0, A->blocks[3], z0.val, r1.val);
00881
00882     // Preconditioning All block
00883     mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
00884     mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x, 0.0);
00885
00886     for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);

```

```

00887     fasp_darray_cp(N1, mgl[1]->x.val, z1.val);
00888
00889     // r2 = r2 - A6*z0 - A7*z1
00890     fasp_blas_dcsr_aAxy(-1.0, A->blocks[6], z0.val, r2.val);
00891     fasp_blas_dcsr_aAxy(-1.0, A->blocks[7], z1.val, r2.val);
00892
00893     // Preconditioning A22 block
00894     mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
00895     mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x, 0.0);
00896
00897     for(i=0;i<l;++i) fasp_solver_mgcycle(mgl[2], amgparam);
00898     fasp_darray_cp(N2, mgl[2]->x.val, z2.val);
00899
00900     // r1 = r1 - A5*z2
00901     fasp_blas_dcsr_aAxy(-1.0, A->blocks[5], z2.val, r1.val);
00902
00903     // Preconditioning A11 block
00904     mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
00905     mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x, 0.0);
00906
00907     for(i=0;i<l;++i) fasp_solver_mgcycle(mgl[1], amgparam);
00908     fasp_darray_cp(N1, mgl[1]->x.val, z1.val);
00909
00910     // r0 = r0 - A1*z1 - A2*z2
00911     fasp_blas_dcsr_aAxy(-1.0, A->blocks[1], z1.val, r0.val);
00912     fasp_blas_dcsr_aAxy(-1.0, A->blocks[2], z2.val, r0.val);
00913
00914     // Preconditioning A00 block
00915     mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
00916     mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);
00917
00918     for(i=0;i<l;++i) fasp_solver_mgcycle(mgl[0], amgparam);
00919     fasp_darray_cp(N0, mgl[0]->x.val, z0.val);
00920
00921     // restore r
00922     fasp_darray_cp(N, tempr->val, r);
00923 }
00924
00939 void fasp_precond_dblc_sweeping (REAL *r,
00940                                REAL *z,
00941                                void *data)
00942 {
00943     #if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!
00944
00945     precondition_data_sweeping *precd_data = (precondition_data_sweeping *)data;
00946
00947     INT NumLayers = precd_data->NumLayers;
00948     dBLCmat *A = precd_data->A;
00949     dBLCmat *Ai = precd_data->Ai;
00950     dCSRmat *local_A = precd_data->local_A;
00951     ivector *local_index = precd_data->local_index;
00952
00953     #if WITH_UMFPACK
00954         void **local_LU = precd_data->local_LU;
00955     #endif
00956
00957     dvector *r_backup = &(precd_data->r);
00958     REAL *w = precd_data->w;
00959
00960     // local variables
00961     INT i,l;
00962     dvector temp_r;
00963     dvector temp_e;
00964
00965     dvector *local_r = (dvector *)fasp_mem_calloc(NumLayers, sizeof(dvector));
00966     dvector *local_e = (dvector *)fasp_mem_calloc(NumLayers, sizeof(dvector));
00967
00968     // calculate the size and generate block local_r and local_z
00969     INT N=0;
00970
00971     for (l=0;l<NumLayers; l++) {
00972
00973         local_r[l].row = A->blocks[l*NumLayers+1]->row;
00974         local_r[l].val = r+N;
00975
00976         local_e[l].row = A->blocks[l*NumLayers+1]->col;
00977         local_e[l].val = z+N;
00978
00979         N = N+A->blocks[l*NumLayers+1]->col;
00980
00981     }

```

```

00982
00983     temp_r.val = w;
00984     temp_e.val = w+N;
00985
00986     // back up r, setup z;
00987     fasp_darray_cp(N, r, r_backup->val);
00988     fasp_darray_cp(N, r, z);
00989
00990     // L^{-1}r
00991     for (l=0; l<NumLayers-1; l++){
00992
00993         temp_r.row = local_A[l].row;
00994         temp_e.row = local_A[l].row;
00995
00996         fasp_dvec_set(local_A[l].row, &temp_r, 0.0);
00997
00998         for (i=0; i<local_e[l].row; i++){
00999             temp_r.val[local_index[l].val[i]] = local_e[l].val[i];
01000         }
01001
01002 #if WITH_UMFPACK
01003     /* use UMFPACK direct solver */
01004     fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
01005 #elif WITH_SuperLU
01006     /* use SuperLU direct solver */
01007     fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
01008 #endif
01009
01010     for (i=0; i<local_r[l].row; i++){
01011         local_r[l].val[i] = temp_e.val[local_index[l].val[i]];
01012     }
01013
01014     fasp_blas_dcsr_aAxy(-1.0, Ai->blocks[(l+1)*NumLayers+1], local_r[l].val,
01015                        local_e[l+1].val);
01016 }
01017
01018 // D^{-1}L^{-1}r
01019 for (l=0; l<NumLayers; l++){
01020
01021     temp_r.row = local_A[l].row;
01022     temp_e.row = local_A[l].row;
01023
01024     fasp_dvec_set(local_A[l].row, &temp_r, 0.0);
01025
01026     for (i=0; i<local_e[l].row; i++){
01027         temp_r.val[local_index[l].val[i]] = local_e[l].val[i];
01028     }
01029
01030 #if WITH_UMFPACK
01031     /* use UMFPACK direct solver */
01032     fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
01033 #elif WITH_SuperLU
01034     /* use SuperLU direct solver */
01035     fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
01036 #endif
01037
01038     for (i=0; i<local_e[l].row; i++){
01039         local_e[l].val[i] = temp_e.val[local_index[l].val[i]];
01040     }
01041 }
01042
01043 // L^{-t}D^{-1}L^{-1}u
01044 for (l=NumLayers-2; l>=0; l--){
01045
01046     temp_r.row = local_A[l].row;
01047     temp_e.row = local_A[l].row;
01048
01049     fasp_dvec_set(local_A[l].row, &temp_r, 0.0);
01050
01051     fasp_blas_dcsr_mxy (Ai->blocks[l*NumLayers+1+1], local_e[l+1].val, local_r[l].val);
01052
01053     for (i=0; i<local_r[l].row; i++){
01054         temp_r.val[local_index[l].val[i]] = local_r[l].val[i];
01055     }
01056
01057 #if WITH_UMFPACK
01058     /* use UMFPACK direct solver */
01059     fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
01060 #elif WITH_SuperLU

```

```

01063      /* use SuperLU direct solver */
01064      fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
01065 #endif
01066
01067      for (i=0; i<local_e[l].row; i++){
01068          local_e[l].val[i] = local_e[l].val[i] - temp_e.val[local_index[l].val[i]];
01069      }
01070
01071  }
01072
01073      // restore r
01074      fasp_darray_cp(N, r_backup->val, r);
01075
01076 #endif
01077 }
01078
01079 /*-----*/
01080 /*--      End of File      --*/
01081 /*-----*/

```

9.157 PreBSR.c File Reference

Preconditioners for [dBSRmat](#) matrices.

```

#include "fasp.h"
#include "fasp_funcs.h"
#include "PreMGUtil.inl"

```

Functions

- void [fasp_precond_dbsr_diag](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z=inv(D)*r$.*
- void [fasp_precond_dbsr_diag_nc2](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z=inv(D)*r$.*
- void [fasp_precond_dbsr_diag_nc3](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z=inv(D)*r$.*
- void [fasp_precond_dbsr_diag_nc5](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z=inv(D)*r$.*
- void [fasp_precond_dbsr_diag_nc7](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z=inv(D)*r$.*
- void [fasp_precond_dbsr_ilu](#) (REAL *r, REAL *z, void *data)
ILU preconditioner.
- void [fasp_precond_dbsr_ilu_mc_omp](#) (REAL *r, REAL *z, void *data)
Multi-thread Parallel ILU preconditioner based on graph coloring.
- void [fasp_precond_dbsr_ilu_ls_omp](#) (REAL *r, REAL *z, void *data)
Multi-thread Parallel ILU preconditioner based on level schedule strategy.
- void [fasp_precond_dbsr_amg](#) (REAL *r, REAL *z, void *data)
AMG preconditioner.
- void [fasp_precond_dbsr_amg_nk](#) (REAL *r, REAL *z, void *data)
AMG with extra near kernel solve preconditioner.
- void [fasp_precond_dbsr_namli](#) (REAL *r, REAL *z, void *data)
Nonlinear AMLI-cycle AMG preconditioner.

9.157.1 Detailed Description

Preconditioners for [dBSRmat](#) matrices.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxParam.c](#), [AuxThreads.c](#), [AuxVector.c](#), [BlaSmallMat.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), [KrySPcg.c](#), [KrySPvgmres.c](#), [PreMGCycle.c](#), and [PreMGRecurAMLI.c](#)

Copyright (C) 2010–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreBSR.c](#).

9.157.2 Function Documentation

9.157.2.1 fasp_precond_dbsr_amg()

```
void fasp_precond_dbsr_amg (  
    REAL * r,  
    REAL * z,  
    void * data )
```

AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

08/07/2011

Definition at line [986](#) of file [PreBSR.c](#).

9.157.2.2 fasp_precond_dbsr_amg_nk()

```
void fasp_precond_dbsr_amg_nk (  
    REAL * r,  
    REAL * z,  
    void * data )
```

AMG with extra near kernel solve preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 1030 of file [PreBSR.c](#).**9.157.2.3 fasp_precond_dbsr_diag()**

```
void fasp_precond_dbsr_diag (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.**Parameters**

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zhou Zhiyang, Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/24/2012

Note

Works for general nb (Xiaozhe)

Definition at line 49 of file [PreBSR.c](#).**9.157.2.4 fasp_precond_dbsr_diag_nc2()**

```
void fasp_precond_dbsr_diag_nc2 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.**Parameters**

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zhou Zhiyang, Xiaozhe Hu

Date

11/18/2011

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/24/2012

Note

Works for 2-component (Xiaozhe)

Definition at line 121 of file [PreBSR.c](#).

9.157.2.5 fasp_precond_dbsr_diag_nc3()

```
void fasp_precond_dbsr_diag_nc3 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zhou Zhiyang, Xiaozhe Hu

Date

01/06/2011

Modified by Chunsheng Feng Xiaoqiang Yue on 05/24/2012

Note

Works for 3-component (Xiaozhe)

Definition at line 169 of file [PreBSR.c](#).

9.157.2.6 fasp_precond_dbsr_diag_nc5()

```
void fasp_precond_dbsr_diag_nc5 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zhou Zhiyang, Xiaozhe Hu

Date

01/06/2011

Modified by Chunsheng Feng, Xiaoqiang Yue on 05/24/2012

Note

Works for 5-component (Xiaozhe)

Definition at line [217](#) of file [PreBSR.c](#).

9.157.2.7 fasp_precond_dbsr_diag_nc7()

```
void fasp_precond_dbsr_diag_nc7 (
    REAL * r,
    REAL * z,
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zhou Zhiyang, Xiaozhe Hu

Date

01/06/2011

Modified by Chunsheng Feng Xiaoqiang Yue on 05/24/2012

Note

Works for 7-component (Xiaozhe)

Definition at line [265](#) of file [PreBSR.c](#).

9.157.2.8 fasp_precond_dbsr_ilu()

```
void fasp_precond_dbsr_ilu (  
    REAL * r,  
    REAL * z,  
    void * data )
```

ILU preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang, Xiaozhe Hu

Date

11/09/2010

Note

Works for general nb (Xiaozhe)

Definition at line 311 of file [PreBSR.c](#).

9.157.2.9 fasp_precond_dbsr_ilu_ls_omp()

```
void fasp_precond_dbsr_ilu_ls_omp (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Multi-thread Parallel ILU preconditioner based on level schedule strategy.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zheng Li

Date

12/04/2016

Note

Only works for nb 1, 2, and 3 (Zheng)

Definition at line 773 of file [PreBSR.c](#).

9.157.2.10 fasp_precond_dbsr_ilu_mc_omp()

```
void fasp_precond_dbsr_ilu_mc_omp (
    REAL * r,
    REAL * z,
    void * data )
```

Multi-thread Parallel ILU preconditioner based on graph coloring.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Zheng Li

Date

12/04/2016

Note

Only works for nb 1, 2, and 3 (Zheng)

Definition at line 569 of file [PreBSR.c](#).

9.157.2.11 fasp_precond_dbsr_namli()

```
void fasp_precond_dbsr_namli (
    REAL * r,
    REAL * z,
    void * data )
```

Nonlinear AMLI-cycle AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/06/2012

Definition at line 1124 of file PreBSR.c.

9.158 PreBSR.c

[Go to the documentation of this file.](#)

```

00001
00016 #ifdef _OPENMP
00017 #include <omp.h>
00018 #endif
00019
00020 #include "fasp.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*--  Declare Private Functions  --*/
00025 /*-----*/
00026
00027 #include "PreMGUtil.inl"
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00049 void fasp_precond_dbsr_diag (REAL *r,
00050                             REAL *z,
00051                             void *data)
00052 {
00053     precondition_dbsr * diag = (precondition_dbsr *)data;
00054     const INT nb = diag->nb;
00055
00056     switch (nb) {
00057
00058         case 2:
00059             fasp_precond_dbsr_diag_nc2( r, z, diag);
00060             break;
00061         case 3:
00062             fasp_precond_dbsr_diag_nc3( r, z, diag);
00063             break;
00064
00065         case 5:
00066             fasp_precond_dbsr_diag_nc5( r, z, diag);
00067             break;
00068
00069         case 7:
00070             fasp_precond_dbsr_diag_nc7( r, z, diag);
00071             break;
00072
00073         default:
00074         {
00075             REAL *diagptr = diag->diag.val;
00076             const INT nb2 = nb*nb;
00077             const INT m = diag->diag.row/nb2;
00078             INT i;
00079
00080 #ifdef _OPENMP
00081             if (m > OPENMP_HOLDS) {
00082                 INT myid, mybegin, myend;
00083                 INT nthreads = fasp_get_num_threads();
00084                 #pragma omp parallel for private(myid, mybegin, myend, i)
00085                 for (myid = 0; myid < nthreads; myid++) {
00086                     fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00087                     for (i=mybegin; i<myend; ++i) {
00088                         fasp_blas_smat_mnv(&(diagptr[i*nb2]), &(r[i*nb]), &(z[i*nb]), nb);
00089                     }
00090                 }
00091             }
00092             else {
00093 #endif
00094                 for (i = 0; i < m; ++i) {
00095                     fasp_blas_smat_mnv(&(diagptr[i*nb2]), &(r[i*nb]), &(z[i*nb]), nb);
00096                 }
00097 #ifdef _OPENMP
00098             }
00099 #endif

```

```

00100         break;
00101     }
00102 }
00103 }
00104
00121 void fasp_precond_dbsr_diag_nc2 (REAL *r,
00122                                REAL *z,
00123                                void *data)
00124 {
00125     precondition_dbsr * diag = (precondition_dbsr *)data;
00126     REAL * diagptr = diag->diag.val;
00127
00128     INT i;
00129     const INT m = diag->diag.row/4;
00130
00131 #ifdef _OPENMP
00132     if (m > OPENMP_HOLDS) {
00133         INT myid, mybegin, myend;
00134         INT nthreads = fasp_get_num_threads();
00135 #pragma omp parallel for private(myid, mybegin, myend, i)
00136         for (myid = 0; myid < nthreads; myid++) {
00137             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00138             for (i = mybegin; i < myend; ++i) {
00139                 fasp_blas_smat_mnv_nc2(&(diagptr[i*4]), &(r[i*2]), &(z[i*2]));
00140             }
00141         }
00142     }
00143     else {
00144 #endif
00145         for (i = 0; i < m; ++i) {
00146             fasp_blas_smat_mnv_nc2(&(diagptr[i*4]), &(r[i*2]), &(z[i*2]));
00147         }
00148 #ifdef _OPENMP
00149     }
00150 #endif
00151 }
00152
00169 void fasp_precond_dbsr_diag_nc3 (REAL *r,
00170                                REAL *z,
00171                                void *data)
00172 {
00173     precondition_dbsr * diag = (precondition_dbsr *)data;
00174     REAL * diagptr = diag->diag.val;
00175
00176     const INT m = diag->diag.row/9;
00177     INT i;
00178
00179 #ifdef _OPENMP
00180     if (m > OPENMP_HOLDS) {
00181         INT myid, mybegin, myend;
00182         INT nthreads = fasp_get_num_threads();
00183 #pragma omp parallel for private(myid, mybegin, myend, i)
00184         for (myid = 0; myid < nthreads; myid++) {
00185             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00186             for (i = mybegin; i < myend; ++i) {
00187                 fasp_blas_smat_mnv_nc3(&(diagptr[i*9]), &(r[i*3]), &(z[i*3]));
00188             }
00189         }
00190     }
00191     else {
00192 #endif
00193         for (i = 0; i < m; ++i) {
00194             fasp_blas_smat_mnv_nc3(&(diagptr[i*9]), &(r[i*3]), &(z[i*3]));
00195         }
00196 #ifdef _OPENMP
00197     }
00198 #endif
00199 }
00200
00217 void fasp_precond_dbsr_diag_nc5 (REAL *r,
00218                                REAL *z,
00219                                void *data)
00220 {
00221     precondition_dbsr * diag = (precondition_dbsr *)data;
00222     REAL * diagptr = diag->diag.val;
00223
00224     const INT m = diag->diag.row/25;
00225     INT i;
00226
00227 #ifdef _OPENMP
00228     if (m > OPENMP_HOLDS) {

```

```

00229         INT myid, mybegin, myend;
00230         INT nthreads = fasp_get_num_threads();
00231 #pragma omp parallel for private(myid, mybegin, myend, i)
00232         for (myid = 0; myid < nthreads; myid++) {
00233             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00234             for (i = mybegin; i < myend; ++i) {
00235                 fasp_blas_smat_mxv_nc5(&(diagptr[i*25]), &(r[i*5]), &(z[i*5]));
00236             }
00237         }
00238     }
00239     else {
00240 #endif
00241         for (i = 0; i < m; ++i) {
00242             fasp_blas_smat_mxv_nc5(&(diagptr[i*25]), &(r[i*5]), &(z[i*5]));
00243         }
00244 #ifdef _OPENMP
00245     }
00246 #endif
00247 }
00248
00265 void fasp_precond_dbsr_diag_nc7 (REAL *r,
00266                                 REAL *z,
00267                                 void *data)
00268 {
00269     precondition_dbsr * diag = (precondition_dbsr *)data;
00270     REAL * diagptr = diag->diag.val;
00271
00272     const INT m = diag->diag.row/49;
00273     INT i;
00274
00275 #ifdef _OPENMP
00276     if (m > OPENMP_HOLDS) {
00277         INT myid, mybegin, myend;
00278         INT nthreads = fasp_get_num_threads();
00279 #pragma omp parallel for private(myid, mybegin, myend, i)
00280         for (myid = 0; myid < nthreads; myid++) {
00281             fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
00282             for (i = mybegin; i < myend; ++i) {
00283                 fasp_blas_smat_mxv_nc7(&(diagptr[i*49]), &(r[i*7]), &(z[i*7]));
00284             }
00285         }
00286     }
00287     else {
00288 #endif
00289         for (i = 0; i < m; ++i) {
00290             fasp_blas_smat_mxv_nc7(&(diagptr[i*49]), &(r[i*7]), &(z[i*7]));
00291         }
00292 #ifdef _OPENMP
00293     }
00294 #endif
00295 }
00296
00311 void fasp_precond_dbsr_ilu (REAL *r,
00312                             REAL *z,
00313                             void *data)
00314 {
00315     const ILU_data *iludata=(ILU_data *)data;
00316     const INT m=iludata->row, mm1=m-1, mm2=m-2, memneed=2*m;
00317     const INT nb=iludata->nb, nb2=nb*nb, size=m*nb;
00318
00319     INT *ijlu=iludata->ijlu;
00320     REAL *lu=iludata->luval;
00321
00322     INT ib, ibstart, ibstart1;
00323     INT i, j, jj, begin_row, end_row;
00324     REAL *zz, *zr, *mult;
00325
00326     if (iludata->nwork<memneed) {
00327         printf("### ERROR: Need %d memory, only %d available!\n",
00328             memneed, iludata->nwork);
00329         fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00330     }
00331
00332     zz = iludata->work;
00333     zr = zz + size;
00334     mult = zr + size;
00335
00336     memcpy(zr, r, size*sizeof(REAL));
00337
00338     switch (nb) {
00339

```

```

00340     case 1:
00341
00342         // forward sweep: solve unit lower matrix equation L*zz=zr
00343         zz[0]=zr[0];
00344         for (i=1;i<=mm1;++i) {
00345             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00346             for (j=begin_row;j<=end_row;++j) {
00347                 jj=ijlu[j];
00348                 if (jj<i) zr[i]-=lu[j]*zz[jj];
00349                 else break;
00350             }
00351             zz[i]=zr[i];
00352         }
00353
00354         // backward sweep: solve upper matrix equation U*z=zz
00355         z[mm1]=zz[mm1]*lu[mm1];
00356         for (i=mm2;i>=0;i--) {
00357             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00358             for (j=end_row;j>=begin_row;j--) {
00359                 jj=ijlu[j];
00360                 if (jj>i) zz[i]-=lu[j]*z[jj];
00361                 else break;
00362             }
00363             z[i]=zz[i]*lu[i];
00364         }
00365
00366         break; //end (if nb==1)
00367
00368     case 3:
00369
00370         // forward sweep: solve unit lower matrix equation L*zz=zr
00371         zz[0] = zr[0];
00372         zz[1] = zr[1];
00373         zz[2] = zr[2];
00374
00375         for (i=1;i<=mm1;++i) {
00376             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00377             ibstart=i*nb;
00378             for (j=begin_row;j<=end_row;++j) {
00379                 jj=ijlu[j];
00380                 if (jj<i)
00381                 {
00382                     fasp_blas_smat_m xv_nc3(&(lu[j*nb2]),&(zz[jj*nb]),mult);
00383                     for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
00384                 }
00385                 else break;
00386             }
00387
00388             zz[ibstart] = zr[ibstart];
00389             zz[ibstart+1] = zr[ibstart+1];
00390             zz[ibstart+2] = zr[ibstart+2];
00391         }
00392
00393         // backward sweep: solve upper matrix equation U*z=zz
00394         ibstart=mm1*nb2;
00395         ibstart1=mm1*nb;
00396         fasp_blas_smat_m xv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));
00397
00398         for (i=mm2;i>=0;i--) {
00399             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00400             ibstart=i*nb2;
00401             ibstart1=i*nb;
00402             for (j=end_row;j>=begin_row;j--) {
00403                 jj=ijlu[j];
00404                 if (jj>i) {
00405                     fasp_blas_smat_m xv_nc3(&(lu[j*nb2]),&(z[jj*nb]),mult);
00406                     for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
00407                 }
00408                 else break;
00409             }
00410         }
00411
00412         fasp_blas_smat_m xv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));
00413     }
00414
00415     break; // end (if nb=3)
00416
00417 case 5:
00418
00419     // forward sweep: solve unit lower matrix equation L*zz=zr
00420

```



```

00421     fasp_darray_cp(nb, &(zr[0]), &(zz[0]));
00422
00423     for (i=1; i<=mml; ++i) {
00424         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00425         ibstart=i*nb;
00426         for (j=begin_row; j<=end_row; ++j) {
00427             jj=ijlu[j];
00428             if (jj<i) {
00429                 fasp_blas_smat_mxv_nc5(&(lu[j*nb2]), &(zz[jj*nb]), mult);
00430                 for (ib=0; ib<nb; ++ib) zr[ibstart+ib]-=mult[ib];
00431             }
00432             else break;
00433         }
00434
00435         fasp_darray_cp(nb, &(zr[ibstart]), &(zz[ibstart]));
00436     }
00437
00438     // backward sweep: solve upper matrix equation U*z=zz
00439     ibstart=mml*nb2;
00440     ibstart1=mml*nb;
00441     fasp_blas_smat_mxv_nc5(&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00442
00443     for (i=mm2; i>=0; i--) {
00444         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00445         ibstart=i*nb2;
00446         ibstart1=i*nb;
00447         for (j=end_row; j>=begin_row; j--) {
00448             jj=ijlu[j];
00449             if (jj>i) {
00450                 fasp_blas_smat_mxv_nc5(&(lu[j*nb2]), &(z[jj*nb]), mult);
00451                 for (ib=0; ib<nb; ++ib) zz[ibstart1+ib]-=mult[ib];
00452             }
00453             else break;
00454         }
00455
00456         fasp_blas_smat_mxv_nc5(&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00457     }
00458
00459     }
00460
00461     break; //end (if nb==5)
00462
00463     case 7:
00464
00465         // forward sweep: solve unit lower matrix equation L*zz=zr
00466         fasp_darray_cp(nb, &(zr[0]), &(zz[0]));
00467
00468         for (i=1; i<=mml; ++i) {
00469             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00470             ibstart=i*nb;
00471             for (j=begin_row; j<=end_row; ++j) {
00472                 jj=ijlu[j];
00473                 if (jj<i) {
00474                     fasp_blas_smat_mxv_nc7(&(lu[j*nb2]), &(zz[jj*nb]), mult);
00475                     for (ib=0; ib<nb; ++ib) zr[ibstart+ib]-=mult[ib];
00476                 }
00477                 else break;
00478             }
00479
00480             fasp_darray_cp(nb, &(zr[ibstart]), &(zz[ibstart]));
00481         }
00482
00483         // backward sweep: solve upper matrix equation U*z=zz
00484         ibstart=mml*nb2;
00485         ibstart1=mml*nb;
00486         fasp_blas_smat_mxv_nc7(&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00487
00488         for (i=mm2; i>=0; i--) {
00489             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00490             ibstart=i*nb2;
00491             ibstart1=i*nb;
00492             for (j=end_row; j>=begin_row; j--) {
00493                 jj=ijlu[j];
00494                 if (jj>i) {
00495                     fasp_blas_smat_mxv_nc7(&(lu[j*nb2]), &(z[jj*nb]), mult);
00496                     for (ib=0; ib<nb; ++ib) zz[ibstart1+ib]-=mult[ib];
00497                 }
00498                 else break;
00499             }
00500         }
00501

```

```

00502         fasp_blas_smat_mnv_nc7(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));
00503     }
00504
00505     break; //end (if nb==7)
00506
00507     default:
00508
00509         // forward sweep: solve unit lower matrix equation L*zz=zr
00510         fasp_darray_cp(nb,&(zr[0]),&(zz[0]));
00511
00512         for (i=1;i<=mm1;++i) {
00513             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00514             ibstart=i*nb;
00515             for (j=begin_row;j<=end_row;++j) {
00516                 jj=ijlu[j];
00517                 if (jj<i) {
00518                     fasp_blas_smat_mnv(&(lu[j*nb2]),&(zz[jj*nb]),mult,nb);
00519                     for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
00520                 }
00521                 else break;
00522             }
00523
00524             fasp_darray_cp(nb,&(zr[ibstart]),&(zz[ibstart]));
00525         }
00526
00527         // backward sweep: solve upper matrix equation U*z=zz
00528         ibstart=mm1*nb2;
00529         ibstart1=mm1*nb;
00530         fasp_blas_smat_mnv(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]),nb);
00531
00532         for (i=mm2;i>=0;i--) {
00533             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00534             ibstart=i*nb2;
00535             ibstart1=i*nb;
00536             for (j=end_row;j>=begin_row;j--) {
00537                 jj=ijlu[j];
00538                 if (jj>i) {
00539                     fasp_blas_smat_mnv(&(lu[j*nb2]),&(z[jj*nb]),mult,nb);
00540                     for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
00541                 }
00542                 else break;
00543             }
00544
00545             fasp_blas_smat_mnv(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]),nb);
00546         }
00547
00548         break; // end everything else
00549     }
00550 }
00551
00552 return;
00553 }
00554
00569 void fasp_precond_dbsr_ilu_mc_omp (REAL *r,
00570                                     REAL *z,
00571                                     void *data)
00572 {
00573     #ifdef _OPENMP
00574         const ILU_data *iludata=(ILU_data *)data;
00575         const INT m=iludata->row, memneed=2*m;
00576         const INT nb=iludata->nb, nb2=nb*nb, size=m*nb;
00577
00578         INT *ijlu=iludata->ijlu;
00579         REAL *lu=iludata->luval;
00580         INT ncolors = iludata->nlevL;
00581         INT *ic = iludata->ilevL;
00582
00583         INT ib, ibstart,ibstart1;
00584         INT i, j, jj, k, begin_row, end_row;
00585         REAL *zz, *zr, *mult;
00586
00587         if (iludata->nwork<memneed) {
00588             printf("### ERROR: Need %d memory, only %d available!\n",
00589                 memneed, iludata->nwork);
00590             fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00591         }
00592
00593         zz = iludata->work;
00594         zr = zz + size;
00595
00596         memcpy(zr, r, size*sizeof(REAL));

```

```

00597
00598     switch (nb) {
00599
00600         case 1:
00601             // forward sweep: solve unit lower matrix equation L*zz=zr
00602             for (k=0; k<ncolors; ++k) {
00603 #pragma omp parallel for private(i,begin_row,end_row,j,jj)
00604                 for (i=ic[k]; i<ic[k+1]; ++i) {
00605                     begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00606                     for (j=begin_row; j<=end_row; ++j) {
00607                         jj=ijlu[j];
00608                         if (jj<i) zr[i]-=lu[j]*zz[jj];
00609                         else break;
00610                     }
00611                     zz[i]=zr[i];
00612                 }
00613             }
00614             // backward sweep: solve upper matrix equation U*z=zz
00615             for (k=ncolors-1; k>=0; k--) {
00616 #pragma omp parallel for private(i,begin_row,end_row,j,jj)
00617                 for (i=ic[k+1]-1; i>=ic[k]; i--) {
00618                     begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00619                     for (j=end_row; j>=begin_row; j--) {
00620                         jj=ijlu[j];
00621                         if (jj>i) zz[i]-=lu[j]*z[jj];
00622                         else break;
00623                     }
00624                     z[i]=zz[i]*lu[i];
00625                 }
00626             }
00627             break; //end (if nb==1)
00628
00629         case 2:
00630
00631             for (k=0; k<ncolors; ++k) {
00632 #pragma omp parallel private(i,begin_row,end_row,ibstart,j,jj,ib,mult)
00633                 {
00634                     mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
00635 #pragma omp for
00636                     for (i=ic[k]; i<ic[k+1]; ++i) {
00637                         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00638                         ibstart=i*nb;
00639                         for (j=begin_row; j<=end_row; ++j) {
00640                             jj=ijlu[j];
00641                             if (jj<i)
00642                             {
00643                                 fasp_blas_smat_m xv_nc2 (&(lu[j*nb2]), &(zz[jj*nb]), mult);
00644                                 for (ib=0; ib<nb; ++ib) zr[ibstart+ib]-=mult[ib];
00645                             }
00646                             else break;
00647                         }
00648                     }
00649                     zz[ibstart] = zr[ibstart];
00650                     zz[ibstart+1] = zr[ibstart+1];
00651                 }
00652             }
00653             fasp_mem_free(mult); mult = NULL;
00654
00655         }
00656     }
00657
00658     for (k=ncolors-1; k>=0; k--) {
00659 #pragma omp parallel private(i,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
00660         {
00661             mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
00662 #pragma omp for
00663             for (i=ic[k+1]-1; i>=ic[k]; i--) {
00664                 begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00665                 ibstart=i*nb2;
00666                 ibstart1=i*nb;
00667                 for (j=end_row; j>=begin_row; j--) {
00668                     jj=ijlu[j];
00669                     if (jj>i) {
00670                         fasp_blas_smat_m xv_nc2 (&(lu[j*nb2]), &(z[jj*nb]), mult);
00671                         for (ib=0; ib<nb; ++ib) zz[ibstart1+ib]-=mult[ib];
00672                     }
00673                     else break;
00674                 }
00675             }
00676             fasp_blas_smat_m xv_nc2 (&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00677

```

```

00678
00679         }
00680
00681         fasp_mem_free(mult); mult = NULL;
00682     }
00683 }
00684
00685     break; // end (if nb=2)
00686 case 3:
00687
00688     for (k=0; k<ncolors; ++k) {
00689 #pragma omp parallel private(i,begin_row,end_row,ibstart,j,jj,ib,mult)
00690     {
00691         mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
00692 #pragma omp for
00693         for (i=ic[k]; i<ic[k+1]; ++i) {
00694             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00695             ibstart=i*nb;
00696             for (j=begin_row; j<=end_row; ++j) {
00697                 jj=ijlu[j];
00698                 if (jj<i)
00699                 {
00700                     fasp_blas_smat_m xv_nc3(&(lu[j*nb2]), &(zz[jj*nb]), mult);
00701                     for (ib=0; ib<nb; ++ib) zr[ibstart+ib] -= mult[ib];
00702                 }
00703                 else break;
00704             }
00705
00706             zz[ibstart] = zr[ibstart];
00707             zz[ibstart+1] = zr[ibstart+1];
00708             zz[ibstart+2] = zr[ibstart+2];
00709         }
00710
00711         fasp_mem_free(mult); mult = NULL;
00712     }
00713 }
00714
00715     for (k=ncolors-1; k>=0; k--) {
00716 #pragma omp parallel private(i,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
00717     {
00718         mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
00719 #pragma omp for
00720         for (i=ic[k+1]-1; i>=ic[k]; i--) {
00721             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00722             ibstart=i*nb2;
00723             ibstart1=i*nb;
00724             for (j=end_row; j>=begin_row; j--) {
00725                 jj=ijlu[j];
00726                 if (jj>i) {
00727                     fasp_blas_smat_m xv_nc3(&(lu[j*nb2]), &(z[jj*nb]), mult);
00728                     for (ib=0; ib<nb; ++ib) zz[ibstart1+ib] -= mult[ib];
00729                 }
00730                 else break;
00731             }
00732
00733             fasp_blas_smat_m xv_nc3(&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00734
00735         }
00736
00737         fasp_mem_free(mult); mult = NULL;
00738     }
00739 }
00740
00741     break; // end (if nb=3)
00742
00743 default:
00744 {
00745     if (nb > 3) {
00746         printf("### ERROR: Multi-thread Parallel ILU for %d components \
00747 has not yet been implemented!!!", nb);
00748         fasp_chkerr(ERROR_UNKNOWN, __FUNCTION__);
00749     }
00750     break;
00751 }
00752 }
00753 }
00754
00755     return;
00756 #endif
00757 }
00758

```

```

00773 void fasp_precond_dbsr_ilu_ls_omp (REAL *r,
00774                                     REAL *z,
00775                                     void *data)
00776 {
00777     #ifdef _OPENMP
00778         const ILU_data *iludata=(ILU_data *)data;
00779         const INT      m=iludata->row, memneed=2*m;
00780         const INT      nb=iludata->nb, nb2=nb*nb, size=m*nb;
00781
00782         INT            *ijlu=iludata->ijlu;
00783         REAL            *lu=iludata->luval;
00784         INT            nlevL = iludata->nlevL;
00785         INT            ilevL = iludata->ilevL;
00786         INT            jlevL = iludata->jlevL;
00787         INT            nlevU = iludata->nlevU;
00788         INT            ilevU = iludata->ilevU;
00789         INT            jlevU = iludata->jlevU;
00790
00791         INT            ib, ibstart, ibstart1;
00792         INT            i, ii, j, jj, k, begin_row, end_row;
00793         REAL            *zz, *zr, *mult;
00794
00795         if (iludata->nwork<memneed) {
00796             printf("### ERROR: Need %d memory, only %d available!\n",
00797                   memneed, iludata->nwork);
00798             fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00799         }
00800
00801         zz = iludata->work;
00802         zr = zz + size;
00803         //mult = zr + size;
00804
00805         memcpy(zr, r, size*sizeof(REAL));
00806
00807         switch (nb) {
00808
00809             case 1:
00810                 // forward sweep: solve unit lower matrix equation L*zz=zr
00811                 for (k=0; k<nlevL; ++k) {
00812                     #pragma omp parallel for private(i,ii,begin_row,end_row,j,jj)
00813                     for (ii=ilevL[k]; ii<ilevL[k+1]; ++ii) {
00814                         i = jlevL[ii];
00815                         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00816                         for (j=begin_row; j<=end_row; ++j) {
00817                             jj=ijlu[j];
00818                             if (jj<i) zr[i]-=lu[j]*zz[jj];
00819                             else break;
00820                         }
00821                         zz[i]=zr[i];
00822                     }
00823                 }
00824                 // backward sweep: solve upper matrix equation U*z=zz
00825                 for (k=0; k<nlevU; k++) {
00826                     #pragma omp parallel for private(i,ii,begin_row,end_row,j,jj)
00827                     for (ii=ilevU[k+1]-1; ii>=ilevU[k]; ii--) {
00828                         i = jlevU[ii];
00829                         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00830                         for (j=end_row; j>=begin_row; j--) {
00831                             jj=ijlu[j];
00832                             if (jj>i) zz[i]-=lu[j]*z[jj];
00833                             else break;
00834                         }
00835                         z[i]=zz[i]*lu[i];
00836                     }
00837                 }
00838
00839                 break; //end (if nb==1)
00840
00841             case 2:
00842
00843                 for (k=0; k<nlevL; ++k) {
00844                     #pragma omp parallel private(i,ii,begin_row,end_row,ibstart,j,jj,ib,mult)
00845                     {
00846                         mult = (REAL*)fasp_mem_calloc(nb, sizeof(REAL));
00847                     #pragma omp for
00848                     for (ii=ilevL[k]; ii<ilevL[k+1]; ++ii) {
00849                         i = jlevL[ii];
00850                         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00851                         ibstart=i*nb;
00852                         for (j=begin_row; j<=end_row; ++j) {
00853                             jj=ijlu[j];

```

```

00854             if (jj<i)
00855             {
00856                 fasp_blas_smat_m xv_nc2 (&(lu[j*nb2]), &(zz[jj*nb]), mult);
00857                 for (ib=0; ib<nb; ++ib) zr[ibstart+ib] -= mult[ib];
00858             }
00859             else break;
00860         }
00861
00862         zz[ibstart] = zr[ibstart];
00863         zz[ibstart+1] = zr[ibstart+1];
00864     }
00865
00866     fasp_mem_free(mult); mult = NULL;
00867 }
00868 }
00869
00870     for (k=0; k<nlevU; k++) {
00871 #pragma omp parallel private(i, ii, begin_row, end_row, ibstart, ibstart1, j, jj, ib, mult)
00872     {
00873         mult = (REAL*) fasp_mem_calloc(nb, sizeof(REAL));
00874 #pragma omp for
00875         for (ii=ilevU[k+1]-1; ii>=ilevU[k]; ii--) {
00876             i = jlevU[ii];
00877             begin_row=jlu[i]; end_row=jlu[i+1]-1;
00878             ibstart=i*nb2;
00879             ibstart1=i*nb;
00880             for (j=end_row; j>=begin_row; j--) {
00881                 jj=jlu[j];
00882                 if (jj>i) {
00883                     fasp_blas_smat_m xv_nc2 (&(lu[j*nb2]), &(zz[jj*nb]), mult);
00884                     for (ib=0; ib<nb; ++ib) zz[ibstart1+ib] -= mult[ib];
00885                 }
00886             }
00887             else break;
00888         }
00889
00890         fasp_blas_smat_m xv_nc2 (&(lu[ibstart]), &(zz[ibstart1]), &(z[ibstart1]));
00891     }
00892 }
00893
00894     fasp_mem_free(mult); mult = NULL;
00895 }
00896 }
00897
00898     break; // end (if nb=2)
00899 case 3:
00900
00901     for (k=0; k<nlevL; ++k) {
00902 #pragma omp parallel private(i, ii, begin_row, end_row, ibstart, j, jj, ib, mult)
00903     {
00904         mult = (REAL*) fasp_mem_calloc(nb, sizeof(REAL));
00905 #pragma omp for
00906         for (ii=ilevL[k]; ii<ilevL[k+1]; ++ii) {
00907             i = jlevL[ii];
00908             begin_row=jlu[i]; end_row=jlu[i+1]-1;
00909             ibstart=i*nb;
00910             for (j=begin_row; j<=end_row; ++j) {
00911                 jj=jlu[j];
00912                 if (jj<i)
00913                 {
00914                     fasp_blas_smat_m xv_nc3 (&(lu[j*nb2]), &(zz[jj*nb]), mult);
00915                     for (ib=0; ib<nb; ++ib) zr[ibstart+ib] -= mult[ib];
00916                 }
00917                 else break;
00918             }
00919
00920             zz[ibstart] = zr[ibstart];
00921             zz[ibstart+1] = zr[ibstart+1];
00922             zz[ibstart+2] = zr[ibstart+2];
00923         }
00924
00925         fasp_mem_free(mult); mult = NULL;
00926     }
00927 }
00928
00929     for (k=0; k<nlevU; k++) {
00930 #pragma omp parallel private(i, ii, begin_row, end_row, ibstart, ibstart1, j, jj, ib, mult)
00931     {
00932         mult = (REAL*) fasp_mem_calloc(nb, sizeof(REAL));
00933 #pragma omp for
00934         for (ii=ilevU[k+1]-1; ii>=ilevU[k]; ii--) {

```

```

00935         i = jlevU[ii];
00936         begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00937         ibstart=i*nb2;
00938         ibstartl=i*nb;
00939         for (j=end_row; j>=begin_row; j--) {
00940             jj=ijlu[j];
00941             if (jj>1) {
00942                 fasp_blas_smat_mxv_nc3(&(lu[j*nb2]), &(z[jj*nb]), mult);
00943                 for (ib=0; ib<nb; ++ib) zz[ibstartl+ib]-=mult[ib];
00944             }
00945             else break;
00946         }
00947     }
00948
00949     fasp_blas_smat_mxv_nc3(&(lu[ibstart]), &(zz[ibstartl]), &(z[ibstartl]));
00950
00951 }
00952
00953     fasp_mem_free(mult); mult = NULL;
00954 }
00955 }
00956
00957     break; // end (if nb=3)
00958
00959     default:
00960     {
00961         if (nb > 3) {
00962             printf("### ERROR: Multi-thread Parallel ILU for %d components \
00963 has not yet been implemented!!!", nb);
00964             fasp_chkerr(ERROR_UNKNOWN, __FUNCTION__);
00965         }
00966         break;
00967     }
00968 }
00969
00970     return;
00971 #endif
00972 }
00973
00986 void fasp_precond_dbsr_amg (REAL *r,
00987                             REAL *z,
00988                             void *data)
00989 {
00990     precondition_data_bsr *predata=(precondition_data_bsr *)data;
00991     const INT row=predata->mgl_data[0].A.ROW;
00992     const INT nb = predata->mgl_data[0].A.nb;
00993     const INT maxit=predata->maxit;
00994     const INT m = row*nb;
00995
00996     INT i;
00997
00998     AMG_param amgparam; fasp_param_amg_init(&amgparam);
00999     amgparam.cycle_type = predata->cycle_type;
01000     amgparam.smoother = predata->smoother;
01001     amgparam.smooth_order = predata->smooth_order;
01002     amgparam.presmooth_iter = predata->presmooth_iter;
01003     amgparam.postsmooth_iter = predata->postsmooth_iter;
01004     amgparam.relaxation = predata->relaxation;
01005     amgparam.coarse_scaling = predata->coarse_scaling;
01006     amgparam.tentative_smooth = predata->tentative_smooth;
01007     amgparam.ILU_levels = predata->mgl_data->ILU_levels;
01008
01009     AMG_data_bsr *mgl = predata->mgl_data;
01010     mgl->b.row=m; fasp_darray_cp(m, r, mgl->b.val); // residual is an input
01011     mgl->x.row=m; fasp_dvec_set(m, &mgl->x, 0.0);
01012
01013     for ( i=maxit; i--; ) fasp_solver_mgcycle_bsr(mgl, &amgparam);
01014
01015     fasp_darray_cp(m, mgl->x.val, z);
01016 }
01017
01030 void fasp_precond_dbsr_amg_nk (REAL *r,
01031                                REAL *z,
01032                                void *data)
01033 {
01034     precondition_data_bsr *predata=(precondition_data_bsr *)data;
01035     const INT row=predata->mgl_data[0].A.ROW;
01036     const INT nb = predata->mgl_data[0].A.nb;
01037     const INT maxit=predata->maxit;
01038     const INT m = row*nb;
01039

```

```

01040     INT i;
01041
01042     dCSRmat *A_nk = predata->A_nk;
01043     dCSRmat *P_nk = predata->P_nk;
01044     dCSRmat *R_nk = predata->R_nk;
01045
01046     fasp_darray_set(m, z, 0.0);
01047
01048     // local variables
01049     dvector r_nk, z_nk;
01050     fasp_dvec_alloc(A_nk->row, &r_nk);
01051     fasp_dvec_alloc(A_nk->row, &z_nk);
01052
01053     //-----
01054     // extra kernel solve
01055     //-----
01056     // r_nk = R_nk*r
01057     fasp_blas_dcsr_mxv(R_nk, r, r_nk.val);
01058
01059     // z_nk = A_nk^{-1}*r_nk
01060 #if WITH_UMFPACK // use UMFPACK directly
01061     fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
01062 #else
01063     fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
01064 #endif
01065
01066     // z = z + P_nk*z_nk;
01067     fasp_blas_dcsr_aAxy(1.0, P_nk, z_nk.val, z);
01068
01069     //-----
01070     // AMG solve
01071     //-----
01072     AMG_param amgparam; fasp_param_amg_init(&amgparam);
01073     amgparam.cycle_type = predata->cycle_type;
01074     amgparam.smoother = predata->smoother;
01075     amgparam.smooth_order = predata->smooth_order;
01076     amgparam.presmooth_iter = predata->presmooth_iter;
01077     amgparam.postsmooth_iter = predata->postsmooth_iter;
01078     amgparam.relaxation = predata->relaxation;
01079     amgparam.coarse_scaling = predata->coarse_scaling;
01080     amgparam.tentative_smooth = predata->tentative_smooth;
01081     amgparam.ILU_levels = predata->mgl_data->ILU_levels;
01082
01083     AMG_data_bsr *mgl = predata->mgl_data;
01084     mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
01085     mgl->x.row=m; //fasp_dvec_set(m,&mgl->x,0.0);
01086     fasp_darray_cp(m, z, mgl->x.val);
01087
01088     for ( i=maxit; i--; ) fasp_solver_mgcycle_bsr(mgl,&amgparam);
01089
01090     fasp_darray_cp(m,mgl->x.val,z);
01091
01092     //-----
01093     // extra kernel solve
01094     //-----
01095     // r = r - A*z
01096     fasp_blas_dbsr_aAxy(-1.0, &(predata->mgl_data[0].A), z, mgl->b.val);
01097
01098     // r_nk = R_nk*r
01099     fasp_blas_dcsr_mxv(R_nk, mgl->b.val, r_nk.val);
01100
01101     // z_nk = A_nk^{-1}*r_nk
01102 #if WITH_UMFPACK // use UMFPACK directly
01103     fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
01104 #else
01105     fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
01106 #endif
01107
01108     // z = z + P_nk*z_nk;
01109     fasp_blas_dcsr_aAxy(1.0, P_nk, z_nk.val, z);
01110 }
01111
01124 void fasp_precond_dbsr_namli (REAL *r,
01125                             REAL *z,
01126                             void *data)
01127 {
01128     precondition_data_bsr *pcdata=(precondition_data_bsr *)data;
01129     const INT row=pcdata->mgl_data[0].A.ROW;
01130     const INT nb=pcdata->mgl_data[0].A.nb;
01131     const INT maxit=pcdata->maxit;
01132     const SHORT num_levels=pcdata->max_levels;

```



```

01133     const INT m=row*nb;
01134
01135     INT i;
01136
01137     AMG_param amgparam;
01138     fasp_param_amg_init(&amgparam);
01139     fasp_param_precbsr_to_amg(&amgparam,pdata);
01140
01141     AMG_data_bsr *mgl = pdata->mgl_data;
01142     mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
01143     mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);
01144
01145     for ( i=maxit; i--; ) fasp_solver_namli_bsr(mgl,&amgparam,0, num_levels);
01146
01147     fasp_darray_cp(m,mgl->x.val,z);
01148 }
01149
01150 /*-----*/
01151 /*--      End of File      --*/
01152 /*-----*/

```

9.159 PreCSR.c File Reference

Preconditioners for dCSRmat matrices.

```

#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"

```

Functions

- `fasp_precond_setup` (const `SHORT` `precond_type`, `AMG_param` *`amgparam`, `ILU_param` *`iluparam`, `dCSRmat` *`A`)
Setup preconditioner interface for iterative methods.
- void `fasp_precond_diag` (`REAL` *`r`, `REAL` *`z`, void *`data`)
Diagonal preconditioner $z = \text{inv}(D)r$.
- void `fasp_precond_ilu` (`REAL` *`r`, `REAL` *`z`, void *`data`)
ILU preconditioner.
- void `fasp_precond_ilu_forward` (`REAL` *`r`, `REAL` *`z`, void *`data`)
ILU preconditioner: only forward sweep.
- void `fasp_precond_ilu_backward` (`REAL` *`r`, `REAL` *`z`, void *`data`)
ILU preconditioner: only backward sweep.
- void `fasp_precond_swz` (`REAL` *`r`, `REAL` *`z`, void *`data`)
get z from r by Schwarz
- void `fasp_precond_amg` (`REAL` *`r`, `REAL` *`z`, void *`data`)
AMG preconditioner.
- void `fasp_precond_famg` (`REAL` *`r`, `REAL` *`z`, void *`data`)
Full AMG preconditioner.
- void `fasp_precond_amli` (`REAL` *`r`, `REAL` *`z`, void *`data`)
AMLI AMG preconditioner.
- void `fasp_precond_namli` (`REAL` *`r`, `REAL` *`z`, void *`data`)
Nonlinear AMLI AMG preconditioner.
- void `fasp_precond_amg_nk` (`REAL` *`r`, `REAL` *`z`, void *`data`)
AMG with extra near kernel solve as preconditioner.

9.159.1 Detailed Description

Preconditioners for [dCSRmat](#) matrices.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxParam.c](#), [AuxVector.c](#), [BlalLUSetupCSR.c](#), [BlaSchwarzSetup.c](#), [BlaSparseCSR.c](#), [BlaSpmvCSR.c](#), [KrySPcg.c](#), [KrySPvgmres.c](#), [PreAMGSetupRS.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreDataInit.c](#), [PreMGCycle.c](#), [PreMGCycleFull.c](#), and [PreMGRecurAMLI.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreCSR.c](#).

9.159.2 Function Documentation

9.159.2.1 fasp_precond_amg()

```
void fasp_precond_amg (
    REAL * r,
    REAL * z,
    void * data )
```

AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Chensong Zhang

Date

04/06/2010

Definition at line [416](#) of file [PreCSR.c](#).

9.159.2.2 fasp_precond_amg_nk()

```
void fasp_precond_amg_nk (
    REAL * r,
    REAL * z,
    void * data )
```

AMG with extra near kernel solve as preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
----------	---

Parameters

<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 548 of file [PreCSR.c](#).**9.159.2.3 fasp_precond_amli()**

```
void fasp_precond_amli (  
    REAL * r,  
    REAL * z,  
    void * data )
```

AMLI AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

01/23/2011

Definition at line 482 of file [PreCSR.c](#).**9.159.2.4 fasp_precond_diag()**

```
void fasp_precond_diag (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Chensong Zhang

Date

04/06/2010

Definition at line 172 of file [PreCSR.c](#).**9.159.2.5 fasp_precond_famg()**

```
void fasp_precond_famg (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Full AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

02/27/2011

Definition at line 449 of file [PreCSR.c](#).**9.159.2.6 fasp_precond_ilu()**

```
void fasp_precond_ilu (  
    REAL * r,  
    REAL * z,  
    void * data )
```

ILU preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/06/2010

Definition at line 198 of file [PreCSR.c](#).**9.159.2.7 fasp_precond_ilu_backward()**

```
void fasp_precond_ilu_backward (
    REAL * r,
    REAL * z,
    void * data )
```

ILU preconditioner: only backward sweep.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu, Shiquan Zhang

Date

04/06/2010

Definition at line 317 of file [PreCSR.c](#).**9.159.2.8 fasp_precond_ilu_forward()**

```
void fasp_precond_ilu_forward (
    REAL * r,
    REAL * z,
    void * data )
```

ILU preconditioner: only forward sweep.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu, Shiquang Zhang

Date

04/06/2010

Definition at line 263 of file [PreCSR.c](#).

9.159.2.9 fasp_precond_namli()

```
void fasp_precond_namli (
    REAL * r,
    REAL * z,
    void * data )
```

Nonlinear AMLI AMG preconditioner.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

04/25/2011

Definition at line 515 of file [PreCSR.c](#).

9.159.2.10 fasp_precond_setup()

```
precond * fasp_precond_setup (
    const SHORT precondition_type,
    AMG_param * amgparam,
    ILU_param * iluparam,
    dCSRmat * A )
```

Setup preconditioner interface for iterative methods.

Parameters

<i>precond_type</i>	Preconditioner type
<i>amgparam</i>	Pointer to AMG parameters
<i>iluparam</i>	Pointer to ILU parameters
<i>A</i>	Pointer to the coefficient matrix

Returns

Pointer to preconditioner

Author

Feiteng Huang

Date

05/18/2009

Definition at line 46 of file [PreCSR.c](#).

9.159.2.11 fasp_precond_swz()

```
void fasp_precond_swz (
    REAL * r,
    REAL * z,
    void * data )
```

get z from r by Schwarz

Parameters

<i>r</i>	Pointer to residual
<i>z</i>	Pointer to preconditioned residual
<i>data</i>	Pointer to precondition data

Author

Xiaozhe Hu

Date

03/22/2010

Note

Change Schwarz interface by Zheng Li on 11/18/2014

Definition at line 371 of file [PreCSR.c](#).

9.160 PreCSR.c

[Go to the documentation of this file.](#)

```
00001
00017 #include "fasp.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/
00021 /*--  Declare Private Functions  --*/
00022 /*-----*/
00023
00024 #include "PreMGUtil.inl"
00025
00026 /*-----*/
00027 /*--      Public Functions      --*/
00028 /*-----*/
00029
00046 precondition *fasp_precond_setup (const SHORT   precondition_type,
00047                                   AMG_param     *amgparam,
00048                                   ILU_param      *iluparam,
00049                                   dCSRmat        *A)
00050 {
00051     precondition *pc = NULL;
00052     AMG_data      *mgl = NULL;
00053     precondition_data *pcdata = NULL;
00054     ILU_data      *ILU = NULL;
00055     dvector       *diag = NULL;
00056
00057     INT           max_levels, nnz, m, n;
00058
00059     switch (precond_type) {
00060
00061     case PREC_AMG: // AMG preconditioner
00062
00063         pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
00064         max_levels = amgparam->max_levels;
00065         nnz = A->nnz; m = A->row; n = A->col;
00066
```

```

00067         // initialize A, b, x for mgl[0]
00068         mgl=fasp_amg_data_create(max_levels);
00069         mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
00070         mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);
00071
00072         // setup preconditioner
00073         switch (amgparam->AMG_type) {
00074         case SA_AMG: // Smoothed Aggregation AMG
00075             fasp_amg_setup_sa(mgl, amgparam); break;
00076         case UA_AMG: // Unsmoothed Aggregation AMG
00077             fasp_amg_setup_ua(mgl, amgparam); break;
00078         default: // Classical AMG
00079             fasp_amg_setup_rs(mgl, amgparam); break;
00080         }
00081
00082         pcd_data = (precond_data *)fasp_mem_calloc(1, sizeof(precond_data));
00083         fasp_param_amg_to_prec(pcd_data, amgparam);
00084         pcd_data->max_levels = mgl[0].num_levels;
00085         pcd_data->mgl_data = mgl;
00086
00087         pc->data = pcd_data;
00088
00089         switch (amgparam->cycle_type) {
00090         case AMLI_CYCLE: // AMLI cycle
00091             pc->fct = fasp_precond_amli; break;
00092         case NL_AMLI_CYCLE: // Nonlinear AMLI
00093             pc->fct = fasp_precond_namli; break;
00094         default: // V,W-cycles or hybrid cycles
00095             pc->fct = fasp_precond_amg; break;
00096         }
00097
00098         break;
00099
00100     case PREC_FMG: // FMG preconditioner
00101
00102         pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
00103         max_levels = amgparam->max_levels;
00104         nnz = A->nnz; m = A->row; n = A->col;
00105
00106         // initialize A, b, x for mgl[0]
00107         mgl=fasp_amg_data_create(max_levels);
00108         mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
00109         mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);
00110
00111         // setup preconditioner
00112         switch (amgparam->AMG_type) {
00113         case SA_AMG: // Smoothed Aggregation AMG
00114             fasp_amg_setup_sa(mgl, amgparam); break;
00115         case UA_AMG: // Unsmoothed Aggregation AMG
00116             fasp_amg_setup_ua(mgl, amgparam); break;
00117         default: // Classical AMG
00118             fasp_amg_setup_rs(mgl, amgparam); break;
00119         }
00120
00121         pcd_data = (precond_data *)fasp_mem_calloc(1, sizeof(precond_data));
00122         fasp_param_amg_to_prec(pcd_data, amgparam);
00123         pcd_data->max_levels = mgl[0].num_levels;
00124         pcd_data->mgl_data = mgl;
00125
00126         pc->data = pcd_data; pc->fct = fasp_precond_famg;
00127
00128         break;
00129
00130     case PREC_ILU: // ILU preconditioner
00131
00132         pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
00133         ILU = (ILU_data *)fasp_mem_calloc(1, sizeof(ILU_data));
00134         fasp_ilu_dcsr_setup(A, ILU, iluparam);
00135         pc->data = ILU;
00136         pc->fct = fasp_precond_ilu;
00137
00138         break;
00139
00140     case PREC_DIAG: // Diagonal preconditioner
00141
00142         pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
00143         diag = (dvector *)fasp_mem_calloc(1, sizeof(dvector));
00144         fasp_dcsr_getdiag(0, A, diag);
00145
00146         pc->data = diag;
00147         pc->fct = fasp_precond_diag;

```



```

00148
00149         break;
00150
00151     default: // No preconditioner
00152
00153         break;
00154
00155     }
00156
00157     return pc;
00158 }
00159
00172 void fasp_precond_diag (REAL *r,
00173                         REAL *z,
00174                         void *data)
00175 {
00176     dvector *diag=(dvector *)data;
00177     REAL *diagptr=diag->val;
00178     INT i, m=diag->row;
00179
00180     memcpy(z,r,m*sizeof(REAL));
00181     for (i=0;i<m;++i) {
00182         if (ABS(diag->val[i])>SMALLREAL) z[i]/=diagptr[i];
00183     }
00184 }
00185
00198 void fasp_precond_ilu (REAL *r,
00199                       REAL *z,
00200                       void *data)
00201 {
00202     ILU_data *iludata=(ILU_data *)data;
00203     const INT m=iludata->row, mml=m-1, memneed=2*m;
00204     REAL *zz, *zr;
00205
00206     if (iludata->nwork<memneed) goto MEMERR; // check this outside this subroutine!!
00207
00208     zz = iludata->work;
00209     zr = iludata->work+m;
00210     fasp_darray_cp(m, r, zr);
00211
00212     {
00213         INT i, j, jj, begin_row, end_row, mm2=m-2;
00214         INT *ijlu=iludata->ijlu;
00215         REAL *lu=iludata->luval;
00216
00217         // forward sweep: solve unit lower matrix equation L*zz=zr
00218         zz[0]=zr[0];
00219
00220         for (i=1;i<=mml;++i) {
00221             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00222             for (j=begin_row;j<=end_row;++j) {
00223                 jj=ijlu[j];
00224                 if (jj<i) zr[i]-=lu[j]*zz[jj];
00225                 else break;
00226             }
00227             zz[i]=zr[i];
00228         }
00229
00230         // backward sweep: solve upper matrix equation U*z=zz
00231         z[mml]=zz[mml]*lu[mml];
00232         for (i=mm2;i>=0;i--) {
00233             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00234             for (j=end_row;j>=begin_row;j--) {
00235                 jj=ijlu[j];
00236                 if (jj>i) zz[i]-=lu[j]*z[jj];
00237                 else break;
00238             }
00239             z[i]=zz[i]*lu[i];
00240         }
00241     }
00242
00243     return;
00244
00245 MEMERR:
00246     printf("### ERROR: Need %d memory, only %d available!\n",
00247           memneed, iludata->nwork);
00248     fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
00249 }
00250
00263 void fasp_precond_ilu_forward (REAL *r,
00264                               REAL *z,

```

```

00265                                     void *data)
00266 {
00267     ILU_data *iludata=(ILU_data *)data;
00268     const INT m=iludata->row, mml=m-1, memneed=2*m;
00269     REAL *zz, *zr;
00270
00271     if (iludata->nwork<memneed) goto MEMERR;
00272
00273     zz = iludata->work;
00274     zr = iludata->work+m;
00275     fasp_darray_cp(m, r, zr);
00276
00277     {
00278         INT i, j, jj, begin_row, end_row;
00279         INT *ijlu=iludata->ijlu;
00280         REAL *lu=iludata->luval;
00281
00282         // forward sweep: solve unit lower matrix equation L*z=r
00283         zz[0]=zr[0];
00284         for (i=1;i<=mml;++i) {
00285             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00286             for (j=begin_row;j<=end_row;++j) {
00287                 jj=ijlu[j];
00288                 if (jj<i) zr[i]-=lu[j]*zz[jj];
00289                 else break;
00290             }
00291             zz[i]=zr[i];
00292         }
00293     }
00294
00295     fasp_darray_cp(m, zz, z);
00296
00297     return;
00298
00299 MEMERR:
00300     printf("### ERROR: Need %d memory, only %d available!",
00301         memneed, iludata->nwork);
00302     fasp_chkerf(ERROR_ALLOC_MEM, __FUNCTION__);
00303 }
00304
00317 void fasp_precond_ilu_backward (REAL *r,
00318                                REAL *z,
00319                                void *data)
00320 {
00321     ILU_data *iludata=(ILU_data *)data;
00322     const INT m=iludata->row, mml=m-1, memneed=2*m;
00323     REAL *zz;
00324
00325     if (iludata->nwork<memneed) goto MEMERR;
00326
00327     zz = iludata->work;
00328     fasp_darray_cp(m, r, zz);
00329
00330     {
00331         INT i, j, jj, begin_row, end_row, mm2=m-2;
00332         INT *ijlu=iludata->ijlu;
00333         REAL *lu=iludata->luval;
00334
00335         // backward sweep: solve upper matrix equation U*z=zz
00336         z[mml]=zz[mml]*lu[mml];
00337         for (i=mm2;i>=0;i--) {
00338             begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
00339             for (j=end_row;j>=begin_row;j--) {
00340                 jj=ijlu[j];
00341                 if (jj>i) zz[i]-=lu[j]*z[jj];
00342                 else break;
00343             }
00344             z[i]=zz[i]*lu[i];
00345         }
00346     }
00347
00348     return;
00349
00350 MEMERR:
00351     printf("### ERROR: Need %d memory, only %d available!",
00352         memneed, iludata->nwork);
00353     fasp_chkerf(ERROR_ALLOC_MEM, __FUNCTION__);
00354 }
00355
00371 void fasp_precond_swz (REAL *r,

```

```

00372             REAL *z,
00373             void *data)
00374 {
00375     SWZ_data * swzdata = (SWZ_data *)data;
00376     SWZ_param * swzparam = swzdata->swzparam;
00377     const INT swztype = swzdata->SWZ_type;
00378     const INT n = swzdata->A.row;
00379
00380     dvector x, b;
00381
00382     fasp_dvec_alloc(n, &x);
00383     fasp_dvec_alloc(n, &b);
00384     fasp_darray_cp(n, r, b.val);
00385
00386     fasp_dvec_set(n, &x, 0);
00387
00388     switch (swztype) {
00389         case SCHWARZ_BACKWARD:
00390             fasp_dcsr_swz_backward(swzdata, swzparam, &x, &b);
00391             break;
00392         case SCHWARZ_SYMMETRIC:
00393             fasp_dcsr_swz_forward(swzdata, swzparam, &x, &b);
00394             fasp_dcsr_swz_backward(swzdata, swzparam, &x, &b);
00395             break;
00396         default:
00397             fasp_dcsr_swz_forward(swzdata, swzparam, &x, &b);
00398             break;
00399     }
00400
00401     fasp_darray_cp(n, x.val, z);
00402 }
00403
00416 void fasp_precond_amg (REAL *r,
00417                       REAL *z,
00418                       void *data)
00419 {
00420     precondition_data *pcdata=(precondition_data *)data;
00421     const INT m=pcdata->mgl_data[0].A.row;
00422     const INT maxit=pcdata->maxit;
00423     INT i;
00424
00425     AMG_param amgparam; fasp_param_amg_init (&amgparam);
00426     fasp_param_prec_to_amg (&amgparam,pcdata);
00427
00428     AMG_data *mgl = pcdata->mgl_data;
00429     mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
00430     mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);
00431
00432     for ( i=maxit; i--; ) fasp_solver_mgcycle(mgl,&amgparam);
00433
00434     fasp_darray_cp(m,mgl->x.val,z);
00435 }
00436
00449 void fasp_precond_famg (REAL *r,
00450                       REAL *z,
00451                       void *data)
00452 {
00453     precondition_data *pcdata=(precondition_data *)data;
00454     const INT m=pcdata->mgl_data[0].A.row;
00455     const INT maxit=pcdata->maxit;
00456     INT i;
00457
00458     AMG_param amgparam; fasp_param_amg_init (&amgparam);
00459     fasp_param_prec_to_amg (&amgparam,pcdata);
00460
00461     AMG_data *mgl = pcdata->mgl_data;
00462     mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
00463     mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);
00464
00465     for ( i=maxit; i--; ) fasp_solver_fmecycle(mgl,&amgparam);
00466
00467     fasp_darray_cp(m,mgl->x.val,z);
00468 }
00469
00482 void fasp_precond_amli (REAL *r,
00483                       REAL *z,
00484                       void *data)
00485 {
00486     precondition_data *pcdata=(precondition_data *)data;
00487     const INT m=pcdata->mgl_data[0].A.row;
00488     const INT maxit=pcdata->maxit;

```

```

00489     INT i;
00490
00491     AMG_param amgparam; fasp_param_amg_init(&amgparam);
00492     fasp_param_prec_to_amg(&amgparam, pcddata);
00493
00494     AMG_data *mgl = pcddata->mgl_data;
00495     mgl->b.row=m; fasp_darray_cp(m, r, mgl->b.val); // residual is an input
00496     mgl->x.row=m; fasp_dvec_set(m, &mgl->x, 0.0);
00497
00498     for ( i=maxit; i--; ) fasp_solver_amli(mgl, &amgparam, 0);
00499
00500     fasp_darray_cp(m, mgl->x.val, z);
00501 }
00502
00515 void fasp_precond_namli (REAL *r,
00516                        REAL *z,
00517                        void *data)
00518 {
00519     precondition_data *pcdata=(precondition_data *)data;
00520     const INT m=pcdata->mgl_data[0].A.row;
00521     const INT maxit=pcdata->maxit;
00522     const SHORT num_levels = pcdata->max_levels;
00523     INT i;
00524
00525     AMG_param amgparam; fasp_param_amg_init(&amgparam);
00526     fasp_param_prec_to_amg(&amgparam, pcddata);
00527
00528     AMG_data *mgl = pcddata->mgl_data;
00529     mgl->b.row=m; fasp_darray_cp(m, r, mgl->b.val); // residual is an input
00530     mgl->x.row=m; fasp_dvec_set(m, &mgl->x, 0.0);
00531
00532     for ( i=maxit; i--; ) fasp_solver_namli(mgl, &amgparam, 0, num_levels);
00533     fasp_darray_cp(m, mgl->x.val, z);
00534 }
00535
00548 void fasp_precond_amg_nk (REAL *r,
00549                          REAL *z,
00550                          void *data)
00551 {
00552     precondition_data *pcdata=(precondition_data *)data;
00553     const INT m=pcdata->mgl_data[0].A.row;
00554     const INT maxit=pcdata->maxit;
00555     INT i;
00556
00557     dCSRmat *A_nk = pcdata->A_nk;
00558     dCSRmat *P_nk = pcdata->P_nk;
00559     dCSRmat *R_nk = pcdata->R_nk;
00560
00561     fasp_darray_set(m, z, 0.0);
00562
00563     // local variables
00564     dvector r_nk, z_nk;
00565     fasp_dvec_alloc(A_nk->row, &r_nk);
00566     fasp_dvec_alloc(A_nk->row, &z_nk);
00567
00568     //-----
00569     // extra kernel solve
00570     //-----
00571     // r_nk = R_nk*r
00572     fasp_blas_dcsr_mxv(R_nk, r, r_nk.val);
00573
00574     // z_nk = A_nk^{-1}*r_nk
00575     #if WITH_UMFPACK // use UMFPACK directly
00576     fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
00577     #else
00578     fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
00579     #endif
00580
00581     // z = z + P_nk*z_nk;
00582     fasp_blas_dcsr_aAxy(1.0, P_nk, z_nk.val, z);
00583
00584     //-----
00585     // AMG solve
00586     //-----
00587     AMG_param amgparam; fasp_param_amg_init(&amgparam);
00588     fasp_param_prec_to_amg(&amgparam, pcddata);
00589
00590     AMG_data *mgl = pcddata->mgl_data;
00591     mgl->b.row=m; fasp_darray_cp(m, r, mgl->b.val); // residual is an input
00592     mgl->x.row=m; //fasp_dvec_set(m, &mgl->x, 0.0);
00593     fasp_darray_cp(m, z, mgl->x.val);

```

```

00594
00595     for ( i=maxit; i--; ) fasp_solver_mgcycle(mgl,&amgparam);
00596
00597     fasp_darray_cp(m,mgl->x.val,z);
00598
00599     //-----
00600     // extra kernel solve
00601     //-----
00602     // r = r - A*z
00603     fasp_blas_dcsr_aApy(-1.0, &(pdata->mgl_data[0].A), z, mgl->b.val);
00604
00605     // r_nk = R_nk*r
00606     fasp_blas_dcsr_mxv(R_nk, mgl->b.val, r_nk.val);
00607
00608     // z_nk = A_nk^{-1}*r_nk
00609 #if WITH_UMFPACK // use UMFPACK directly
00610     fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
00611 #else
00612     fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
00613 #endif
00614
00615     // z = z + P_nk*z_nk;
00616     fasp_blas_dcsr_aApy(1.0, P_nk, z_nk.val, z);
00617 }
00618
00619 /*-----*/
00620 /*--          End of File          --*/
00621 /*-----*/

```

9.161 PreDataInit.c File Reference

Initialize important data structures.

```

#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void [fasp_precond_data_init](#) ([precond_data](#) *pdata)

Initialize [precond_data](#).
- [AMG_data](#) * [fasp_amg_data_create](#) ([SHORT](#) max_levels)

Create and initialize [AMG_data](#) for classical and SA AMG.
- void [fasp_amg_data_free](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)

Free [AMG_data](#) data memeory space.
- [AMG_data_bsr](#) * [fasp_amg_data_bsr_create](#) ([SHORT](#) max_levels)

Create and initialize [AMG_data](#) data sturcture for AMG/SAMG (BSR format)
- void [fasp_amg_data_bsr_free](#) ([AMG_data_bsr](#) *mgl)

Free [AMG_data_bsr](#) data memeory space.
- void [fasp_ilu_data_create](#) (const [INT](#) iwk, const [INT](#) nwork, [ILU_data](#) *iludata)

Allocate workspace for ILU factorization.
- void [fasp_ilu_data_free](#) ([ILU_data](#) *iludata)

Create [ILU_data](#) sturcture.
- void [fasp_swz_data_free](#) ([SWZ_data](#) *swzdata)

Free [SWZ_data](#) data memeory space.

9.161.1 Detailed Description

Initialize important data structures.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxMemory.c](#), [AuxVector.c](#), [BlaSparseBSR.c](#), and [BlaSparseCSR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Warning

Every structures should be initialized before usage.

Definition in file [PreDataInit.c](#).

9.161.2 Function Documentation

9.161.2.1 fasp_amg_data_bsr_create()

```
AMG_data_bsr * fasp_amg_data_bsr_create (
    SHORT max_levels )
```

Create and initialize [AMG_data](#) data sturcture for AMG/SAMG (BSR format)

Parameters

<i>max_levels</i>	Max number of levels allowed
-------------------	------------------------------

Returns

Pointer to the [AMG_data](#) data structure

Author

Xiaozhe Hu

Date

08/07/2011

Definition at line [181](#) of file [PreDataInit.c](#).

9.161.2.2 fasp_amg_data_bsr_free()

```
void fasp_amg_data_bsr_free (
    AMG_data_bsr * mgl )
```

Free [AMG_data_bsr](#) data memeory space.

Parameters

<i>mgl</i>	Pointer to the AMG_data_bsr
------------	---

Author

Xiaozhe Hu, Chensong Zhang

Date

2013/02/13

Modified by Chensong Zhang on 08/14/2017: Check for max_levels == 1
Definition at line 213 of file [PreDataInit.c](#).

9.161.2.3 fasp_amg_data_create()

```
AMG_data * fasp_amg_data_create (
    SHORT max_levels )
```

Create and initialize [AMG_data](#) for classical and SA AMG.

Parameters

<i>max_levels</i>	Max number of levels allowed
-------------------	------------------------------

Returns

Pointer to the [AMG_data](#) data structure

Author

Chensong Zhang

Date

2010/04/06

Definition at line 64 of file [PreDataInit.c](#).

9.161.2.4 fasp_amg_data_free()

```
void fasp_amg_data_free (
    AMG_data * mgl,
    AMG_param * param )
```

Free [AMG_data](#) data memeory space.

Parameters

<i>mgl</i>	Pointer to the AMG_data
<i>param</i>	Pointer to AMG parameters

Author

Chensong Zhang

Date

2010/04/06

Modified by Chensong Zhang on 05/05/2013: Clean up param as well! Modified by Hongxuan Zhang on 12/15/2015: Free memory for Intel MKL PARDISO Modified by Chunsheng Feng on 02/12/2017: Permute A back to its origin for ILUtp Modified by Chunsheng Feng on 08/11/2017: Check for max_levels == 1
Definition at line 101 of file [PreDataInit.c](#).

9.161.2.5 fasp_ilu_data_create()

```
void fasp_ilu_data_create (
    const INT iwk,
    const INT nwork,
    ILU_data * iludata )
```

Allocate workspace for ILU factorization.

Parameters

<i>iwk</i>	Size of the index array
<i>nwork</i>	Size of the work array
<i>iludata</i>	Pointer to the ILU_data

Author

Chensong Zhang

Date

2010/04/06

Modified by Chunsheng Feng on 02/12/2017: add iperm array for ILUtp
Definition at line 265 of file [PreDataInit.c](#).

9.161.2.6 fasp_ilu_data_free()

```
void fasp_ilu_data_free (
    ILU_data * iludata )
```

Create [ILU_data](#) sturcture.

Parameters

<i>iludata</i>	Pointer to ILU_data
----------------	-------------------------------------

Author

Chensong Zhang

Date

2010/04/03

Modified by Chunsheng Feng on 02/12/2017: add iperm array for ILUtp
Definition at line 300 of file [PreDataInit.c](#).

9.161.2.7 fasp_precond_data_init()

```
void fasp_precond_data_init (
    precondition_data * pcddata )
```

Initialize [precond_data](#).

Parameters

<i>pcddata</i>	Preconditioning data structure
----------------	--------------------------------

Author

Chensong Zhang

Date

2010/03/23

Definition at line 33 of file [PreDataInit.c](#).

9.161.2.8 fasp_swz_data_free()

```
void fasp_swz_data_free (
    SWZ_data * swzdata )
```

Free [SWZ_data](#) data memeory space.

Parameters

<i>swzdata</i>	Pointer to the SWZ_data for Schwarz methods
----------------	---

Author

Xiaozhe Hu

Date

2010/04/06

Definition at line 341 of file [PreDataInit.c](#).

9.162 PreDataInit.c

[Go to the documentation of this file.](#)

```
00001
00016 #include "fasp.h"
00017 #include "fasp_functs.h"
00018
00019 /*-----*/
00020 /*--      Public Functions      --*/
00021 /*-----*/
00022
00033 void fasp_precond_data_init (precond_data *pcddata)
00034 {
00035     pcddata->AMG_type           = CLASSIC_AMG;
00036     pcddata->print_level        = PRINT_NONE;
00037     pcddata->maxit               = 500;
00038     pcddata->max_levels          = 20;
00039     pcddata->tol                 = 1e-8;
00040     pcddata->cycle_type          = V_CYCLE;
```

```

00041     pcddata->smoother          = SMOOTHER_GS;
00042     pcddata->smooth_order      = CF_ORDER;
00043     pcddata->presmooth_iter    = 1;
00044     pcddata->postsmooth_iter   = 1;
00045     pcddata->relaxation         = 1.1;
00046     pcddata->coarsening_type   = 1;
00047     pcddata->coarse_scaling     = ON;
00048     pcddata->amli_degree       = 1;
00049     pcddata->nl_amli_krylov_type = SOLVER_GCG;
00050 }
00051
00064 AMG_data * fasp_amg_data_create (SHORT max_levels)
00065 {
00066     max_levels = MAX(1, max_levels); // at least allocate one level
00067
00068     AMG_data *mgl = (AMG_data *)fasp_mem_calloc(max_levels, sizeof(AMG_data));
00069
00070     INT i;
00071     for ( i=0; i<max_levels; ++i ) {
00072         mgl[i].max_levels = max_levels;
00073         mgl[i].num_levels = 0;
00074         mgl[i].near_kernel_dim = 0;
00075         mgl[i].near_kernel_basis = NULL;
00076         mgl[i].cycle_type = 0;
00077         #if MULTI_COLOR_ORDER
00078         mgl[i].GS_Theta = 0.0E-2; //0.0; //1.0E-2;
00079         #endif
00080     }
00081
00082     return(mgl);
00083 }
00084
00101 void fasp_amg_data_free (AMG_data *mgl,
00102                         AMG_param *param)
00103 {
00104     const INT max_levels = MAX(1, mgl[0].num_levels);
00105
00106     INT i;
00107
00108     switch (param->coarse_solver) {
00109
00110     #if WITH_MUMPS
00111         /* Destroy MUMPS direct solver on the coarsest level */
00112         case SOLVER_MUMPS: {
00113             mgl[max_levels-1].mumps.job = 3;
00114             fasp_solver_mumps_steps(&mgl[max_levels-1].A, &mgl[max_levels-1].b,
00115                                     &mgl[max_levels-1].x, &mgl[max_levels-1].mumps);
00116             break;
00117         }
00118     #endif
00119
00120     #if WITH_UMFPACK
00121         /* Destroy UMFPACK direct solver on the coarsest level */
00122         case SOLVER_UMFPACK: {
00123             fasp_mem_free(mgl[max_levels-1].Numeric); mgl[max_levels-1].Numeric = NULL;
00124             break;
00125         }
00126     #endif
00127
00128     #if WITH_PARDISO
00129         /* Destroy PARDISO direct solver on the coarsest level */
00130         case SOLVER_PARDISO: {
00131             fasp_pardiso_free_internal_mem(&mgl[max_levels-1].pdata);
00132             break;
00133         }
00134     #endif
00135
00136     default: // Do nothing!
00137         break;
00138     }
00139
00140     for ( i=0; i<max_levels; ++i ) {
00141         fasp_ilu_data_free(&mgl[i].LU);
00142         fasp_dcsr_free(&mgl[i].A);
00143         if ( max_levels > 1 ) {
00144             fasp_dcsr_free(&mgl[i].P);
00145             fasp_dcsr_free(&mgl[i].R);
00146         }
00147         fasp_dvec_free(&mgl[i].b);
00148         fasp_dvec_free(&mgl[i].x);
00149         fasp_dvec_free(&mgl[i].w);

```

```

00150         fasp_ivec_free(&mgl[i].cfmark);
00151         fasp_swz_data_free(&mgl[i].Schwarz);
00152     }
00153
00154     for ( i=0; i<mgl->near_kernel_dim; ++i ) {
00155         fasp_mem_free(mgl->near_kernel_basis[i]); mgl->near_kernel_basis[i] = NULL;
00156     }
00157
00158     fasp_mem_free(mgl->near_kernel_basis); mgl->near_kernel_basis = NULL;
00159     fasp_mem_free(mgl); mgl = NULL;
00160
00161     if ( param == NULL ) return; // exit if no param given
00162
00163     if ( param->cycle_type == AMLI_CYCLE ) {
00164         fasp_mem_free(param->aqli_coef); param->aqli_coef = NULL;
00165     }
00166 }
00167 }
00168
00181 AMG_data_bsr * fasp_amg_data_bsr_create (SHORT max_levels)
00182 {
00183     max_levels = MAX(1, max_levels); // at least allocate one level
00184
00185     AMG_data_bsr *mgl = (AMG_data_bsr *)fasp_mem_calloc(max_levels, sizeof(AMG_data_bsr));
00186
00187     INT i;
00188     for (i=0; i<max_levels; ++i) {
00189         mgl[i].max_levels = max_levels;
00190         mgl[i].num_levels = 0;
00191         mgl[i].near_kernel_dim = 0;
00192         mgl[i].near_kernel_basis = NULL;
00193         mgl[i].A_nk = NULL;
00194         mgl[i].P_nk = NULL;
00195         mgl[i].R_nk = NULL;
00196     }
00197
00198     return(mgl);
00199 }
00200
00213 void fasp_amg_data_bsr_free (AMG_data_bsr *mgl)
00214 {
00215     const INT max_levels = MAX(1, mgl[0].num_levels);
00216
00217     INT i;
00218
00219     for ( i = 0; i < max_levels; ++i ) {
00220
00221         fasp_ilu_data_free(&mgl[i].LU);
00222         fasp_dbsr_free(&mgl[i].A);
00223         if ( max_levels > 1 ) {
00224             fasp_dbsr_free(&mgl[i].P);
00225             fasp_dbsr_free(&mgl[i].R);
00226         }
00227         fasp_dvec_free(&mgl[i].b);
00228         fasp_dvec_free(&mgl[i].x);
00229         fasp_dvec_free(&mgl[i].diaginv);
00230         fasp_dvec_free(&mgl[i].diaginv_SS);
00231         fasp_dcsr_free(&mgl[i].Ac);
00232
00233         fasp_ilu_data_free(&mgl[i].PP_LU);
00234         fasp_dcsr_free(&mgl[i].PP);
00235         fasp_dbsr_free(&mgl[i].SS);
00236         fasp_dvec_free(&mgl[i].diaginv_SS);
00237         fasp_dvec_free(&mgl[i].w);
00238         fasp_ivec_free(&mgl[i].cfmark);
00239
00240         fasp_mem_free(mgl[i].pw); mgl[i].pw = NULL;
00241         fasp_mem_free(mgl[i].sw); mgl[i].sw = NULL;
00242     }
00243
00244     for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
00245         fasp_mem_free(mgl->near_kernel_basis[i]); mgl->near_kernel_basis[i] = NULL;
00246     }
00247     fasp_mem_free(mgl->near_kernel_basis); mgl->near_kernel_basis = NULL;
00248     fasp_mem_free(mgl); mgl = NULL;
00249 }
00250
00265 void fasp_ilu_data_create (const INT    iwk,
00266                           const INT    nwork,
00267                           ILU_data     *iludata)
00268 {

```

```

00269 #if DEBUG_MODE > 0
00270     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00271     printf("### DEBUG: iwk=%d, nwork=%d\n", iwk, nwork);
00272 #endif
00273
00274     iludata->ijlu=(INT*) fasp_mem_calloc(iwk, sizeof(INT));
00275
00276     if (iludata->type == ILUtp) iludata->iperm=(INT*) fasp_mem_calloc(iludata->row*2, sizeof(INT));
00277
00278     iludata->luval=(REAL*) fasp_mem_calloc(iwk, sizeof(REAL));
00279
00280     iludata->work=(REAL*) fasp_mem_calloc(nwork, sizeof(REAL));
00281 #if DEBUG_MODE > 0
00282     printf("### DEBUG: %s ..... %d [End]\n", __FUNCTION__, __LINE__);
00283 #endif
00284     return;
00285 }
00286
00287
00300 void fasp_ilu_data_free (ILU_data *iludata)
00301 {
00302     if ( iludata == NULL ) return; // There is nothing to do!
00303
00304     fasp_mem_free(iludata->ijlu); iludata->ijlu = NULL;
00305     fasp_mem_free(iludata->luval); iludata->luval = NULL;
00306     fasp_mem_free(iludata->work); iludata->work = NULL;
00307     fasp_mem_free(iludata->ilevL); iludata->ilevL = NULL;
00308     fasp_mem_free(iludata->jlevL); iludata->jlevL = NULL;
00309     fasp_mem_free(iludata->ilevU); iludata->ilevU = NULL;
00310     fasp_mem_free(iludata->jlevU); iludata->jlevU = NULL;
00311
00312     if ( iludata->type == ILUtp ) {
00313
00314         if ( iludata->A != NULL ) {
00315             // To permute the matrix back to its original state use the loop:
00316             INT k;
00317             const INT nnz = iludata->A->nnz;
00318             const INT *iperm = iludata->iperm;
00319             for ( k = 0; k < nnz; k++ ) {
00320                 // iperm is in Fortran array format
00321                 iludata->A->JA[k] = iperm[ iludata->A->JA[k] ] -1;
00322             }
00323         }
00324
00325         fasp_mem_free(iludata->iperm); iludata->iperm = NULL;
00326     }
00327
00328     iludata->row = iludata->col = iludata->nzlu = iludata->nwork = \
00329     iludata->nb = iludata->nlevL = iludata->nlevU = 0;
00330 }
00331
00341 void fasp_swz_data_free (SWZ_data *swzdata)
00342 {
00343     INT i;
00344
00345     if ( swzdata == NULL ) return; // There is nothing to do!
00346
00347     fasp_dcsr_free(&swzdata->A);
00348
00349     for ( i=0; i<swzdata->nblk; ++i ) fasp_dcsr_free (&((swzdata->blk_data)[i]));
00350
00351     swzdata->nblk = 0;
00352
00353     fasp_mem_free (swzdata->iblock); swzdata->iblock = NULL;
00354     fasp_mem_free (swzdata->jblock); swzdata->jblock = NULL;
00355
00356     fasp_dvec_free (&swzdata->rhsloc1);
00357     fasp_dvec_free (&swzdata->xloc1);
00358
00359     swzdata->memt = 0;
00360     fasp_mem_free (swzdata->mask); swzdata->mask = NULL;
00361     fasp_mem_free (swzdata->maxa); swzdata->maxa = NULL;
00362
00363 #if WITH_MUMPS
00364     if ( swzdata->mumps == NULL ) return;
00365
00366     for ( i=0; i<swzdata->nblk; ++i ) fasp_mumps_free (&((swzdata->mumps)[i]));
00367 #endif
00368 }
00369
00370 /*-----*/

```

```
00371 /*--          End of File          --*/
00372 /*-----*
```

9.163 PreMGCycle.c File Reference

Abstract multigrid cycle – non-recursive version.

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"
#include "PreMGSmother.inl"
```

Functions

- void [fasp_solver_mgcycle](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
Solve $Ax=b$ with non-recursive multigrid cycle.
- void [fasp_solver_mgcycle_bsr](#) ([AMG_data_bsr](#) *mgl, [AMG_param](#) *param)
Solve $Ax=b$ with non-recursive multigrid cycle.

9.163.1 Detailed Description

Abstract multigrid cycle – non-recursive version.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaArray.c](#), [BlaSchwarzSetup.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), [ltrSmootherBSR.c](#), [ltrSmootherCSR.c](#), [ltrSmootherCSRpoly.c](#), [KryPcg.c](#), [KryPvgmres.c](#), [KrySPcg.c](#), and [KrySPvgmres.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreMGCycle.c](#).

9.163.2 Function Documentation

9.163.2.1 [fasp_solver_mgcycle\(\)](#)

```
void fasp_solver_mgcycle (
    AMG\_data * mgl,
    AMG\_param * param )
```

Solve $Ax=b$ with non-recursive multigrid cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Author

Chensong Zhang

Date

10/06/2010

Modified by Chensong Zhang on 02/27/2013: update direct solvers. Modified by Chensong Zhang on 12/30/2014: update Schwarz smoothers.

Definition at line 48 of file [PreMGCycle.c](#).

9.163.2.2 fasp_solver_mgcycle_bsr()

```
void fasp_solver_mgcycle_bsr (
    AMG_data_bsr * mgl,
    AMG_param * param )
```

Solve $Ax=b$ with non-recursive multigrid cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data_bsr
<i>param</i>	Pointer to AMG parameters: AMG_param

Author

Xiaozhe Hu

Date

08/07/2011

Definition at line 268 of file [PreMGCycle.c](#).

9.164 PreMGCycle.c

[Go to the documentation of this file.](#)

```
00001
00017 #include <math.h>
00018 #include <time.h>
00019
00020 #include "fasp.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*-- Declare Private Functions --*/
00025 /*-----*/
00026
00027 #include "PreMGUtil.inl"
00028 #include "PreMGSmoothing.inl"
00029
00030 /*-----*/
00031 /*-- Public Functions --*/
00032 /*-----*/
00033
00048 void fasp_solver_mgcycle (AMG_data *mgl,
00049                          AMG_param *param)
00050 {
00051     const SHORT prtlvl = param->print_level;
00052     const SHORT amg_type = param->AMG_type;
00053     const SHORT smoother = param->smoother;
00054     const SHORT smooth_order = param->smooth_order;
```

```

00055     const SHORT cycle_type = param->cycle_type;
00056     const SHORT coarse_solver = param->coarse_solver;
00057     const SHORT nl = mgl[0].num_levels;
00058     const REAL relax = param->relaxation;
00059     const REAL tol = param->tol * 1e-4;
00060     const SHORT ndeg = param->polynomial_degree;
00061
00062     // Schwarz parameters
00063     SWZ_param swzparam;
00064     if ( param->SWZ_levels > 0 ) {
00065         swzparam.SWZ_bklsolver = param->SWZ_bklsolver;
00066     }
00067
00068     // local variables
00069     REAL alpha = 1.0;
00070     INT num_lvl[MAX_AMG_LVL] = {0}, l = 0;
00071
00072     // more general cycling types on each level --zcs 05/07/2020
00073     INT ncycles[MAX_AMG_LVL] = {1};
00074     SHORT i;
00075     for ( i = 0; i < MAX_AMG_LVL; ++i ) ncycles[i] = 1; // initially V-cycle
00076     switch(cycle_type) {
00077         case 12:
00078             for ( i = MAX_AMG_LVL-2; i > 0; i -= 2 ) ncycles[i] = 2;
00079             break;
00080         case 21:
00081             for ( i = MAX_AMG_LVL-1; i > 0; i -= 2 ) ncycles[i] = 2;
00082             break;
00083         default:
00084             for ( i = 0; i < MAX_AMG_LVL; i += 1 ) ncycles[i] = cycle_type;
00085     }
00086
00087     #if DEBUG_MODE > 0
00088         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00089         printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
00090     #endif
00091
00092     #if DEBUG_MODE > 1
00093         printf("### DEBUG: AMG_level = %d, ILU_level = %d\n", nl, mgl->ILU_levels);
00094     #endif
00095
00096     ForwardSweep:
00097         while ( l < nl-1 ) {
00098
00099             num_lvl[l]++;
00100
00101             // pre-smoothing with ILU method
00102             if ( l < mgl->ILU_levels ) {
00103                 fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
00104             }
00105
00106             // or pre-smoothing with Schwarz method
00107             else if ( l < mgl->SWZ_levels ) {
00108                 switch (mgl[l].Schwarz.SWZ_type) {
00109                     case SCHWARZ_SYMMETRIC:
00110                         fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00111                         fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00112                         break;
00113                     default:
00114                         fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00115                         break;
00116                 }
00117             }
00118
00119             // or pre-smoothing with standard smoother
00120             else {
00121                 #if MULTI_COLOR_ORDER
00122                     // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n", __FUNCTION__, __LINE__);
00123                     fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter, 1);
00124                 #else
00125                     fasp_dcsr_presmoothing(smoother, &mgl[l].A, &mgl[l].b, &mgl[l].x,
00126                                           param->presmooth_iter, 0, mgl[l].A.row-1, 1,
00127                                           relax, ndeg, smooth_order, mgl[l].cfmark.val);
00128                 #endif
00129             }
00130
00131             // form residual r = b - A x
00132             fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
00133             fasp_blas_dcsr_aAxy(-1.0, &mgl[l].A, mgl[l].x.val, mgl[l].w.val);
00134
00135             // restriction r_l = R*r0

```

```

00136         switch ( amg_type ) {
00137             case UA_AMG:
00138                 fasp_blas_dcsr_mxv_agg(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
00139                 break;
00140             default:
00141                 fasp_blas_dcsr_mxv(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
00142                 break;
00143         }
00144
00145         // prepare for the next level
00146         ++l; fasp_dvec_set(mgl[l].A.row, &mgl[l].x, 0.0);
00147
00148     }
00149
00150     // If AMG only has one level or we have arrived at the coarsest level,
00151     // call the coarse space solver:
00152     switch ( coarse_solver ) {
00153
00154     #if WITH_PARDISO
00155         case SOLVER_PARDISO: {
00156             /* use Intel MKL PARDISO direct solver on the coarsest level */
00157             fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
00158             break;
00159         }
00160     #endif
00161
00162     #if WITH_MUMPS
00163         case SOLVER_MUMPS: {
00164             // use MUMPS direct solver on the coarsest level
00165             mgl[nl-1].mumps.job = 2;
00166             fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
00167             break;
00168         }
00169     #endif
00170
00171     #if WITH_UMFPACK
00172         case SOLVER_UMFPACK: {
00173             // use UMFPACK direct solver on the coarsest level
00174             fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
00175             break;
00176         }
00177     #endif
00178
00179     #if WITH_SuperLU
00180         case SOLVER_SUPERLU: {
00181             // use SuperLU direct solver on the coarsest level
00182             fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
00183             break;
00184         }
00185     #endif
00186
00187         default:
00188             // use iterative solver on the coarsest level
00189             fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);
00190     }
00191
00192     // BackwardSweep:
00193     while ( l > 0 ) {
00194
00195         --l;
00196
00197         // find the optimal scaling factor alpha
00198         if ( param->coarse_scaling == ON ) {
00199             alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
00200                 / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
00201             alpha = MIN(alpha, 1.0); // Add this for safety! --Chensong on 10/04/2014
00202         }
00203
00204         // prolongation u = u + alpha*P*e1
00205         switch ( amg_type ) {
00206             case UA_AMG:
00207                 fasp_blas_dcsr_aApy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
00208                 break;
00209             default:
00210                 fasp_blas_dcsr_aApy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
00211                 break;
00212         }
00213
00214         // post-smoothing with ILU method
00215         if ( l < mgl->ILU_levels ) {

```



```

00217         fasp_smoother_dcsr_ilu(&mgl[1].A, &mgl[1].b, &mgl[1].x, &mgl[1].LU);
00218     }
00219
00220     // post-smoothing with Schwarz method
00221     else if ( 1 < mgl->SWZ_levels ) {
00222         switch (mgl[1].Schwarz.SWZ_type) {
00223             case SCHWARZ_SYMMETRIC:
00224                 fasp_dcsr_swz_backward(&mgl[1].Schwarz,&swzparam, &mgl[1].x, &mgl[1].b);
00225                 fasp_dcsr_swz_forward(&mgl[1].Schwarz, &swzparam, &mgl[1].x, &mgl[1].b);
00226                 break;
00227             default:
00228                 fasp_dcsr_swz_backward(&mgl[1].Schwarz,&swzparam, &mgl[1].x, &mgl[1].b);
00229                 break;
00230         }
00231     }
00232
00233     // post-smoothing with standard methods
00234     else {
00235 #if MULTI_COLOR_ORDER
00236         fasp_smoother_dcsr_gs_multicolor (&mgl[1].x, &mgl[1].A, &mgl[1].b, param->postsmooth_iter,-1);
00237 #else
00238         fasp_dcsr_postsmoothing(smoother, &mgl[1].A, &mgl[1].b, &mgl[1].x,
00239                                param->postsmooth_iter, 0, mgl[1].A.row-1, -1,
00240                                relax, ndeg, smooth_order, mgl[1].cfmark.val);
00241 #endif
00242     }
00243
00244     // General cycling on each level --zcs
00245     if ( num_lvl[1] < ncycles[1] ) break;
00246     else num_lvl[1] = 0;
00247 }
00248
00249 if ( 1 > 0 ) goto ForwardSweep;
00250
00251 #if DEBUG_MODE > 0
00252 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00253 #endif
00254 }
00255 }
00256
00268 void fasp_solver_mgcycle_bsr (AMG_data_bsr *mgl,
00269                               AMG_param *param)
00270 {
00271     const SHORT prtlvl      = param->print_level;
00272     const SHORT nl          = mgl[0].num_levels;
00273     const SHORT smoother    = param->smoother;
00274     const SHORT cycle_type   = param->cycle_type;
00275     const SHORT coarse_solver = param->coarse_solver;
00276     const REAL relax        = param->relaxation;
00277     INT steps               = param->presmooth_iter;
00278
00279     // local variables
00280     INT nu_l[MAX_AMG_LVL] = {0}, l = 0;
00281     REAL alpha = 1.0;
00282     INT i;
00283
00284     dvector r_nk, z_nk;
00285
00286     if ( mgl[0].A_nk != NULL ) {
00287         fasp_dvec_alloc(mgl[0].A_nk->row, &r_nk);
00288         fasp_dvec_alloc(mgl[0].A_nk->row, &z_nk);
00289     }
00290
00291 #if DEBUG_MODE > 0
00292 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00293 #endif
00294
00295 #if DEBUG_MODE > 1
00296 printf("### DEBUG: AMG_level = %d, ILU_level = %d\n", nl, mgl->ILU_levels);
00297 #endif
00298
00299 ForwardSweep:
00300     while ( 1 < nl-1 ) {
00301         nu_l[1]++;
00302         // pre smoothing
00303         if ( 1 < mgl->ILU_levels ) {
00304             fasp_smoother_dbsr_ilu(&mgl[1].A, &mgl[1].b, &mgl[1].x, &mgl[1].LU);
00305             for ( i=0; i<steps; i++ )
00306                 fasp_smoother_dbsr_gs_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x, mgl[1].diaginv.val);
00307         }
00308         else {

```

```

00309         if ( steps > 0 ) {
00310             switch ( smoother ) {
00311                 case SMOOTHER_JACOBI:
00312                     for (i=0; i<steps; i++)
00313                         fasp_smoother_dbsr_jacobi1(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00314                                                  mgl[1].diaginv.val);
00315                     break;
00316                 case SMOOTHER_GS:
00317                     for (i=0; i<steps; i++)
00318                         fasp_smoother_dbsr_gs_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00319                                                    mgl[1].diaginv.val);
00320                     break;
00321                 case SMOOTHER_SGS:
00322                     for (i=0; i<steps; i++){
00323                         fasp_smoother_dbsr_gs_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00324                                                    mgl[1].diaginv.val);
00325                         fasp_smoother_dbsr_gs_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00326                                                    mgl[1].diaginv.val);
00327                     }
00328                     break;
00329                 case SMOOTHER_SOR:
00330                     for (i=0; i<steps; i++)
00331                         fasp_smoother_dbsr_sor_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00332                                                    mgl[1].diaginv.val, relax);
00333                     break;
00334                 case SMOOTHER_SSOR:
00335                     for (i=0; i<steps; i++) {
00336                         fasp_smoother_dbsr_sor_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00337                                                    mgl[1].diaginv.val, relax);
00338                     }
00339                     fasp_smoother_dbsr_sor_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00340                                                    mgl[1].diaginv.val, relax);
00341                     break;
00342                 default:
00343                     printf("### ERROR: Unknown smoother type %d!\n", smoother);
00344                     fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
00345             }
00346         }
00347     }
00348
00349     // extra kernel solve
00350     if (mgl[1].A_nk != NULL) {
00351         //-----
00352         // extra kernel solve
00353         //-----
00354         // form residual r = b - A x
00355         fasp_darray_cp(mgl[1].A.ROW*mgl[1].A.nb, mgl[1].b.val, mgl[1].w.val);
00356         fasp_blas_dbsr_aAxy(-1.0,&mgl[1].A, mgl[1].x.val, mgl[1].w.val);
00357
00358         // r_nk = R_nk*r
00359         fasp_blas_dcsr_mxv(mgl[1].R_nk, mgl[1].w.val, r_nk.val);
00360
00361         // z_nk = A_nk^{(-1)}*r_nk
00362         #if WITH_UMFPACK // use UMFPACK directly
00363             fasp_solver_umfpack(mgl[1].A_nk, &r_nk, &z_nk, 0);
00364         #else
00365             fasp_coarse_itsolver(mgl[1].A_nk, &r_nk, &z_nk, 1e-12, 0);
00366         #endif
00367
00368         // z = z + P_nk*z_nk;
00369         fasp_blas_dcsr_aAxy(1.0, mgl[1].P_nk, z_nk.val, mgl[1].x.val);
00370     }
00371
00372     // form residual r = b - A x
00373     fasp_darray_cp(mgl[1].A.ROW*mgl[1].A.nb, mgl[1].b.val, mgl[1].w.val);
00374     fasp_blas_dbsr_aAxy(-1.0,&mgl[1].A, mgl[1].x.val, mgl[1].w.val);
00375
00376     // restriction r1 = R*r0
00377     fasp_blas_dbsr_mxv(&mgl[1].R, mgl[1].w.val, mgl[1+1].b.val);
00378
00379     // prepare for the next level
00380     ++l; fasp_dvec_set(mgl[1].A.ROW*mgl[1].A.nb, &mgl[1].x, 0.0);
00381
00382 }
00383
00384 // If AMG only has one level or we have arrived at the coarsest level,
00385 // call the coarse space solver:
00386 switch ( coarse_solver ) {
00387     #if WITH_PARDISO

```

```

00390         case SOLVER_PARDISO: {
00391             /* use Intel MKL PARDISO direct solver on the coarsest level */
00392             fasp_pardiso_solve(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
00393             break;
00394         }
00395     #endif
00396
00397     #if WITH_MUMPS
00398         case SOLVER_MUMPS:
00399             /* use MUMPS direct solver on the coarsest level */
00400             mgl[nl-1].mumps.job = 2;
00401             fasp_solver_mumps_steps(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
00402             break;
00403     #endif
00404
00405     #if WITH_UMFPACK
00406         case SOLVER_UMFPACK:
00407             /* use UMFPACK direct solver on the coarsest level */
00408             fasp_umfpack_solve(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
00409             break;
00410     #endif
00411
00412     #if WITH_SuperLU
00413         case SOLVER_SUPERLU:
00414             /* use SuperLU direct solver on the coarsest level */
00415             fasp_solver_superlu(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, 0);
00416             break;
00417     #endif
00418
00419     default: {
00420         /* use iterative solver on the coarsest level */
00421         const INT csize = mgl[nl-1].A.ROW*mgl[nl-1].A.nb;
00422         const INT cmaxit = MIN(csize*csize, 200); // coarse level iteration number
00423         const REAL ctol = param->tol; // coarse level tolerance
00424         if ( fasp_solver_dbsr_pvgmres(&mgl[nl-1].A,&mgl[nl-1].b,&mgl[nl-1].x,
00425                                     NULL,ctol,cmaxit,25,1,0) < 0 ) {
00426             if ( prtlvl > PRINT_MIN ) {
00427                 printf("### WARNING: Coarse level solver did not converge!\n");
00428                 printf("### WARNING: Consider to increase maxit to %d!\n", 2*cmaxit);
00429             }
00430         }
00431     }
00432 }
00433
00434 // BackwardSweep:
00435 while ( l > 0 ) {
00436     --l;
00437
00438     // prolongation u = u + alpha*P*e1
00439     if ( param->coarse_scaling == ON ) {
00440         dvector PeH, Aeh;
00441         PeH.row = Aeh.row = mgl[l].b.row;
00442         PeH.val = mgl[l].w.val + mgl[l].b.row;
00443         Aeh.val = PeH.val + mgl[l].b.row;
00444
00445         fasp_blas_dbsr_mxv (&mgl[l].P, mgl[l+1].x.val, PeH.val);
00446         fasp_blas_dbsr_mxv (&mgl[l].A, PeH.val, Aeh.val);
00447
00448         alpha = (fasp_blas_darray_dotprod (mgl[l].b.row, Aeh.val, mgl[l].w.val))
00449             / (fasp_blas_darray_dotprod (mgl[l].b.row, Aeh.val, Aeh.val));
00450         alpha = MIN(alpha, 1.0); // Add this for safety! --Chensong on 10/04/2014
00451         fasp_blas_darray_axpy (mgl[l].b.row, alpha, PeH.val, mgl[l].x.val);
00452     }
00453     else {
00454         fasp_blas_dbsr_aAxy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
00455     }
00456
00457     // extra kernel solve
00458     if ( mgl[l].A_nk != NULL ) {
00459         //-----
00460         // extra kernel solve
00461         //-----
00462         // form residual r = b - A x
00463         fasp_darray_cp(mgl[l].A.ROW*mgl[l].A.nb, mgl[l].b.val, mgl[l].w.val);
00464         fasp_blas_dbsr_aAxy(-1.0, &mgl[l].A, mgl[l].x.val, mgl[l].w.val);
00465
00466         // r_nk = R_nk*r
00467         fasp_blas_dcsr_mxv(mgl[l].R_nk, mgl[l].w.val, r_nk.val);
00468
00469         // z_nk = A_nk^{-1}*r_nk
00470     #if WITH_UMFPACK // use UMFPACK directly

```

```

00471         fasp_solver_umfpack(mgl[1].A_nk, &r_nk, &z_nk, 0);
00472 #else
00473         fasp_coarse_itsolver(mgl[1].A_nk, &r_nk, &z_nk, 1e-12, 0);
00474 #endif
00475
00476         // z = z + P_nk*z_nk;
00477         fasp_blas_dcsr_aAxy(1.0, mgl[1].P_nk, z_nk.val, mgl[1].x.val);
00478     }
00479
00480     // post-smoothing
00481     if ( l < mgl->ILU_levels ) {
00482         fasp_smoother_dbsr_ilu(&mgl[1].A, &mgl[1].b, &mgl[1].x, &mgl[1].LU);
00483         for ( i=0; i<steps; i++ )
00484             fasp_smoother_dbsr_gs_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00485                                         mgl[1].diaginv.val);
00486     }
00487     else {
00488         if ( steps > 0 ) {
00489             switch ( smoother ) {
00490                 case SMOOTHER_JACOBI:
00491                     for ( i=0; i<steps; i++ )
00492                         fasp_smoother_dbsr_jacobi1(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00493                                                  mgl[1].diaginv.val);
00494                     break;
00495                 case SMOOTHER_GS:
00496                     for ( i=0; i<steps; i++ )
00497                         fasp_smoother_dbsr_gs_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00498                                                    mgl[1].diaginv.val);
00499                     break;
00500                 case SMOOTHER_SGS:
00501                     for ( i=0; i<steps; i++ ) {
00502                         fasp_smoother_dbsr_gs_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00503                                                    mgl[1].diaginv.val);
00504                         fasp_smoother_dbsr_gs_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00505                                                    mgl[1].diaginv.val);
00506                     }
00507                     break;
00508                 case SMOOTHER_SOR:
00509                     for ( i=0; i<steps; i++ )
00510                         fasp_smoother_dbsr_sor_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00511                                                       mgl[1].diaginv.val, relax);
00512                     break;
00513                 case SMOOTHER_SSOR:
00514                     for ( i=0; i<steps; i++ )
00515                         fasp_smoother_dbsr_sor_ascend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00516                                                    mgl[1].diaginv.val, relax);
00517                     fasp_smoother_dbsr_sor_descend(&mgl[1].A, &mgl[1].b, &mgl[1].x,
00518                                                    mgl[1].diaginv.val, relax);
00519                     break;
00520                 default:
00521                     printf("### ERROR: Unknown smoother type %d!\n", smoother);
00522                     fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
00523             }
00524         }
00525     }
00526
00527     if ( nu_l[1] < cycle_type ) break;
00528     else nu_l[1] = 0;
00529 }
00530
00531 if ( l > 0 ) goto ForwardSweep;
00532
00533 #if DEBUG_MODE > 0
00534     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00535 #endif
00536
00537 }
00538
00539 /*-----*/
00540 /*--          End of File          --*/
00541 /*-----*/

```

9.165 PreMGCycleFull.c File Reference

Abstract non-recursive full multigrid cycle.

```

#include <math.h>
#include <time.h>

```

```
#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"
#include "PreMGSmother.inl"
```

Functions

- void [fasp_solver_fmecycle](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
Solve $Ax=b$ with non-recursive full multigrid K-cycle.

9.165.1 Detailed Description

Abstract non-recursive full multigrid cycle.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaSchwarzSetup.c](#), [BlaArray.c](#), [BlaSpmvCSR.c](#), [BlaVector.c](#), [ItrSmootherCSR.c](#), [ItrSmootherCSRpoly.c](#), [KryPcg.c](#), [KrySPcg.c](#), and [KrySPvgmres.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreMGCycleFull.c](#).

9.165.2 Function Documentation

9.165.2.1 [fasp_solver_fmecycle\(\)](#)

```
void fasp_solver_fmecycle (
    AMG\_data * mgl,
    AMG\_param * param )
```

Solve $Ax=b$ with non-recursive full multigrid K-cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Author

Chensong Zhang

Date

02/27/2011

Modified by Chensong Zhang on 06/01/2012: fix a bug when there is only one level. Modified by Hongxuan Zhang on 12/15/2015: update direct solvers.

Definition at line 47 of file [PreMGCycleFull.c](#).

9.166 PreMGCycleFull.c

[Go to the documentation of this file.](#)

```

00001
00016 #include <math.h>
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 #include "PreMGUtil.inl"
00027 #include "PreMGSmoothing.inl"
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00047 void fasp_solver_fmecycle (AMG_data *mgl,
00048                          AMG_param *param)
00049 {
00050     const SHORT maxit = 3; // Max allowed V-cycles in each level
00051     const SHORT amg_type = param->AMG_type;
00052     const SHORT prtlvl = param->print_level;
00053     const SHORT nl = mgl[0].num_levels;
00054     const SHORT smoother = param->smoother;
00055     const SHORT smooth_order = param->smooth_order;
00056     const SHORT coarse_solver = param->coarse_solver;
00057
00058     const REAL relax = param->relaxation;
00059     const SHORT ndeg = param->polynomial_degree;
00060     const REAL tol = param->tol*1e-4;
00061
00062     // local variables
00063     INT l, i, lvl, num_cycle;
00064     REAL alpha = 1.0, relerr;
00065
00066     // Schwarz parameters
00067     SWZ_param swzparam;
00068     if ( param->SWZ_levels > 0 ) {
00069         swzparam.SWZ_bksolver = param->SWZ_bksolver;
00070     }
00071
00072 #if DEBUG_MODE > 0
00073     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00074     printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
00075 #endif
00076
00077     if ( prtlvl >= PRINT_MOST )
00078         printf("FMG_level = %d, ILU_level = %d\n", nl, param->ILU_levels);
00079
00080     // restriction r1 = R*r0
00081     switch (amg_type) {
00082
00083     case UA_AMG:
00084         for (l=0; l<nl-1; l++)
00085             fasp_blas_dcsr_mnv_agg(&mgl[l].R, mgl[l].b.val, mgl[l+1].b.val);
00086         break;
00087
00088     default:
00089         for (l=0; l<nl-1; l++)
00090             fasp_blas_dcsr_mnv(&mgl[l].R, mgl[l].b.val, mgl[l+1].b.val);
00091         break;
00092     }
00093
00094     fasp_dvec_set(mgl[1].A.row, &mgl[1].x, 0.0); // initial guess
00095
00096     // If only one level, just direct solver
00097     if ( nl==1 ) {
00098
00099         switch (coarse_solver) {
00100
00101         case SOLVER_PARDISO: {
00102             /* use Intel MKL PARDISO direct solver on the coarsest level */
00103             fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
00104
00105         }
00106     }

```

```

00106             break;
00107         }
00108     #endif
00109     #if WITH_SuperLU
00110         case SOLVER_SUPERLU:
00111             /* use SuperLU direct solver on the coarsest level */
00112             fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
00113             break;
00114     #endif
00115     #if WITH_UMFPACK
00116         case SOLVER_UMFPACK:
00117             /* use UMFPACK direct solver on the coarsest level */
00118             fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
00119             break;
00120     #endif
00121     #if WITH_MUMPS
00122         case SOLVER_MUMPS:
00123             /* use MUMPS direct solver on the coarsest level */
00124             mgl[nl-1].mumps.job = 2;
00125             fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
00126             break;
00127     #endif
00128     default:
00129         /* use iterative solver on the coarsest level */
00130         fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);
00131     }
00132     return;
00133 }
00134
00135 for ( i=1; i<nl; i++ ) {
00136     // Coarse Space Solver:
00137     switch (coarse_solver) {
00138     #if WITH_PARDISO
00139         case SOLVER_PARDISO: {
00140             /* use Intel MKL PARDISO direct solver on the coarsest level */
00141             fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
00142             break;
00143         }
00144     #endif
00145     #if WITH_SuperLU
00146         case SOLVER_SUPERLU:
00147             /* use SuperLU direct solver on the coarsest level */
00148             fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
00149             break;
00150     #endif
00151     #if WITH_UMFPACK
00152         case SOLVER_UMFPACK:
00153             /* use UMFPACK direct solver on the coarsest level */
00154             fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
00155             break;
00156     #endif
00157     #if WITH_MUMPS
00158         case SOLVER_MUMPS:
00159             /* use MUMPS direct solver on the coarsest level */
00160             mgl[nl-1].mumps.job = 2;
00161             fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
00162             break;
00163     #endif
00164     default:
00165         /* use iterative solver on the coarsest level */
00166         fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);
00167     }
00168     // Slash part: /-cycle
00169     {
00170         --l; // go back to finer level
00171     }
00172 }

```

```

00187         // find the optimal scaling factor alpha
00188     if ( param->coarse_scaling == ON ) {
00189         alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
00190             / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
00191         alpha = MIN(alpha, 1.0); // Add this for safty! --Chensong on 10/04/2014
00192     }
00193
00194     // prolongation u = u + alpha*P*e1
00195     switch (amg_type) {
00196     case UA_AMG:
00197         fasp_blas_dcsr_aAxy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val); break;
00198     default:
00199         fasp_blas_dcsr_aAxy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val); break;
00200     }
00201 }
00202
00203 // initialzie rel error
00204 num_cycle = 0; relerr = BIGREAL;
00205
00206 while ( relerr > param->tol && num_cycle < maxit) {
00207
00208     ++num_cycle;
00209
00210     // form residual r = b - A x
00211     fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
00212     fasp_blas_dcsr_aAxy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);
00213     relerr = fasp_blas_dvec_norm2(&mgl[l].w) / fasp_blas_dvec_norm2(&mgl[l].b);
00214
00215     // Forward Sweep
00216     for ( lvl=0; lvl<i; lvl++ ) {
00217
00218         // pre smoothing
00219         if (l<param->ILU_levels) {
00220             fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
00221         }
00222         else if (l<mgl->SWZ_levels) {
00223             switch (mgl[l].Schwarz.SWZ_type) {
00224             case SCHWARZ_SYMMETRIC:
00225                 fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00226                 fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
00227                 break;
00228             default:
00229                 fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00230                 break;
00231             }
00232         }
00233
00234         else {
00235             fasp_dcsr_presmoothing(smoother,&mgl[l].A,&mgl[l].b,&mgl[l].x,param->presmooth_iter,
00236                 0,mgl[l].A.row-1,l,relax,ndeg,smooth_order,mgl[l].cfmark.val);
00237         }
00238
00239         // form residual r = b - A x
00240         fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
00241         fasp_blas_dcsr_aAxy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);
00242
00243         // restriction r1 = R*r0
00244         switch (amg_type) {
00245         case UA_AMG:
00246             fasp_blas_dcsr_mxv_agg(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
00247             break;
00248         default:
00249             fasp_blas_dcsr_mxv(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
00250             break;
00251         }
00252
00253         ++l;
00254
00255         // prepare for the next level
00256         fasp_dvec_set(mgl[l].A.row, &mgl[l].x, 0.0);
00257     } // end for lvl
00258
00259     // CoarseSpaceSolver:
00260     switch (coarse_solver) {
00261
00262     #if WITH_PARDISO
00263     case SOLVER_PARDISO: {
00264         /* use Intel MKL PARDISO direct solver on the coarsest level */
00265         fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
00266         break;
00267     }

```



```

00268         }
00269     #endif
00270
00271     #if WITH_SuperLU
00272         case SOLVER_SUPERLU:
00273             /* use SuperLU direct solver on the coarsest level */
00274             fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
00275             break;
00276     #endif
00277
00278     #if WITH_UMFPACK
00279         case SOLVER_UMFPACK:
00280             /* use UMFPACK direct solver on the coarsest level */
00281             fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
00282             break;
00283     #endif
00284
00285     #if WITH_MUMPS
00286         case SOLVER_MUMPS:
00287             /* use MUMPS direct solver on the coarsest level */
00288             mgl[nl-1].mumps.job = 2;
00289             fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
00290             break;
00291     #endif
00292
00293         default:
00294             /* use iterative solver on the coarsest level */
00295             fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);
00296     }
00297
00298     // Backward Sweep
00299     for ( lvl=0; lvl<i; lvl++ ) {
00300
00301         --l;
00302
00303         // find the optimal scaling factor alpha
00304         if ( param->coarse_scaling == ON ) {
00305             alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
00306                 / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
00307             alpha = MIN(alpha, 1.0); // Add this for safty! --Chensong on 10/04/2014
00308         }
00309
00310         // prolongation u = u + alpha*P*e1
00311         switch (amg_type)
00312         {
00313             case UA_AMG:
00314                 fasp_blas_dcsr_aApy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
00315                 break;
00316             default:
00317                 fasp_blas_dcsr_aApy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
00318                 break;
00319         }
00320
00321         // post-smoothing
00322         if (l<param->ILU_levels) {
00323             fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
00324         }
00325         else if (l<mgl->SWZ_levels) {
00326             switch (mgl[l].Schwarz.SWZ_type) {
00327                 case SCHWARZ_SYMMETRIC:
00328                     fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00329                     fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00330                     break;
00331                 default:
00332                     fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00333                     break;
00334             }
00335         }
00336     }
00337
00338     else {
00339         fasp_dcsr_postsMOOTHING(smooth, &mgl[l].A, &mgl[l].b, &mgl[l].x, param->postsMOOTH_iter,
00340         0, mgl[l].A.row-1, -1, relax, ndeg, smooth_order, mgl[l].cfmark.val);
00341     }
00342
00343     } // end while
00344
00345     } //end while
00346
00347     } // end for

```

```

00348
00349 #if DEBUG_MODE > 0
00350     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00351 #endif
00352
00353     return;
00354 }
00355
00356 /*-----*/
00357 /*--      End of File      --*/
00358 /*-----*/

```

9.167 PreMGRecur.c File Reference

Abstract multigrid cycle – recursive version.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"
#include "PreMGSmother.inl"

```

Functions

- void [fasp_solver_mgrecur](#) ([AMG_data](#) *mgl, [AMG_param](#) *param, [INT](#) level)
Solve $Ax=b$ with recursive multigrid K -cycle.

9.167.1 Detailed Description

Abstract multigrid cycle – recursive version.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMessage.c](#), [AuxVector.c](#), [BlaSpmvCSR.c](#), [ItrSmootherCSR.c](#), [ItrSmootherCSRpoly.c](#), [KryPcg.c](#), [KrySPcg.c](#), and [KrySPvgmres.c](#)

Warning

Not used any more! Deprecated in the future versions.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreMGRecur.c](#).

9.167.2 Function Documentation

9.167.2.1 [fasp_solver_mgrecur\(\)](#)

```

void fasp_solver_mgrecur (
    AMG\_data * mgl,
    AMG\_param * param,
    INT level )

```

Solve $Ax=b$ with recursive multigrid K -cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param
<i>level</i>	Index of the current level

Author

Xuehai Huang, Chensong Zhang

Date

04/06/2010

Modified by Chensong Zhang on 02/27/2013: update direct solvers.
Definition at line 47 of file [PreMGRecur.c](#).

9.168 PreMGRecur.c

[Go to the documentation of this file.](#)

```

00001
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_functs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 #include "PreMGUtil.inl"
00027 #include "PreMGSmoother.inl"
00028
00029 /*-----*/
00030 /*--      Public Functions      --*/
00031 /*-----*/
00032
00047 void fasp_solver_mgrecur (AMG_data *mgl,
00048                          AMG_param *param,
00049                          INT level)
00050 {
00051     const SHORT prtlvl = param->print_level;
00052     const SHORT smoother = param->smoother;
00053     const SHORT cycle_type = param->cycle_type;
00054     const SHORT coarse_solver = param->coarse_solver;
00055     const SHORT smooth_order = param->smooth_order;
00056     const REAL relax = param->relaxation;
00057     const REAL tol = param->tol*1e-4;
00058     const SHORT ndeg = param->polynomial_degree;
00059
00060     dvector *b0 = &mgl[level].b, *e0 = &mgl[level].x; // fine level b and x
00061     dvector *b1 = &mgl[level+1].b, *e1 = &mgl[level+1].x; // coarse level b and x
00062
00063     dCSRmat *A0 = &mgl[level].A; // fine level matrix
00064     dCSRmat *A1 = &mgl[level+1].A; // coarse level matrix
00065     const INT m0 = A0->row, m1 = A1->row;
00066
00067     ILU_data *LU_level = &mgl[level].LU; // fine level ILU decomposition
00068     REAL *r = mgl[level].w.val; // for residual
00069     INT *ordering = mgl[level].cfmark.val; // for smoother ordering
00070
00071 #if DEBUG_MODE > 0
00072     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00073     printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
00074 #endif
00075
00076     if ( prtlvl >= PRINT_MOST )
00077         printf("AMG level %d, smoother %d.\n", level, smoother);
00078
00079     if ( level < mgl[level].num_levels-1 ) {

```

```

00080
00081 // pre smoothing
00082 if ( level < mgl[level].ILU_levels ) {
00083     fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
00084 }
00085 else {
00086     fasp_dcsr_presmoothing(smoother,A0,b0,e0,param->presmooth_iter,
00087                           0,m0-1,1,relax,ndeg,smooth_order,ordering);
00088 }
00089
00090 // form residual r = b - A x
00091 fasp_darray_cp(m0,b0->val,r);
00092 fasp_blas_dcsr_aApy(-1.0,A0,e0->val,r);
00093
00094 // restriction r1 = R*r0
00095 fasp_blas_dcsr_mvx(&mgl[level].R, r, b1->val);
00096
00097 { // call MG recursively: type = 1 for V cycle, type = 2 for W cycle
00098     SHORT i;
00099     fasp_dvec_set(m1,e1,0.0);
00100     for (i=0; i<cycle_type; ++i) fasp_solver_mgrecur (mgl, param, level+1);
00101 }
00102
00103 // prolongation e0 = e0 + P*e1
00104 fasp_blas_dcsr_aApy(1.0, &mgl[level].P, e1->val, e0->val);
00105
00106 // post smoothing
00107 if ( level < mgl[level].ILU_levels ) {
00108     fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
00109 }
00110 else {
00111     fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
00112                            0,m0-1,-1,relax,ndeg,smooth_order,ordering);
00113 }
00114 }
00115
00116 else { // coarsest level solver
00117     switch (coarse_solver) {
00118
00119         #if WITH_PARDISO
00120         case SOLVER_PARDISO: {
00121             /* use Intel MKL PARDISO direct solver on the coarsest level */
00122             fasp_pardiso_solve(A0, b0, e0, &mgl[level].pdata, 0);
00123             break;
00124         }
00125         #endif
00126
00127         #if WITH_SuperLU
00128         case SOLVER_SUPERLU: {
00129             /* use SuperLU direct solver on the coarsest level */
00130             fasp_solver_superlu(A0, b0, e0, 0);
00131             break;
00132         }
00133         #endif
00134
00135         #if WITH_UMFPACK
00136         case SOLVER_UMFPACK: {
00137             /* use UMFPACK direct solver on the coarsest level */
00138             fasp_umfpack_solve(A0, b0, e0, mgl[level].Numeric, 0);
00139             break;
00140         }
00141         #endif
00142
00143         #if WITH_MUMPS
00144         case SOLVER_MUMPS: {
00145             /* use MUMPS direct solver on the coarsest level */
00146             mgl[level].mumps.job = 2;
00147             fasp_solver_mumps_steps(A0, b0, e0, &mgl[level].mumps);
00148             break;
00149         }
00150         #endif
00151
00152         /* use iterative solver on the coarsest level */
00153         default: {
00154             fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);
00155         }
00156     }
00157 }
00158
00159 #if DEBUG_MODE > 0
00160 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);

```

```

00161 #endif
00162 }
00163
00164 /*-----*/
00165 /*--      End of File      --*/
00166 /*-----*/

```

9.169 PreMGRecurAMLI.c File Reference

Abstract AMLI multilevel iteration – recursive version.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"
#include "PreMGSmother.inl"
#include "PreMGRecurAMLI.inl"

```

Functions

- void [fasp_solver_amli](#) ([AMG_data](#) *mgl, [AMG_param](#) *param, [INT](#) l)
Solve $Ax=b$ with recursive AMLI-cycle.
- void [fasp_solver_namli](#) ([AMG_data](#) *mgl, [AMG_param](#) *param, [INT](#) l, [INT](#) num_levels)
Solve $Ax=b$ with recursive nonlinear AMLI-cycle.
- void [fasp_solver_namli_bsr](#) ([AMG_data_bsr](#) *mgl, [AMG_param](#) *param, [INT](#) l, [INT](#) num_levels)
Solve $Ax=b$ with recursive nonlinear AMLI-cycle.
- void [fasp_amg_amli_coef](#) (const [REAL](#) lambda_max, const [REAL](#) lambda_min, const [INT](#) degree, [REAL](#) *coef)
Compute the coefficients of the polynomial used by AMLI-cycle.

9.169.1 Detailed Description

Abstract AMLI multilevel iteration – recursive version.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxParam.c](#), [AuxVector.c](#), [BlaSchwarzSetup.c](#), [BlaArray.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), [ltrSmootherBSR.c](#), [ltrSmootherCSR.c](#), [ltrSmootherCSRpoly.c](#), [KryPcg.c](#), [KryPvfgmres.c](#), [KrySPcg.c](#), [KrySPvgmres.c](#), [PreBSR.c](#), and [PreCSR.c](#)

This file includes both AMLI and non-linear AMLI cycles

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreMGRecurAMLI.c](#).

9.169.2 Function Documentation

9.169.2.1 fasp_amg_amli_coef()

```
void fasp_amg_amli_coef (
    const REAL lambda_max,
    const REAL lambda_min,
    const INT degree,
    REAL * coef )
```

Compute the coefficients of the polynomial used by AMLI-cycle.

Parameters

<i>lambda_max</i>	Maximal lambda
<i>lambda_min</i>	Minimal lambda
<i>degree</i>	Degree of polynomial approximation
<i>coef</i>	Coefficient of AMLI (output)

Author

Xiaozhe Hu

Date

01/23/2011

Definition at line 715 of file [PreMGRecurAMLI.c](#).

9.169.2.2 fasp_solver_amli()

```
void fasp_solver_amli (
    AMG_data * mgl,
    AMG_param * param,
    INT l )
```

Solve $Ax=b$ with recursive AMLI-cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param
<i>l</i>	Current level

Author

Xiaozhe Hu

Date

01/23/2011

Note

AMLI polynomial computed by the best approximation of $1/x$. Refer to Johannes K. Kraus, Panayot S. Vassilevski, Ludmil T. Zikatanov, "Polynomial of best uniform approximation to x^{-1} and smoothing in two-level methods", 2013.

Modified by Chensong Zhang on 02/27/2013: update direct solvers. Modified by Zheng Li on 11/10/2014: update direct solvers. Modified by Hongxuan Zhang on 12/15/2015: update direct solvers.
Definition at line 58 of file [PreMGRecurAMLI.c](#).

9.169.2.3 fasp_solver_namli()

```
void fasp_solver_namli (
    AMG_data * mgl,
    AMG_param * param,
    INT l,
    INT num_levels )
```

Solve $Ax=b$ with recursive nonlinear AMLI-cycle.

Parameters

<i>mgl</i>	Pointer to AMG_data data
<i>param</i>	Pointer to AMG parameters
<i>l</i>	Current level
<i>num_levels</i>	Total number of levels

Author

Xiaozhe Hu

Date

04/06/2010

Note

Refer to Xiazhe Hu, Panayot S. Vassilevski, Jinchao Xu "Comparative Convergence Analysis of Nonlinear AMLI-cycle Multigrid", 2013.

Modified by Chensong Zhang on 02/27/2013: update direct solvers. Modified by Zheng Li on 11/10/2014: update direct solvers. Modified by Hongxuan Zhang on 12/15/2015: update direct solvers.
Definition at line 284 of file [PreMGRecurAMLI.c](#).

9.169.2.4 fasp_solver_namli_bsr()

```
void fasp_solver_namli_bsr (
    AMG_data_bsr * mgl,
    AMG_param * param,
    INT l,
    INT num_levels )
```

Solve $Ax=b$ with recursive nonlinear AMLI-cycle.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param
<i>l</i>	Current level
<i>num_levels</i>	Total number of levels

Author

Xiaozhe Hu

Date

04/06/2010

Note

Nonlinear AMLI-cycle. Refer to Xiaozhe Hu, Panayot S. Vassilevski, Jinchao Xu "Comparative Convergence Analysis of Nonlinear AMLI-cycle Multigrid", 2013.

Modified by Chensong Zhang on 02/27/2013: update direct solvers. Modified by Hongxuan Zhang on 12/15/2015: update direct solvers.

Definition at line 517 of file [PreMGRecurAMLI.c](#).

9.170 PreMGRecurAMLI.c

[Go to the documentation of this file.](#)

```

00001
00019 #include <math.h>
00020 #include <time.h>
00021
00022 #include "fasp.h"
00023 #include "fasp_funcs.h"
00024
00025 /*-----*/
00026 /*--  Declare Private Functions  --*/
00027 /*-----*/
00028
00029 #include "PreMGUtil.inl"
00030 #include "PreMGSmother.inl"
00031 #include "PreMGRecurAMLI.inl"
00032
00033 /*-----*/
00034 /*--      Public Functions      --*/
00035 /*-----*/
00036
00058 void fasp_solver_amli (AMG_data  *mgl,
00059                      AMG_param *param,
00060                      INT        l)
00061 {
00062     const SHORT amg_type=param->AMG_type;
00063     const SHORT prtlvl = param->print_level;
00064     const SHORT smoother = param->smoother;
00065     const SHORT smooth_order = param->smooth_order;
00066     const SHORT coarse_solver = param->coarse_solver;
00067     const SHORT degree= param->amli_degree;
00068     const REAL relax = param->relaxation;
00069     const REAL tol = param->tol*1e-4;
00070     const SHORT ndeg = param->polynomial_degree;
00071
00072     // local variables
00073     REAL alpha = 1.0;
00074     REAL *coef = param->amli_coef;
00075
00076     dvector *b0 = &mgl[l].b, *e0 = &mgl[l].x; // fine level b and x
00077     dvector *b1 = &mgl[l+1].b, *e1 = &mgl[l+1].x; // coarse level b and x
00078
00079     dCSRmat *A0 = &mgl[l].A; // fine level matrix
00080     dCSRmat *A1 = &mgl[l+1].A; // coarse level matrix
00081
00082     const INT m0 = A0->row, m1 = A1->row;
00083
00084     INT *ordering = mgl[l].cfmark.val; // smoother ordering
00085     ILU_data *LU_level = &mgl[l].LU; // fine level ILU decomposition
00086     REAL *r = mgl[l].w.val; // work array for residual
00087     REAL *r1 = mgl[l+1].w.val+m1; // work array for residual
00088
00089     // Schwarz parameters
00090     SWZ_param swzparam;
00091     if ( param->SWZ_levels > 0 ) {

```



```

00092     swzparam.SWZ_blksolver = param->SWZ_blksolver;
00093 }
00094
00095 #if DEBUG_MODE > 0
00096     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00097     printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
00098 #endif
00099
00100     if ( prtlvl >= PRINT_MOST )
00101         printf("AMLI level %d, smoother %d.\n", l, smoother);
00102
00103     if ( l < mgl[l].num_levels-1 ) {
00104         // pre smoothing
00105         if ( l < mgl[l].ILU_levels ) {
00106             fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
00107         }
00108     }
00109
00110     else if ( l < mgl->SWZ_levels ) {
00111         switch (mgl[l].Schwarz.SWZ_type) {
00112             case SCHWARZ_SYMMETRIC:
00113                 fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00114                 fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00115                 break;
00116             default:
00117                 fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00118                 break;
00119         }
00120     }
00121
00122     else {
00123         #if MULTI_COLOR_ORDER
00124             // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n", __FUNCTION__, __LINE__);
00125             fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter, 1);
00126         #else
00127             fasp_dcsr_presmoothing(smoother, A0, b0, e0, param->presmooth_iter,
00128                                   0, m0-1, 1, relax, ndeg, smooth_order, ordering);
00129         #endif
00130     }
00131
00132     // form residual r = b - A x
00133     fasp_darray_cp(m0, b0->val, r);
00134     fasp_blas_dcsr_aAxy(-1.0, A0, e0->val, r);
00135
00136     // restriction r1 = R*r0
00137     switch (amg_type) {
00138         case UA_AMG:
00139             fasp_blas_dcsr_mxv_agg(&mgl[l].R, r, b1->val); break;
00140         default:
00141             fasp_blas_dcsr_mxv(&mgl[l].R, r, b1->val); break;
00142     }
00143
00144     // coarse grid correction
00145     {
00146         INT i;
00147
00148         fasp_darray_cp(m1, b1->val, r1);
00149
00150         for ( i=1; i<=degree; i++ ) {
00151             fasp_dvec_set(m1, e1, 0.0);
00152             fasp_solver_amli(mgl, param, l+1);
00153
00154             // b1 = (coef[degree-i]/coef[degree])*r1 + A1*e1;
00155             // First, compute b1 = A1*e1
00156             fasp_blas_dcsr_mxv(A1, e1->val, b1->val);
00157             // Then, compute b1 = b1 + (coef[degree-i]/coef[degree])*r1
00158             fasp_blas_darray_axpy(m1, coef[degree-i]/coef[degree], r1, b1->val);
00159         }
00160
00161         fasp_dvec_set(m1, e1, 0.0);
00162         fasp_solver_amli(mgl, param, l+1);
00163     }
00164
00165     // find the optimal scaling factor alpha
00166     fasp_blas_darray_ax(m1, coef[degree], e1->val);
00167     if ( param->coarse_scaling == ON ) {
00168         alpha = fasp_blas_darray_dotprod(m1, e1->val, r1)
00169                / fasp_blas_dcsr_vmv(A1, e1->val, e1->val);
00170     }

```

```

00173         alpha = MIN(alpha, 1.0);
00174     }
00175
00176     // prolongation e0 = e0 + alpha * P * e1
00177     switch (amg_type) {
00178     case UA_AMG:
00179         fasp_blas_dcsr_aApy_agg(alpha, &mgl[1].P, e1->val, e0->val);
00180         break;
00181     default:
00182         fasp_blas_dcsr_aApy(alpha, &mgl[1].P, e1->val, e0->val);
00183         break;
00184     }
00185
00186     // post smoothing
00187     if ( 1 < mgl[1].ILU_levels ) {
00188
00189         fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
00190
00191     }
00192
00193     else if (l<mgl->SWZ_levels) {
00194
00195         switch (mgl[1].Schwarz.SWZ_type) {
00196         case SCHWARZ_SYMMETRIC:
00197             fasp_dcsr_swz_backward(&mgl[1].Schwarz, &swzparam,&mgl[1].x, &mgl[1].b);
00198             fasp_dcsr_swz_forward(&mgl[1].Schwarz, &swzparam, &mgl[1].x, &mgl[1].b);
00199             break;
00200         default:
00201             fasp_dcsr_swz_backward(&mgl[1].Schwarz, &swzparam,&mgl[1].x, &mgl[1].b);
00202             break;
00203         }
00204     }
00205
00206     else {
00207     #if MULTI_COLOR_ORDER
00208         fasp_smoother_dcsr_gs_multicolor (&mgl[1].x, &mgl[1].A, &mgl[1].b, param->postsmooth_iter,-1);
00209     #else
00210         fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
00211                                0,m0-1,-1,relax,ndeg,smooth_order,ordering);
00212     #endif
00213     }
00214
00215     }
00216
00217     else { // coarsest level solver
00218
00219         switch (coarse_solver) {
00220
00221     #if WITH_PARDISO
00222         case SOLVER_PARDISO: {
00223             /* use Intel MKL PARDISO direct solver on the coarsest level */
00224             fasp_pardiso_solve(A0, b0, e0, &mgl[1].pdata, 0);
00225             break;
00226         }
00227     #endif
00228
00229     #if WITH_SuperLU
00230         case SOLVER_SUPERLU:
00231             /* use SuperLU direct solver on the coarsest level */
00232             fasp_solver_superlu(A0, b0, e0, 0);
00233             break;
00234     #endif
00235
00236     #if WITH_UMFPACK
00237         case SOLVER_UMFPACK:
00238             /* use UMFPACK direct solver on the coarsest level */
00239             fasp_umfpack_solve(A0, b0, e0, mgl[1].Numeric, 0);
00240             break;
00241     #endif
00242
00243     #if WITH_MUMPS
00244         case SOLVER_MUMPS:
00245             /* use MUMPS direct solver on the coarsest level */
00246             mgl[1].mumps.job = 2;
00247             fasp_solver_mumps_steps(A0, b0, e0, &mgl[1].mumps);
00248             break;
00249     #endif
00250
00251         default:
00252             /* use iterative solver on the coarsest level */
00253             fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);

```

```

00254
00255     }
00256
00257     }
00258
00259     #if DEBUG_MODE > 0
00260     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00261     #endif
00262 }
00263
00284 void fasp_solver_namli (AMG_data *mgl,
00285                        AMG_param *param,
00286                        INT l,
00287                        INT num_levels)
00288 {
00289     const SHORT amg_type=param->AMG_type;
00290     const SHORT prtlvl = param->print_level;
00291     const SHORT smoother = param->smoother;
00292     const SHORT smooth_order = param->smooth_order;
00293     const SHORT coarse_solver = param->coarse_solver;
00294     const REAL relax = param->relaxation;
00295     const REAL tol = param->tol*1e-4;
00296     const SHORT ndeg = param->polynomial_degree;
00297
00298     dvector *b0 = &mgl[l].b, *e0 = &mgl[l].x; // fine level b and x
00299     dvector *b1 = &mgl[l+1].b, *e1 = &mgl[l+1].x; // coarse level b and x
00300
00301     dCSRmat *A0 = &mgl[l].A; // fine level matrix
00302     dCSRmat *A1 = &mgl[l+1].A; // coarse level matrix
00303
00304     const INT m0 = A0->row, m1 = A1->row;
00305
00306     INT *ordering = mgl[l].cfmark.val; // smoother ordering
00307     ILU_data *LU_level = &mgl[l].LU; // fine level ILU decomposition
00308     REAL *r = mgl[l].w.val; // work array for residual
00309
00310     dvector uH; // for coarse level correction
00311     uH.row = m1; uH.val = mgl[l+1].w.val + m1;
00312
00313     // Schwarz parameters
00314     SWZ_param swzparam;
00315     if ( param->SWZ_levels > 0 )
00316         swzparam.SWZ_blksolver = param->SWZ_blksolver;
00317
00318     #if DEBUG_MODE > 0
00319     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00320     printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
00321     #endif
00322
00323     if ( prtlvl >= PRINT_MOST )
00324         printf("Nonlinear AMLI level %d, smoother %d.\n", num_levels, smoother);
00325
00326     if ( l < num_levels-1 ) {
00327
00328         // pre smoothing
00329         if ( l < mgl[l].ILU_levels ) {
00330
00331             fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
00332
00333         }
00334
00335         else if ( l < mgl->SWZ_levels ) {
00336
00337             switch (mgl[l].Schwarz.SWZ_type) {
00338                 case SCHWARZ_SYMMETRIC:
00339                     fasp_dcsr_swz_forward (&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00340                     fasp_dcsr_swz_backward (&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00341                     break;
00342                 default:
00343                     fasp_dcsr_swz_forward (&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
00344                     break;
00345             }
00346         }
00347
00348         else {
00349             #if MULTI_COLOR_ORDER
00350                 // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n", __FUNCTION__, __LINE__);
00351                 fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter,1);
00352             #else
00353                 fasp_dcsr_presmoothing(smoother,A0,b0,e0,param->presmooth_iter,
00354                                     0,m0-1,1,relax,ndeg,smooth_order,ordering);
00355             #endif
00356         }
00357     }

```

```

00355 #endif
00356     }
00357
00358     // form residual r = b - A x
00359     fasp_darray_cp(m0,b0->val,r);
00360     fasp_blas_dcsr_aAxy(-1.0,A0,e0->val,r);
00361
00362     // restriction r1 = R*r0
00363     switch (amg_type) {
00364     case UA_AMG:
00365         fasp_blas_dcsr_mxv_agg(&mgl[1].R, r, b1->val);
00366         break;
00367     default:
00368         fasp_blas_dcsr_mxv(&mgl[1].R, r, b1->val);
00369     }
00370
00371     // call nonlinear AMLI-cycle recursively
00372     {
00373         fasp_dvec_set(m1,e1,0.0);
00374
00375         // V-cycle will be enforced when needed !!!
00376         if ( mgl[1+1].cycle_type <= 1 ) {
00377
00378             fasp_solver_namli(&mgl[1+1], param, 0, num_levels-1);
00379
00380         }
00381
00382         else { // recursively call preconditioned Krylov method on coarse grid
00383
00384             precondition_data pcddata;
00385
00386             fasp_param_amg_to_prec(&pcdata, param);
00387             pcddata.maxit = 1;
00388             pcddata.max_levels = num_levels-1;
00389             pcddata.mgl_data = &mgl[1+1];
00390
00391             precondition pc;
00392             pc.data = &pcdata;
00393             pc.fct = fasp_precond_namli;
00394
00395             fasp_darray_cp (m1, e1->val, uH.val);
00396
00397             switch (param->nl_amli_krylov_type) {
00398             case SOLVER_GCG: // Use GCG
00399                 Kcycle_dcsr_pgcg(A1, b1, &uH, &pc);
00400                 break;
00401             default: // Use GCR
00402                 Kcycle_dcsr_pgcr(A1, b1, &uH, &pc);
00403             }
00404
00405             fasp_darray_cp (m1, uH.val, e1->val);
00406         }
00407     }
00408
00409     // prolongation e0 = e0 + P*e1
00410     switch (amg_type) {
00411     case UA_AMG:
00412         fasp_blas_dcsr_aAxy_agg(1.0, &mgl[1].P, e1->val, e0->val);
00413         break;
00414     default:
00415         fasp_blas_dcsr_aAxy(1.0, &mgl[1].P, e1->val, e0->val);
00416     }
00417
00418     // post smoothing
00419     if ( 1 < mgl[1].ILU_levels ) {
00420
00421         fasp_smoothing_dcsr_ilu(A0, b0, e0, LU_level);
00422
00423     }
00424
00425     else if ( 1 < mgl->SWZ_levels ) {
00426
00427         switch (mgl[1].Schwarz.SWZ_type) {
00428         case SCHWARZ_SYMMETRIC:
00429             fasp_dcsr_swz_backward(&mgl[1].Schwarz, &swzparam,&mgl[1].x, &mgl[1].b);
00430             fasp_dcsr_swz_forward(&mgl[1].Schwarz, &swzparam, &mgl[1].x, &mgl[1].b);
00431             break;
00432         default:
00433             fasp_dcsr_swz_backward(&mgl[1].Schwarz, &swzparam,&mgl[1].x, &mgl[1].b);
00434         }
00435     }

```

```

00436     }
00437
00438     else {
00439 #if MULTI_COLOR_ORDER
00440         fasp_smoother_dcsr_gs_multicolor (&mgl[1].x, &mgl[1].A, &mgl[1].b, param->postsmooth_iter,-1);
00441 #else
00442         fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
00443                                0,m0-1,-1,relax,ndeg,smooth_order,ordering);
00444 #endif
00445     }
00446 }
00447 }
00448
00449     else { // coarsest level solver
00450
00451         switch (coarse_solver) {
00452
00453 #if WITH_PARDISO
00454             case SOLVER_PARDISO: {
00455                 /* use Intel MKL PARDISO direct solver on the coarsest level */
00456                 fasp_pardiso_solve(A0, b0, e0, &mgl[1].pdata, 0);
00457                 break;
00458             }
00459 #endif
00460
00461 #if WITH_SuperLU
00462             case SOLVER_SUPERLU:
00463                 /* use SuperLU direct solver on the coarsest level */
00464                 fasp_solver_superlu(A0, b0, e0, 0);
00465                 break;
00466 #endif
00467
00468 #if WITH_UMFPACK
00469             case SOLVER_UMFPACK:
00470                 /* use UMFPACK direct solver on the coarsest level */
00471                 fasp_umfpack_solve(A0, b0, e0, mgl[1].Numeric, 0);
00472                 break;
00473 #endif
00474
00475 #if WITH_MUMPS
00476             case SOLVER_MUMPS:
00477                 /* use MUMPS direct solver on the coarsest level */
00478                 mgl[1].mumps.job = 2;
00479                 fasp_solver_mumps_steps(A0, b0, e0, &mgl[1].mumps);
00480                 break;
00481 #endif
00482
00483             default:
00484                 /* use iterative solver on the coarsest level */
00485                 fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);
00486
00487         }
00488     }
00489 }
00490
00491 #if DEBUG_MODE > 0
00492     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00493 #endif
00494 }
00495
00517 void fasp_solver_namli_bsr (AMG_data_bsr *mgl,
00518                             AMG_param *param,
00519                             INT l,
00520                             INT num_levels)
00521 {
00522     const SHORT prtlvl = param->print_level;
00523     const SHORT smoother = param->smoother;
00524     const SHORT coarse_solver = param->coarse_solver;
00525     const REAL relax = param->relaxation;
00526     const REAL tol = param->tol;
00527     INT i;
00528
00529     dvector *b0 = &mgl[1].b, *e0 = &mgl[1].x; // fine level b and x
00530     dvector *b1 = &mgl[1+1].b, *e1 = &mgl[1+1].x; // coarse level b and x
00531
00532     dBSRmat *A0 = &mgl[1].A; // fine level matrix
00533     dBSRmat *A1 = &mgl[1+1].A; // coarse level matrix
00534     const INT m0 = A0->ROW*A0->nb, m1 = A1->ROW*A1->nb;
00535
00536     ILU_data *LU_level = &mgl[1].LU; // fine level ILU decomposition
00537     REAL *r = mgl[1].w.val; // for residual

```

```

00538
00539     dvector uH, bH; // for coarse level correction
00540     uH.row = m1; uH.val = mgl[l+1].w.val + m1;
00541     bH.row = m1; bH.val = mgl[l+1].w.val + 2*m1;
00542
00543 #if DEBUG_MODE > 0
00544     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00545     printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.ROW, mgl[0].A.NNZ);
00546 #endif
00547
00548     if (prtlvl>=PRINT_MOST)
00549         printf("Nonlinear AMLI: level %d, smoother %d.\n", l, smoother);
00550
00551     if (l < num_levels-1) {
00552
00553         // pre smoothing
00554         if (l<param->ILU_levels) {
00555             fasp_smoother_dbsr_ilu(A0, b0, e0, LU_level);
00556         }
00557         else {
00558             SHORT steps = param->presmooth_iter;
00559
00560             if (steps > 0) {
00561                 switch (smoother) {
00562                     case SMOOTHER_JACOBI:
00563                         for (i=0; i<steps; i++)
00564                             fasp_smoother_dbsr_jacobi (A0, b0, e0);
00565                         break;
00566                     case SMOOTHER_GS:
00567                         for (i=0; i<steps; i++)
00568                             fasp_smoother_dbsr_gs (A0, b0, e0, ASCEND, NULL);
00569                         break;
00570                     case SMOOTHER_SOR:
00571                         for (i=0; i<steps; i++)
00572                             fasp_smoother_dbsr_sor (A0, b0, e0, ASCEND, NULL, relax);
00573                         break;
00574                     default:
00575                         printf("### ERROR: Unknown smoother type %d!\n", smoother);
00576                         fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
00577                 }
00578             }
00579         }
00580
00581         // form residual r = b - A x
00582         fasp_darray_cp(m0, b0->val, r);
00583         fasp_blas_dbsr_aAxy(-1.0, A0, e0->val, r);
00584
00585         fasp_blas_dbsr_mxv(&mgl[l].R, r, b1->val);
00586
00587         // call nonlinear AMLI-cycle recursively
00588         {
00589             fasp_dvec_set(m1, e1, 0.0);
00590
00591             // The coarsest problem is solved exactly.
00592             // No need to call Krylov method on second coarsest level
00593             if (l == num_levels-2) {
00594                 fasp_solver_namli_bsr(&mgl[l+1], param, 0, num_levels-1);
00595             }
00596             else { // recursively call preconditioned Krylov method on coarse grid
00597                 precondition_data_bsr pcddata;
00598
00599                 fasp_param_amg_to_precbsr (&pcdata, param);
00600                 pcddata.maxit = 1;
00601                 pcddata.max_levels = num_levels-1;
00602                 pcddata.mgl_data = &mgl[l+1];
00603
00604                 precondition pc;
00605                 pc.data = &pcdata;
00606                 pc.fct = fasp_precond_dbsr_namli;
00607
00608                 fasp_darray_cp (m1, b1->val, bH.val);
00609                 fasp_darray_cp (m1, e1->val, uH.val);
00610
00611                 const INT maxit = param->amli_degree+1;
00612
00613                 fasp_solver_dbsr_pvfgmres(A1, &bH, &uH, &pc, param->tol,
00614                                         maxit, MIN(maxit, 30), 1, PRINT_NONE);
00615
00616                 fasp_darray_cp (m1, bH.val, b1->val);
00617                 fasp_darray_cp (m1, uH.val, e1->val);
00618             }
00619         }
00620     }

```

```

00619
00620     }
00621
00622     fasp_blas_dbsr_aAxy(1.0, &mgl[1].P, e1->val, e0->val);
00623
00624     // post smoothing
00625     if (l < param->ILU_levels) {
00626         fasp_smoother_dbsr_ilu(A0, b0, e0, LU_level);
00627     }
00628     else {
00629         SHORT steps = param->postsmooth_iter;
00630
00631         if (steps > 0) {
00632             switch (smoother) {
00633                 case SMOOTHER_JACOBI:
00634                     for (i=0; i<steps; i++)
00635                         fasp_smoother_dbsr_jacobi (A0, b0, e0);
00636                     break;
00637                 case SMOOTHER_GS:
00638                     for (i=0; i<steps; i++)
00639                         fasp_smoother_dbsr_gs(A0, b0, e0, ASCEND, NULL);
00640                     break;
00641                 case SMOOTHER_SOR:
00642                     for (i=0; i<steps; i++)
00643                         fasp_smoother_dbsr_sor(A0, b0, e0, ASCEND, NULL, relax);
00644                     break;
00645                 default:
00646                     printf("### ERROR: Unknown smoother type %d!\n", smoother);
00647                     fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
00648             }
00649         }
00650     }
00651 }
00652 }
00653
00654 else { // coarsest level solver
00655
00656     switch (coarse_solver) {
00657
00658 #if WITH_PARDISO
00659         case SOLVER_PARDISO: {
00660             /* use Intel MKL PARDISO direct solver on the coarsest level */
00661             fasp_pardiso_solve(&mgl[1].Ac, b0, e0, &mgl[1].pdata, 0);
00662             break;
00663         }
00664 #endif
00665
00666 #if WITH_SuperLU
00667         case SOLVER_SUPERLU:
00668             /* use SuperLU direct solver on the coarsest level */
00669             fasp_solver_superlu(&mgl[1].Ac, b0, e0, 0);
00670             break;
00671 #endif
00672
00673 #if WITH_UMFPACK
00674         case SOLVER_UMFPACK:
00675             /* use UMFPACK direct solver on the coarsest level */
00676             fasp_umfpack_solve(&mgl[1].Ac, b0, e0, mgl[1].Numeric, 0);
00677             break;
00678 #endif
00679
00680 #if WITH_MUMPS
00681         case SOLVER_MUMPS:
00682             /* use MUMPS direct solver on the coarsest level */
00683             mgl[1].mumps.job = 2;
00684             fasp_solver_mumps_steps(&mgl[1].Ac, b0, e0, &mgl[1].mumps);
00685             break;
00686 #endif
00687
00688         default:
00689             /* use iterative solver on the coarsest level */
00690             fasp_coarse_itsolver(&mgl[1].Ac, b0, e0, tol, prtlvl);
00691     }
00692 }
00693
00694 }
00695
00696 #if DEBUG_MODE > 0
00697     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00698 #endif
00699 }

```

```

00700
00715 void fasp_amg_amli_coef (const REAL  lambda_max,
00716                          const REAL  lambda_min,
00717                          const INT    degree,
00718                          REAL        *coef)
00719 {
00720     const REAL mu0 = 1.0/lambda_max, mu1 = 1.0/lambda_min;
00721     const REAL c = (sqrt(mu0)+sqrt(mu1))*(sqrt(mu0)+sqrt(mu1));
00722     const REAL a = (4*mu0*mu1)/(c);
00723
00724     const REAL kappa = lambda_max/lambda_min; // condition number
00725     const REAL delta = (sqrt(kappa) - 1.0)/(sqrt(kappa)+1.0);
00726     const REAL b = delta*delta;
00727
00728     if (degree == 0) {
00729         coef[0] = 0.5*(mu0+mu1);
00730     }
00731
00732     else if (degree == 1) {
00733         coef[0] = 0.5*c;
00734         coef[1] = -1.0*mu0*mu1;
00735     }
00736
00737     else if (degree > 1) {
00738         INT i;
00739
00740         // allocate memory
00741         REAL *work = (REAL *)fasp_mem_calloc(2*degree-1, sizeof(REAL));
00742         REAL *coef_k, *coef_kml;
00743         coef_k = work; coef_kml = work+degree;
00744
00745         // get q_k
00746         fasp_amg_amli_coef(lambda_max, lambda_min, degree-1, coef_k);
00747         // get q_kml
00748         fasp_amg_amli_coef(lambda_max, lambda_min, degree-2, coef_kml);
00749
00750         // get coef
00751         coef[0] = a - b*coef_kml[0] + (1+b)*coef_k[0];
00752
00753         for (i=1; i<degree-1; i++) {
00754             coef[i] = -b*coef_kml[i] + (1+b)*coef_k[i] - a*coef_k[i-1];
00755         }
00756
00757         coef[degree-1] = (1+b)*coef_k[degree-1] - a*coef_k[degree-2];
00758
00759         coef[degree] = -a*coef_k[degree-1];
00760
00761         // clean memory
00762         fasp_mem_free(work); work = NULL;
00763     }
00764
00765     else {
00766         printf("### ERROR: Wrong AMLI degree %d!\n", degree);
00767         fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
00768     }
00769
00770     return;
00771 }
00772
00773 /*-----*/
00774 /*--      End of File      --*/
00775 /*-----*/

```

9.171 PreMGSolve.c File Reference

Algebraic multigrid iterations: SOLVE phase.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```


Functions

- [INT fasp_amg_solve](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
AMG – SOLVE phase.
- [INT fasp_amg_solve_amli](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
AMLI – SOLVE phase.
- [INT fasp_amg_solve_namli](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
Nonlinear AMLI – SOLVE phase.
- void [fasp_famg_solve](#) ([AMG_data](#) *mgl, [AMG_param](#) *param)
FMG – SOLVE phase.

9.171.1 Detailed Description

Algebraic multigrid iterations: SOLVE phase.

Note

Solve $Ax=b$ using multigrid method. This is SOLVE phase only and is independent of SETUP method used! Should be called after multigrid hierarchy has been generated!

This file contains Level-4 (Pre) functions. It requires: [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSpmvCSR.c](#), [BlaVector.c](#), [PreMGCycle.c](#), [PreMGCycleFull.c](#), and [PreMGRecurAMLI.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreMGSolve.c](#).

9.171.2 Function Documentation

9.171.2.1 fasp_amg_solve()

```
INT fasp_amg_solve (
    AMG_data * mgl,
    AMG_param * param )
```

AMG – SOLVE phase.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

Iteration number if converges; ERROR otherwise.

Author

Xuehai Huang, Chensong Zhang

Date

04/02/2010

Modified by Chensong 04/21/2013: Fix an output typo
Definition at line 49 of file [PreMGSolve.c](#).

9.171.2.2 fasp_amg_solve_amli()

```
INT fasp_amg_solve_amli (
    AMG_data * mgl,
    AMG_param * param )
```

AMLI – SOLVE phase.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

01/23/2011

Modified by Chensong 04/21/2013: Fix an output typo

Note

AMLI polynomial computed by the best approximation of $1/x$. Refer to Johannes K. Kraus, Panayot S. Vassilevski, Ludmil T. Zikatanov, "Polynomial of best uniform approximation to x^{-1} and smoothing in two-level methods", 2013.

Definition at line 142 of file [PreMGSolve.c](#).

9.171.2.3 fasp_amg_solve_namli()

```
INT fasp_amg_solve_namli (
    AMG_data * mgl,
    AMG_param * param )
```

Nonlinear AMLI – SOLVE phase.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

04/30/2011

Modified by Chensong 04/21/2013: Fix an output typo

Note

Nonlinear AMLI-cycle.

Refer to Xiaozhe Hu, Panayot S. Vassilevski, Jinchao Xu "Comparative Convergence Analysis of Nonlinear AMLI-cycle Multigrid", 2013.

Definition at line [230](#) of file [PreMGSolve.c](#).

9.171.2.4 fasp_famg_solve()

```
void fasp_famg_solve (
    AMG_data * mgl,
    AMG_param * param )
```

FMG – SOLVE phase.

Parameters

<i>mgl</i>	Pointer to AMG data: AMG_data
<i>param</i>	Pointer to AMG parameters: AMG_param

Author

Chensong Zhang

Date

01/10/2012

Definition at line [308](#) of file [PreMGSolve.c](#).

9.172 PreMGSolve.c

[Go to the documentation of this file.](#)

```
00001
00019 #include <time.h>
00020
00021 #include "fasp.h"
00022 #include "fasp_funcs.h"
00023
00024 /*-----*/
00025 /*--  Declare Private Functions  --*/
00026 /*-----*/
00027
00028 #include "KryUtil.inl"
00029
```

```

00030 /*-----*/
00031 /*--      Public Functions      --*/
00032 /*-----*/
00033
00049 INT fasp_amg_solve (AMG_data  *mgl,
00050                    AMG_param  *param)
00051 {
00052     dCSRmat      *ptrA = &mgl[0].A;
00053     dvector      *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;
00054
00055     const SHORT   prtlvl = param->print_level;
00056     const INT      MaxIt  = param->maxit;
00057     const REAL     tol    = param->tol;
00058     const REAL     sumb    = fasp_blas_dvec_norm2(b); // L2norm(b)
00059
00060     // local variables
00061     REAL solve_start, solve_end;
00062     REAL relres1 = 1.0, absres0 = sumb, absres, factor;
00063     INT  iter = 0;
00064
00065     #if DEBUG_MODE > 0
00066         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00067         printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
00068             mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00069     #endif
00070
00071     fasp_gettime(&solve_start);
00072
00073     // Print iteration information if needed
00074     fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, sumb, 0.0);
00075
00076     // If b = 0, set x = 0 to be a trivial solution
00077     if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);
00078
00079     // MG solver here
00080     while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) {
00081
00082         #if TRUE
00083             // Call one multigrid cycle -- non recursive version
00084             fasp_solver_mgcycle(mgl, param);
00085         #else
00086             // Call one multigrid cycle -- recursive version
00087             fasp_solver_mgrecur(mgl, param, 0);
00088         #endif
00089
00090         // Form residual r = b - A*x
00091         fasp_dvec_cp(b, r);
00092         fasp_blas_dcsr_aAxy(-1.0, ptrA, x->val, r->val);
00093
00094         // Compute norms of r and convergence factor
00095         absres = fasp_blas_dvec_norm2(r); // residual ||r||
00096         relres1 = absres/MAX(SMALLREAL, sumb); // relative residual ||r||/||b||
00097         factor = absres/absres0; // contraction factor
00098         absres0 = absres; // prepare for next iteration
00099
00100         // Print iteration information if needed
00101         fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, absres, factor);
00102
00103         // Check convergence
00104         if ( relres1 < tol ) break;
00105     }
00106
00107     if ( prtlvl > PRINT_NONE ) {
00108         ITS_FINAL(iter, MaxIt, relres1);
00109         fasp_gettime(&solve_end);
00110         fasp_cputime("AMG solve", solve_end - solve_start);
00111     }
00112
00113     #if DEBUG_MODE > 0
00114         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00115     #endif
00116
00117     if ( iter > MaxIt )
00118         return ERROR_SOLVER_MAXIT;
00119     else
00120         return iter;
00121
00142 INT fasp_amg_solve_amli (AMG_data  *mgl,
00143                        AMG_param  *param)
00144 {
00145     dCSRmat      *ptrA = &mgl[0].A;

```

```

00146     dvector      *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;
00147
00148     const INT     MaxIt  = param->maxit;
00149     const SHORT   prtlvl = param->print_level;
00150     const REAL    tol    = param->tol;
00151     const REAL    sumb    = fasp_blas_dvec_norm2(b); // L2norm(b)
00152
00153     // local variables
00154     REAL          solve_start, solve_end;
00155     REAL          relresl = 1.0, absres0 = sumb, absres, factor;
00156     INT           iter = 0;
00157
00158 #if DEBUG_MODE > 0
00159     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00160     printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
00161           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00162 #endif
00163
00164     fasp_gettime(&solve_start);
00165
00166     // Print iteration information if needed
00167     fasp_itinfo(prtlvl, STOP_REL_RES, iter, relresl, sumb, 0.0);
00168
00169     // If b = 0, set x = 0 to be a trivial solution
00170     if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);
00171
00172     // MG solver here
00173     while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) {
00174
00175         // Call one AMLI cycle
00176         fasp_solver_amli(mgl, param, 0);
00177
00178         // Form residual r = b-A*x
00179         fasp_dvec_cp(b, r);
00180         fasp_blas_dcsr_aAxy(-1.0, ptrA, x->val, r->val);
00181
00182         // Compute norms of r and convergence factor
00183         absres = fasp_blas_dvec_norm2(r); // residual ||r||
00184         relresl = absres/MAX(SMALLREAL, sumb); // relative residual ||r||/||b||
00185         factor = absres/absres0; // contraction factor
00186         absres0 = absres; // prepare for next iteration
00187
00188         // Print iteration information if needed
00189         fasp_itinfo(prtlvl, STOP_REL_RES, iter, relresl, absres, factor);
00190
00191         // Check convergence
00192         if ( relresl < tol ) break;
00193     }
00194
00195     if ( prtlvl > PRINT_NONE ) {
00196         ITS_FINAL(iter, MaxIt, relresl);
00197         fasp_gettime(&solve_end);
00198         fasp_cputime("AMLI solve", solve_end - solve_start);
00199     }
00200
00201 #if DEBUG_MODE > 0
00202     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00203 #endif
00204
00205     if ( iter > MaxIt )
00206         return ERROR_SOLVER_MAXIT;
00207     else
00208         return iter;
00209 }
00210
00230 INT fasp_amg_solve_namli (AMG_data *mgl,
00231                          AMG_param *param)
00232 {
00233     dCSRmat      *ptrA = &mgl[0].A;
00234     dvector      *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;
00235
00236     const INT     MaxIt  = param->maxit;
00237     const SHORT   prtlvl = param->print_level;
00238     const REAL    tol    = param->tol;
00239     const REAL    sumb    = fasp_blas_dvec_norm2(b); // L2norm(b)
00240
00241     // local variables
00242     REAL          solve_start, solve_end;
00243     REAL          relresl = 1.0, absres0 = sumb, absres, factor;
00244     INT           iter = 0;
00245

```

```

00246 #if DEBUG_MODE > 0
00247     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00248     printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
00249           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00250 #endif
00251
00252     fasp_gettime(&solve_start);
00253
00254     // Print iteration information if needed
00255     fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, sumb, 0.0);
00256
00257     // If b = 0, set x = 0 to be a trivial solution
00258     if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);
00259
00260     while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) // MG solver here
00261     {
00262         // one multigrid cycle
00263         fasp_solver_namli(mgl, param, 0, mgl[0].num_levels);
00264
00265         // r = b-A*x
00266         fasp_dvec_cp(b, r);
00267         fasp_blas_dcsr_aAxy(-1.0, ptrA, x->val, r->val);
00268
00269         absres = fasp_blas_dvec_norm2(r); // residual ||r||
00270         relres1 = absres/MAX(SMALLREAL, sumb); // relative residual ||r||/||b||
00271         factor = absres/absres0; // contraction factor
00272
00273         // output iteration information if needed
00274         fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, absres, factor);
00275
00276         if ( relres1 < tol ) break; // early exit condition
00277
00278         absres0 = absres;
00279     }
00280
00281     if ( prtlvl > PRINT_NONE ) {
00282         ITS_FINAL(iter, MaxIt, relres1);
00283         fasp_gettime(&solve_end);
00284         fasp_cputime("Nonlinear AMLI solve", solve_end - solve_start);
00285     }
00286
00287 #if DEBUG_MODE > 0
00288     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00289 #endif
00290
00291     if ( iter > MaxIt )
00292         return ERROR_SOLVER_MAXIT;
00293     else
00294         return iter;
00295 }
00296
00308 void fasp_famg_solve (AMG_data *mgl,
00309                      AMG_param *param)
00310 {
00311     dCSRmat *ptrA = &mgl[0].A;
00312     dvector *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;
00313
00314     const SHORT prtlvl = param->print_level;
00315     const REAL sumb = fasp_blas_dvec_norm2(b); // L2norm(b)
00316
00317     // local variables
00318     REAL solve_start, solve_end;
00319     REAL relres1 = 1.0, absres;
00320
00321 #if DEBUG_MODE > 0
00322     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00323     printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
00324           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
00325 #endif
00326
00327     fasp_gettime(&solve_start);
00328
00329     // If b = 0, set x = 0 to be a trivial solution
00330     if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);
00331
00332     // Call one full multigrid cycle
00333     fasp_solver_fmecycle(mgl, param);
00334
00335     // Form residual r = b-A*x
00336     fasp_dvec_cp(b, r);
00337     fasp_blas_dcsr_aAxy(-1.0, ptrA, x->val, r->val);

```

```

00338
00339 // Compute norms of r and convergence factor
00340 absres = fasp_blas_dvec_norm2(r); // residual ||r||
00341 relres1 = absres/MAX(SMALLREAL, sumb); // relative residual ||r||/||b||
00342
00343 if ( prtlvl > PRINT_NONE ) {
00344     printf("FMG finishes with relative residual %e.\n", relres1);
00345     fasp_gettime(&solve_end);
00346     fasp_cputime("FMG solve", solve_end - solve_start);
00347 }
00348
00349 #if DEBUG_MODE > 0
00350 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00351 #endif
00352
00353 return;
00354 }
00355
00356 /*-----*/
00357 /*--      End of File      --*/
00358 /*-----*/

```

9.173 PreSTR.c File Reference

Preconditioners for [dSTRmat](#) matrices.

```

#include <math.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void [fasp_precond_dstr_diag](#) (REAL *r, REAL *z, void *data)
*Diagonal preconditioner $z = \text{inv}(D) * r$.*
- void [fasp_precond_dstr_ilu0](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(0) decomposition.
- void [fasp_precond_dstr_ilu1](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(1) decomposition.
- void [fasp_precond_dstr_ilu0_forward](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(0) decomposition: $Lz = r$.
- void [fasp_precond_dstr_ilu0_backward](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(0) decomposition: $Uz = r$.
- void [fasp_precond_dstr_ilu1_forward](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(1) decomposition: $Lz = r$.
- void [fasp_precond_dstr_ilu1_backward](#) (REAL *r, REAL *z, void *data)
Preconditioning using STR_ILU(1) decomposition: $Uz = r$.
- void [fasp_precond_dstr_blockgs](#) (REAL *r, REAL *z, void *data)
CPR-type preconditioner (STR format)

9.173.1 Detailed Description

Preconditioners for [dSTRmat](#) matrices.

Note

This file contains Level-4 (Pre) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxVector.c](#), [BlaSmallMat.c](#), [BlaArray.c](#), and [ltrSmootherSTR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [PreSTR.c](#).

9.173.2 Function Documentation

9.173.2.1 fasp_precond_dstr_blockgs()

```
void fasp_precond_dstr_blockgs (
    REAL * r,
    REAL * z,
    void * data )
```

CPR-type preconditioner (STR format)

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

10/17/2010

Definition at line 1715 of file [PreSTR.c](#).

9.173.2.2 fasp_precond_dstr_diag()

```
void fasp_precond_dstr_diag (
    REAL * r,
    REAL * z,
    void * data )
```

Diagonal preconditioner $z = \text{inv}(D) * r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/06/2010

Definition at line 44 of file [PreSTR.c](#).**9.173.2.3 fasp_precond_dstr_ilu0()**

```
void fasp_precond_dstr_ilu0 (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Preconditioning using STR_ILU(0) decomposition.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/21/2010

Definition at line 71 of file [PreSTR.c](#).**9.173.2.4 fasp_precond_dstr_ilu0_backward()**

```
void fasp_precond_dstr_ilu0_backward (  
    REAL * r,  
    REAL * z,  
    void * data )
```

Preconditioning using STR_ILU(0) decomposition: $Uz = r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

06/07/2010

Definition at line 987 of file [PreSTR.c](#).

9.173.2.5 fasp_precond_dstr_ilu0_forward()

```
void fasp_precond_dstr_ilu0_forward (
    REAL * r,
    REAL * z,
    void * data )
```

Preconditioning using STR_ILU(0) decomposition: $Lz = r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

06/07/2010

Definition at line 824 of file [PreSTR.c](#).

9.173.2.6 fasp_precond_dstr_ilu1()

```
void fasp_precond_dstr_ilu1 (
    REAL * r,
    REAL * z,
    void * data )
```

Preconditioning using STR_ILU(1) decomposition.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/21/2010

Definition at line 349 of file [PreSTR.c](#).

9.173.2.7 fasp_precond_dstr_ilu1_backward()

```
void fasp_precond_dstr_ilu1_backward (
    REAL * r,
```

```

    REAL * z,
    void * data )

```

Preconditioning using STR_ILU(1) decomposition: $Uz = r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/21/2010

Definition at line 1434 of file [PreSTR.c](#).

9.173.2.8 fasp_precond_dstr_ilu1_forward()

```

void fasp_precond_dstr_ilu1_forward (
    REAL * r,
    REAL * z,
    void * data )

```

Preconditioning using STR_ILU(1) decomposition: $Lz = r$.

Parameters

<i>r</i>	Pointer to the vector needs preconditioning
<i>z</i>	Pointer to preconditioned vector
<i>data</i>	Pointer to precondition data

Author

Shiquan Zhang

Date

04/21/2010

Definition at line 1168 of file [PreSTR.c](#).

9.174 PreSTR.c

[Go to the documentation of this file.](#)

```

00001
00015 #include <math.h>
00016
00017 #include "fasp.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/

```

```

00021 /*-- Declare Private Functions --*/
00022 /*-----*/
00023
00024 static inline void fasp_darray_cp_nc3 (const REAL *x, REAL *y);
00025 static inline void fasp_darray_cp_nc5 (const REAL *x, REAL *y);
00026 static inline void fasp_darray_cp_nc7 (const REAL *x, REAL *y);
00027
00028 /*-----*/
00029 /*-- Public Functions --*/
00030 /*-----*/
00031
00044 void fasp_precond_dstr_diag (REAL *r,
00045                             REAL *z,
00046                             void *data)
00047 {
00048     const precondition_diag_str *diag = (precondition_diag_str *)data;
00049     const REAL *diagptr = diag->diag.val;
00050     const INT nc = diag->nc, nc2 = nc*nc;
00051     const INT m = diag->diag.row/nc2;
00052
00053     INT i;
00054     for ( i=0; i<m; ++i ) {
00055         fasp_blas_smat_mnv(&(diagptr[i*nc2]), &(r[i*nc]), &(z[i*nc]), nc);
00056     }
00057 }
00058
00071 void fasp_precond_dstr_ilu0 (REAL *r,
00072                             REAL *z,
00073                             void *data)
00074 {
00075     INT i, ic, ic2;
00076     REAL *zz, *zr, *tc;
00077     INT nline, nplane;
00078
00079     dSTRmat *ILU_data=(dSTRmat *)data;
00080     INT m=ILU_data->ngrid;
00081     INT nc=ILU_data->nc;
00082     INT nc2=nc*nc;
00083     INT nx=ILU_data->nx;
00084     INT ny=ILU_data->ny;
00085     INT nz=ILU_data->nz;
00086     INT nxy=ILU_data->nxy;
00087     INT size=m*nc;
00088
00089     #if DEBUG_MODE > 0
00090     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00091     #endif
00092
00093     if (nx == 1) {
00094         nline = ny;
00095         nplane = m;
00096     }
00097     else if (ny == 1) {
00098         nline = nx;
00099         nplane = m;
00100     }
00101     else if (nz == 1) {
00102         nline = nx;
00103         nplane = m;
00104     }
00105     else {
00106         nline = nx;
00107         nplane = nxy;
00108     }
00109
00110     tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));
00111
00112     zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));
00113
00114     zr=(REAL*)fasp_mem_calloc(size, sizeof(REAL));
00115
00116     // copy residual r to zr, to save r
00117     memcpy(zr,r,(size)*sizeof(REAL));
00118
00119     if (nc == 1) {
00120         // forward sweep: solve unit lower matrix equation L*zz=zr
00121         zz[0]=zr[0];
00122         for (i=1; i<m; ++i) {
00123             zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00124             if (i>=nline) zz[i]=zz[i]-ILU_data->offdiag[2][i-nline]*zz[i-nline];
00125             if (i>=nplane) zz[i]=zz[i]-ILU_data->offdiag[4][i-nplane]*zz[i-nplane];

```

```

00126     }
00127
00128     // backward sweep: solve upper matrix equation U*z=zz
00129     z[m-1]=zz[m-1]*ILU_data->diag[m-1];
00130     for (i=m-2;i>=0;i--) {
00131         zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00132         if (i<m-nline) zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline];
00133         if (i<m-nplane) zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nplane];
00134         z[i]=zz[i]*ILU_data->diag[i];
00135     }
00136
00137 } // end if (nc == 1)
00138
00139 else if (nc == 3) {
00140     // forward sweep: solve unit lower matrix equation L*zz=zr
00141     fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));
00142
00143     for (i=1;i<m;++i) {
00144         ic=i*nc;
00145
00146         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00147         fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00148         if (i>=nline) {
00149             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00150             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00151         }
00152         if (i>=nplane) {
00153             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00154             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00155         }
00156         fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
00157     } // end for (i=1;i<m;++i)
00158
00159     // backward sweep: solve upper matrix equation U*z=zz
00160     fasp_blas_smat_mnv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00161
00162     for (i=m-2;i>=0;i--) {
00163         ic=i*nc;
00164         ic2=i*nc2;
00165
00166         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00167         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00168
00169         if (i<m-nline) {
00170             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
00171             fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00172         }
00173
00174         if (i<m-nplane) {
00175             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
00176             fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00177         }
00178
00179         fasp_blas_smat_mnv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00180     } // end for for (i=m-2;i>=0;i--)
00181
00182 } // end else if (nc == 3)
00183
00184 else if (nc == 5) {
00185     // forward sweep: solve unit lower matrix equation L*zz=zr
00186     fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));
00187
00188     for (i=1;i<m;++i) {
00189         ic=i*nc;
00190
00191         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00192         fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00193         if (i>=nline) {
00194             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00195             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00196         }
00197         if (i>=nplane) {
00198             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00199             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00200         }
00201         fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
00202     } // end for (i=1;i<m;++i)
00203
00204     // backward sweep: solve upper matrix equation U*z=zz
00205     fasp_blas_smat_mnv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00206

```

```

00207
00208     for (i=m-2;i>=0;i--) {
00209
00210         ic=i*nc;
00211         ic2=i*nc2;
00212
00213         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00214         fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00215
00216         if (i<m-nline) {
00217             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
00218             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00219         }
00220
00221         if (i<m-nplane) {
00222             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
00223             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00224         }
00225
00226         fasp_blas_smat_mnv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00227     } // end for for (i=m-2;i>=0;i--)
00228
00229 } // end else if (nc == 5)
00230
00231
00232 else if (nc == 7) {
00233     // forward sweep: solve unit lower matrix equation L*zz=zr
00234     fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));
00235
00236     for (i=1;i<m;++i) {
00237         ic=i*nc;
00238
00239         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00240         fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00241         if (i>=nline) {
00242             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00243             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00244         }
00245         if (i>=nplane) {
00246             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00247             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00248         }
00249         fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
00250     } // end for (i=1;i<m;++i)
00251
00252     // backward sweep: solve upper matrix equation U+z=zz
00253     fasp_blas_smat_mnv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00254
00255     for (i=m-2;i>=0;i--) {
00256
00257         ic=i*nc;
00258         ic2=i*nc2;
00259
00260         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00261         fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00262
00263         if (i<m-nline) {
00264             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
00265             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00266         }
00267
00268         if (i<m-nplane) {
00269             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
00270             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00271         }
00272
00273         fasp_blas_smat_mnv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00274     } // end for for (i=m-2;i>=0;i--)
00275
00276 } // end else if (nc == 7)
00277
00278 else {
00279     // forward sweep: solve unit lower matrix equation L*zz=zr
00280     fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
00281     for (i=1;i<m;++i) {
00282         ic=i*nc;
00283
00284         fasp_blas_smat_mnv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
00285         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00286
00287         if (i>=nline) {

```

```

00288         fasp_blas_smat_mnv(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
00289         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00290     }
00291
00292     if (i>=nplane) {
00293         fasp_blas_smat_mnv(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
00294         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00295     }
00296
00297     fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
00298
00299 } // end for (i=1; i<m; ++i)
00300
00301 // backward sweep: solve upper matrix equation U*z=zz
00302 fasp_blas_smat_mnv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);
00303
00304 for (i=m-2;i>=0;i--) {
00305     ic=i*nc;
00306     ic2=i*nc2;
00307
00308     fasp_blas_smat_mnv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
00309     fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
00310
00311     if (i<m-nline) {
00312         fasp_blas_smat_mnv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc,nc);
00313         fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
00314     }
00315
00316     if (i<m-nplane) {
00317         fasp_blas_smat_mnv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc,nc);
00318         fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
00319     }
00320
00321     fasp_blas_smat_mnv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);
00322
00323 } // end for (i=m-2;i>=0;i--)
00324 } // end else
00325
00326 fasp_mem_free(zr); zr = NULL;
00327 fasp_mem_free(zz); zz = NULL;
00328 fasp_mem_free(tc); tc = NULL;
00329
00330 #if DEBUG_MODE > 0
00331     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00332 #endif
00333
00334     return;
00335 }
00336
00349 void fasp_precond_dstr_ilu1 (REAL *r,
00350                             REAL *z,
00351                             void *data)
00352 {
00353     REAL *zz,*zr,*tc;
00354
00355     dSTRmat *ILU_data=(dSTRmat *)data;
00356     INT i,ic, ic2;
00357     INT m=ILU_data->ngrid;
00358     INT nc=ILU_data->nc;
00359     INT nc2=nc*nc;
00360     INT nx=ILU_data->nx;
00361     INT ny=ILU_data->ny;
00362     INT nz=ILU_data->nz;
00363     INT nxy=ILU_data->nxy;
00364     INT size=m*nc;
00365     INT nline, nplane;
00366
00367     if (nx == 1) {
00368         nline = ny;
00369         nplane = m;
00370     }
00371     else if (ny == 1) {
00372         nline = nx;
00373         nplane = m;
00374     }
00375     else if (nz == 1) {
00376         nline = nx;
00377         nplane = m;
00378     }
00379     else {
00380         nline = nx;

```

```

00381     nplane = nxy;
00382 }
00383
00384 tc=(REAL*) fasp_mem_calloc(nc, sizeof(REAL));
00385
00386 zz=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
00387
00388 zr=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
00389
00390 // copy residual r to zr, to save r
00391 for (i=0;i<size;++i) zr[i]=r[i];
00392 if (nc == 1) {
00393     // forward sweep: solve unit lower matrix equation L*zz=zr
00394     zz[0]=zr[0];
00395     for (i=1;i<m;++i) {
00396
00397         zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00398         if (i>=nline-1)
00399             zz[i]=zz[i]-ILU_data->offdiag[2][i-nline+1]*zz[i-nline+1];
00400
00401         if (i>=nline)
00402             zz[i]=zz[i]-ILU_data->offdiag[4][i-nline]*zz[i-nline];
00403         if (i>=nplane-nline)
00404             zz[i]=zz[i]-ILU_data->offdiag[6][i-nplane+nline]*zz[i-nplane+nline];
00405         if (i>=nplane-1)
00406             zz[i]=zz[i]-ILU_data->offdiag[8][i-nplane+1]*zz[i-nplane+1];
00407         if (i>=nplane)
00408             zz[i]=zz[i]-ILU_data->offdiag[10][i-nplane]*zz[i-nplane];
00409     }
00410
00411     // backward sweep: solve upper matrix equation U*z=zz
00412
00413     z[m-1]=zz[m-1]*ILU_data->diag[m-1];
00414     for (i=m-2;i>=0;i--) {
00415
00416         zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00417         if (i+nline-1<m)
00418             zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline-1];
00419         if (i+nline<m)
00420             zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nline];
00421         if (i+nplane-nline<m)
00422             zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nplane-nline];
00423         if (i+nplane-1<m)
00424             zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nplane-1];
00425         if (i+nplane<m)
00426             zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nplane];
00427
00428         z[i]=ILU_data->diag[i]*zz[i];
00429     }
00430 }
00431 // end if (nc == 1)
00432
00433 else if (nc == 3) {
00434
00435     // forward sweep: solve unit lower matrix equation L*zz=zr
00436     fasp_darray_cp_nc3(&(zr[0]), &(zz[0]));
00437
00438     for (i=1;i<m;++i) {
00439         ic=i*nc;
00440
00441         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00442         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]), &(zz[(i-1)*nc]), tc);
00443         fasp_blas_darray_axpy_nc3(-1, tc, &(zr[ic]));
00444
00445         if (i>=nline-1) {
00446             //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
00447             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[2][(i-nline+1)*nc2]), &(zz[(i-nline+1)*nc]), tc);
00448
00449             fasp_blas_darray_axpy_nc3(-1, tc, &(zr[ic]));
00450         }
00451
00452         if (i>=nline) {
00453             //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
00454             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[4][(i-nline)*nc2]), &(zz[(i-nline)*nc]), tc);
00455
00456             fasp_blas_darray_axpy_nc3(-1, tc, &(zr[ic]));
00457         }
00458
00459         if (i>=nplane-nline) {
00460             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];

```



```

00460     fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

00461         fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00462     }
00463     if (i>=nplane-1) {
00464         // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
00465     }
00466     fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
00467     fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00468 }
00469 if (i>=nplane) {
00470     //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
00471     fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00472     fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00473 }
00474 }
00475 fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
00476 }
00477 // backward sweep: solve upper matrix equation U*z=zz
00478 // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
00479 fasp_blas_smat_mnv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00480 for (i=m-2;i>=0;i--) {
00481     ic=i*nc;
00482     ic2=ic+nc;
00483     //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00484     fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00485     fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00486     if (i+nline-1<m) {
00487         //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
00488         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
00489         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00490     }
00491     if (i+nline<m) {
00492         //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
00493         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
00494         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00495     }
00496     if (i+nplane-nline<m) {
00497         //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
00498         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
00499         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00500     }
00501     if (i+nplane<m) {
00502         //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
00503         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
00504         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
00505     }
00506     //z[i]=ILU_data->diag[i]*zz[i];
00507     fasp_blas_smat_mnv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00508 } // end for (i=m-2;i>=0;i--)
00509 } // end if (nc == 3)
00510 else if (nc == 5) {
00511     // forward sweep: solve unit lower matrix equation L*z=zz
00512     fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));
00513     for (i=1;i<m;++i) {
00514         ic=i*nc;
00515     }

```

```

00536         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00537         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00538         fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00539
00540         if (i>=nline-1) {
00541             //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
00542             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
00543
00544             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00545         }
00546         if (i>=nline) {
00547             //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
00548             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00549
00550             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00551         }
00552         if (i>=nplane-nline) {
00553             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
00554             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
00555
00556             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00557         }
00558         if (i>=nplane-1) {
00559             // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
00560             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
00561             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00562         }
00563
00564         if (i>=nplane) {
00565             //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
00566             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00567
00568             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00569         }
00570         fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
00571     }
00572
00573     // backward sweep: solve upper matrix equation U*z=zz
00574
00575     // z[m-1]=zz[m-1]+ILU_data->diag[m-1];
00576     fasp_blas_smat_mnv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00577
00578     for (i=m-2;i>=0;i--) {
00579         ic=i*nc;
00580         ic2=ic+nc;
00581
00582         //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00583         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00584         fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00585
00586         if (i+nline-1<m) {
00587             //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
00588             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
00589             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00590         }
00591
00592         if (i+nline<m) {
00593             //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
00594             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
00595             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00596         }
00597
00598         if (i+nplane-nline<m) {
00599             //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
00600             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
00601
00602             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00603         }
00604
00605         if (i+nplane-1<m) {
00606             //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
00607             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
00608             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00609         }

```

```

00610         if (i+nplane<m) {
00611             //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
00612             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
00613             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
00614         }
00615
00616         //z[i]=ILU_data->diag[i]*zz[i];
00617         fasp_blas_smat_mnv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00618     } // end for (i=m-2;i>=0;i--)
00619 } // end if (nc == 5)
00620
00621
00622 else if (nc == 7) {
00623
00624     // forward sweep: solve unit lower matrix equation L*zz=zr
00625     fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));
00626
00627     for (i=1;i<m;++i) {
00628         ic=i*nc;
00629
00630         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00631         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00632         fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00633
00634         if (i>=nline-1) {
00635             //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
00636             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
00637             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00638         }
00639
00640         if (i>=nline) {
00641             //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
00642             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00643             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00644         }
00645
00646         if (i>=nplane-nline) {
00647             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
00648             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
00649             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00650         }
00651
00652         if (i>=nplane-1) {
00653             // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
00654             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
00655             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00656         }
00657
00658         if (i>=nplane) {
00659             //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
00660             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00661             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00662         }
00663
00664         fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
00665     }
00666
00667     // backward sweep: solve upper matrix equation U*z=zz
00668
00669     // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
00670     fasp_blas_smat_mnv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
00671
00672     for (i=m-2;i>=0;i--) {
00673         ic=i*nc;
00674         ic2=ic+nc;
00675
00676         //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00677         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
00678         fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00679
00680         if (i+nline-1<m) {
00681             //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
00682             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
00683             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00684         }

```

```

00685         if (i+nline<m) {
00686             //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
00687             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
00688             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00689         }
00690     }
00691     if (i+nplane-nline<m) {
00692         //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
00693         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
00694     }
00695     fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00696 }
00697
00698 if (i+nplane-1<m) {
00699     //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
00700     fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
00701     fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00702 }
00703
00704 if (i+nplane<m) {
00705     //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
00706     fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
00707     fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
00708 }
00709
00710 //z[i]=ILU_data->diag[i]*zz[i];
00711 fasp_blas_smat_mnv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
00712 } // end for (i=m-2;i>=0;i--)
00713
00714 } // end if (nc == 7)
00715
00716 else {
00717     // forward sweep: solve unit lower matrix equation L*zz=zr
00718     fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
00719
00720     for (i=1;i<m;++i) {
00721         ic=i*nc;
00722         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00723         fasp_blas_smat_mnv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(z[(i-1)*nc]),tc,nc);
00724         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00725
00726         if (i>=nline-1) {
00727             //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
00728             fasp_blas_smat_mnv(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(z[(i-nline+1)*nc]),tc,nc);
00729             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00730         }
00731
00732         if (i>=nplane) {
00733             //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
00734             fasp_blas_smat_mnv(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(z[(i-nline)*nc]),tc,nc);
00735             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00736         }
00737
00738         if (i>=nplane-nline) {
00739             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
00740             fasp_blas_smat_mnv(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(z[(i-nplane+nline)*nc]),tc,nc);
00741             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00742         }
00743
00744         if (i>=nplane-1) {
00745             // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
00746             fasp_blas_smat_mnv(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(z[(i-nplane+1)*nc]),tc,nc);
00747             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00748         }
00749
00750         if (i>=nplane) {
00751             //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
00752             fasp_blas_smat_mnv(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(z[(i-nplane)*nc]),tc,nc);
00753             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00754         }
00755         fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
00756     }
00757
00758     // backward sweep: solve upper matrix equation U*z=zz

```

```

00759
00760 // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
00761 fasp_blas_smat_mxv(&(ILU_data->diag[(m-1)*nc2]),&(z[(m-1)*nc]),&(z[(m-1)*nc]),nc);
00762
00763 for (i=m-2;i>=0;i--) {
00764     ic=i*nc;
00765     ic2=ic*nc;
00766     //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
00767     fasp_blas_smat_mxv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
00768     fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00769
00770     if (i+nline-1<m) {
00771         //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
00772         fasp_blas_smat_mxv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc,nc);
00773         fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00774     }
00775
00776     if (i+nline<m) {
00777         //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
00778         fasp_blas_smat_mxv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc,nc);
00779         fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00780     }
00781
00782     if (i+nplane-nline<m) {
00783         //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
00784         fasp_blas_smat_mxv(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc,nc);
00785         fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00786     }
00787
00788     if (i+nplane-1<m) {
00789         //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
00790         fasp_blas_smat_mxv(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc,nc);
00791         fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00792     }
00793
00794     if (i+nplane<m) {
00795         //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
00796         fasp_blas_smat_mxv(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc,nc);
00797         fasp_blas_darray_axpy(nc,-1,tc,&(z[ic]));
00798     }
00799
00800     //z[i]=ILU_data->diag[i]*zz[i];
00801     fasp_blas_smat_mxv(&(ILU_data->diag[ic2]),&(z[ic]),&(z[ic]),nc);
00802 }
00803 } // end else
00804
00805 fasp_mem_free(zr); zr = NULL;
00806 fasp_mem_free(zz); zz = NULL;
00807 fasp_mem_free(tc); tc = NULL;
00808
00809 return;
00810 }
00811
00824 void fasp_precond_dstr_ilu0_forward (REAL *r,
00825                                     REAL *z,
00826                                     void *data)
00827 {
00828     INT i, ic;
00829     REAL *zz,*zr,*tc;
00830     INT nline, nplane;
00831
00832     dSTRmat *ILU_data=(dSTRmat *)data;
00833     INT m=ILU_data->ngrid;
00834     INT nc=ILU_data->nc;
00835     INT nc2=nc*nc;
00836     INT nx=ILU_data->nx;
00837     INT ny=ILU_data->ny;
00838     INT nz=ILU_data->nz;
00839     INT nxy=ILU_data->nxy;
00840     INT size=m*nc;
00841
00842     if (nx == 1) {
00843         nline = ny;
00844         nplane = m;
00845     }
00846     else if (ny == 1) {
00847         nline = nx;
00848         nplane = m;
00849     }
00850     else if (nz == 1) {
00851         nline = nx;

```

```

00852     nplane = m;
00853 }
00854 else {
00855     nline = nx;
00856     nplane = nxy;
00857 }
00858
00859 tc=(REAL*) fasp_mem_calloc(nc, sizeof(REAL));
00860
00861 zz=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
00862
00863 zr=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
00864
00865 // copy residual r to zr, to save r
00866 memcpy(zr,r,(size)*sizeof(REAL));
00867 if (nc == 1) {
00868     // forward sweep: solve unit lower matrix equation L*zz=zr
00869     zz[0]=zr[0];
00870     for (i=1;i<m;++i) {
00871         zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
00872         if (i>=nline) zz[i]=zz[i]-ILU_data->offdiag[2][i-nline]*zz[i-nline];
00873         if (i>=nplane) zz[i]=zz[i]-ILU_data->offdiag[4][i-nplane]*zz[i-nplane];
00874     }
00875 } // end if (nc == 1)
00876
00877 else if (nc == 3) {
00878     // forward sweep: solve unit lower matrix equation L*zz=zr
00879     fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));
00880
00881     for (i=1;i<m;++i) {
00882         ic=i*nc;
00883         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00884         fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00885         if (i>=nline) {
00886             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00887             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00888         }
00889         if (i>=nplane) {
00890             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00891             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
00892         }
00893         fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
00894     } // end for (i=1;i<m;++i)
00895
00896 } // end else if (nc == 3)
00897
00898 else if (nc == 5) {
00899     // forward sweep: solve unit lower matrix equation L*zz=zr
00900     fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));
00901
00902     for (i=1;i<m;++i) {
00903         ic=i*nc;
00904         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00905         fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00906         if (i>=nline) {
00907             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00908             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00909         }
00910         if (i>=nplane) {
00911             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00912             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
00913         }
00914         fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
00915     } // end for (i=1;i<m;++i)
00916
00917 } // end else if (nc == 5)
00918
00919
00920 else if (nc == 7) {
00921     // forward sweep: solve unit lower matrix equation L*zz=zr
00922     fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));
00923
00924     for (i=1;i<m;++i) {
00925         ic=i*nc;
00926         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
00927         fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00928         if (i>=nline) {
00929             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
00930             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00931         }
00932         if (i>=nplane) {

```

```

00933         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
00934         fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
00935     }
00936     fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
00937 } // end for (i=1;i<m;++i)
00938
00939 } // end else if (nc == 7)
00940
00941
00942 else {
00943     // forward sweep: solve unit lower matrix equation L*zz=zr
00944     fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
00945     for (i=1;i<m;++i) {
00946         ic=i*nc;
00947         fasp_blas_smat_mnv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
00948         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00949
00950         if (i>=nline) {
00951             fasp_blas_smat_mnv(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
00952             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00953         }
00954
00955         if (i>=nplane) {
00956             fasp_blas_smat_mnv(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
00957             fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
00958         }
00959
00960         fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
00961
00962     } // end for (i=1; i<m; ++i)
00963 } // end else
00964
00965 memcpy(z,zz,(size)*sizeof(REAL));
00966
00967 fasp_mem_free(zr); zr = NULL;
00968 fasp_mem_free(zz); zz = NULL;
00969 fasp_mem_free(tc); tc = NULL;
00970
00971 return;
00972 }
00973 }
00974
00987 void fasp_precond_dstr_ilu0_backward (REAL *r,
00988                                     REAL *z,
00989                                     void *data)
00990 {
00991     INT i, ic, ic2;
00992     REAL *zz,*tc;
00993     INT nline, nplane;
00994
00995     dSTRmat *ILU_data=(dSTRmat *)data;
00996     INT m=ILU_data->ngrid;
00997     INT nc=ILU_data->nc;
00998     INT nc2=nc*nc;
00999     INT nx=ILU_data->nx;
01000     INT ny=ILU_data->ny;
01001     INT nz=ILU_data->nz;
01002     INT nxy=ILU_data->nxy;
01003     INT size=m*nc;
01004
01005     if (nx == 1) {
01006         nline = ny;
01007         nplane = m;
01008     }
01009     else if (ny == 1) {
01010         nline = nx;
01011         nplane = m;
01012     }
01013     else if (nz == 1) {
01014         nline = nx;
01015         nplane = m;
01016     }
01017     else {
01018         nline = nx;
01019         nplane = nxy;
01020     }
01021
01022     tc=(REAL*) fasp_mem_malloc(nc, sizeof(REAL));
01023
01024     zz=(REAL*) fasp_mem_malloc(size, sizeof(REAL));
01025

```

```

01026 // copy residual r to zr, to save r
01027 memcpy(zz,r,(size)*sizeof(REAL));
01028 if (nc == 1) {
01029     // backward sweep: solve upper matrix equation U*z=zz
01030
01031     z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01032     for (i=m-2;i>=0;i--) {
01033         zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01034         if (i<m-nline) zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline];
01035         if (i<m-nplane) zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nplane];
01036         z[i]=zz[i]*ILU_data->diag[i];
01037     }
01038 } // end if (nc == 1)
01039
01040
01041 else if (nc == 3) {
01042     // backward sweep: solve upper matrix equation U*z=zz
01043     fasp_blas_smat_mnv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
01044
01045     for (i=m-2;i>=0;i--) {
01046
01047         ic=i*nc;
01048         ic2=i*nc2;
01049
01050         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
01051         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01052
01053         if (i<m-nline) {
01054             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
01055             fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01056         }
01057
01058         if (i<m-nplane) {
01059             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
01060             fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01061         }
01062
01063         fasp_blas_smat_mnv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01064     } // end for for (i=m-2;i>=0;i--)
01065 } // end else if (nc == 3)
01066
01067
01068 else if (nc == 5) {
01069     // backward sweep: solve upper matrix equation U*z=zz
01070     fasp_blas_smat_mnv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
01071
01072     for (i=m-2;i>=0;i--) {
01073
01074         ic=i*nc;
01075         ic2=i*nc2;
01076
01077         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
01078         fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01079
01080         if (i<m-nline) {
01081             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
01082             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01083         }
01084
01085         if (i<m-nplane) {
01086             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
01087             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01088         }
01089
01090         fasp_blas_smat_mnv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01091     } // end for for (i=m-2;i>=0;i--)
01092 } // end else if (nc == 5)
01093
01094
01095 else if (nc == 7) {
01096     // backward sweep: solve upper matrix equation U*z=zz
01097     fasp_blas_smat_mnv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
01098
01099     for (i=m-2;i>=0;i--) {
01100
01101         ic=i*nc;
01102         ic2=i*nc2;
01103
01104         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
01105         fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01106

```



```

01107         if (i<m-nline) {
01108             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
01109             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01110         }
01111
01112         if (i<m-nplane) {
01113             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
01114             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01115         }
01116
01117         fasp_blas_smat_mnv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01118     } // end for for (i=m-2;i>=0;i--)
01119 } // end else if (nc == 7)
01120
01121
01122 else
01123 {
01124     // backward sweep: solve upper matrix equation U*z=zz
01125     fasp_blas_smat_mnv(&(ILU_data->diag[(m-1)*nc2]),&(z[(m-1)*nc]),&(z[(m-1)*nc]),nc);
01126
01127     for (i=m-2;i>=0;i--) {
01128         ic=i*nc;
01129         ic2=i*nc2;
01130
01131         fasp_blas_smat_mnv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
01132         fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
01133
01134         if (i<m-nline) {
01135             fasp_blas_smat_mnv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc,nc);
01136             fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
01137         }
01138
01139         if (i<m-nplane) {
01140             fasp_blas_smat_mnv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc,nc);
01141             fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
01142         }
01143
01144         fasp_blas_smat_mnv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);
01145
01146         } // end for (i=m-2;i>=0;i--)
01147     } // end else
01148
01149     fasp_mem_free(zz); zz = NULL;
01150     fasp_mem_free(tc); tc = NULL;
01151
01152     return;
01153 }
01154
01155 void fasp_precond_dstr_ilu1_forward (REAL *r,
01156                                     REAL *z,
01157                                     void *data)
01158 {
01159     REAL *zz,*zr,*tc;
01160
01161     dSTRmat *ILU_data=(dSTRmat *)data;
01162     INT i,ic;
01163     INT m=ILU_data->ngrid;
01164     INT nc=ILU_data->nc;
01165     INT nc2=nc*nc;
01166     INT nx=ILU_data->nx;
01167     INT ny=ILU_data->ny;
01168     INT nz=ILU_data->nz;
01169     INT nxy=ILU_data->nxy;
01170     INT size=m*nc;
01171     INT nline, nplane;
01172
01173     if (nx == 1) {
01174         nline = ny;
01175         nplane = m;
01176     }
01177     else if (ny == 1) {
01178         nline = nx;
01179         nplane = m;
01180     }
01181     else if (nz == 1) {
01182         nline = nx;
01183         nplane = m;
01184     }
01185     else {
01186         nline = nx;
01187     }
01188 }

```

```

01200     nplane = nxy;
01201 }
01202
01203 tc=(REAL*) fasp_mem_calloc(nc, sizeof(REAL));
01204
01205 zz=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
01206
01207 zr=(REAL*) fasp_mem_calloc(size, sizeof(REAL));
01208
01209 // copy residual r to zr, to save r
01210 //for (i=0;i<size;++i) zr[i]=r[i];
01211 memcpy(zr,r,(size)*sizeof(REAL));
01212 if (nc == 1) {
01213     // forward sweep: solve unit lower matrix equation L*zz=zr
01214     zz[0]=zr[0];
01215     for (i=1;i<m;++i) {
01216
01217         zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
01218         if (i>=nline-1)
01219             zz[i]=zz[i]-ILU_data->offdiag[2][i-nline+1]*zz[i-nline+1];
01220
01221         if (i>=nline)
01222             zz[i]=zz[i]-ILU_data->offdiag[4][i-nline]*zz[i-nline];
01223         if (i>=nplane-nline)
01224             zz[i]=zz[i]-ILU_data->offdiag[6][i-nplane+nline]*zz[i-nplane+nline];
01225         if (i>=nplane-1)
01226             zz[i]=zz[i]-ILU_data->offdiag[8][i-nplane+1]*zz[i-nplane+1];
01227         if (i>=nplane)
01228             zz[i]=zz[i]-ILU_data->offdiag[10][i-nplane]*zz[i-nplane];
01229     }
01230 } // end if (nc == 1)
01231
01232 else if (nc == 3) {
01233
01234     // forward sweep: solve unit lower matrix equation L*zz=zr
01235     fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));
01236
01237     for (i=1;i<m;++i) {
01238         ic=i*nc;
01239         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
01240         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
01241         fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01242
01243         if (i>=nline-1) {
01244             //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
01245             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
01246
01247             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01248         }
01249
01250         if (i>=nline) {
01251             //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
01252             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
01253
01254             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01255         }
01256         if (i>=nplane-nline) {
01257             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
01258             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
01259
01260             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01261         }
01262         if (i>=nplane-1) {
01263             // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
01264             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
01265             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01266         }
01267         if (i>=nplane) {
01268             //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
01269             fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
01270
01271             fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
01272         }
01273     }
01274     fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));

```

```

01275     }
01276
01277     } // end if (nc == 3)
01278
01279     else if (nc == 5) {
01280
01281         // forward sweep: solve unit lower matrix equation L*zz=zr
01282         fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));
01283
01284         for (i=1;i<m;++i) {
01285             ic=i*nc;
01286             //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
01287             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
01288             fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01289
01290             if (i>=nline-1) {
01291                 //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
01292                 fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
01293
01294                 fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01295             }
01296
01297             if (i>=nline) {
01298                 //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
01299                 fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
01300
01301                 fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01302             }
01303
01304             if (i>=nplane-nline) {
01305                 //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
01306                 fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
01307
01308                 fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01309             }
01310
01311             if (i>=nplane-1) {
01312                 // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
01313                 fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
01314
01315                 fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01316             }
01317
01318             if (i>=nplane) {
01319                 //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
01320                 fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
01321
01322                 fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
01323             }
01324
01325             fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
01326         }
01327     } // end if (nc == 5)
01328
01329     else if (nc == 7) {
01330
01331         // forward sweep: solve unit lower matrix equation L*zz=zr
01332         fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));
01333
01334         for (i=1;i<m;++i) {
01335             ic=i*nc;
01336             //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
01337             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
01338             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01339
01340             if (i>=nline-1) {
01341                 //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
01342                 fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
01343
01344                 fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01345             }
01346
01347             if (i>=nline) {
01348                 //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
01349                 fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
01350
01351                 fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01352             }
01353
01354             if (i>=nplane-nline) {
01355                 //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
01356                 fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
01357
01358                 fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01359             }
01360
01361             if (i>=nplane-1) {
01362                 // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
01363                 fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
01364
01365                 fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01366             }
01367
01368             if (i>=nplane) {
01369                 //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
01370                 fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
01371
01372                 fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01373             }
01374
01375             fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
01376         }
01377     }
01378 }

```

```

01348         if (i>=nplane-nline) {
01349             //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
01350             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

01351             fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01352         }
01353     }
01354     if (i>=nplane-1) {
01355         // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
01356         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
01357         fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01358     }
01359 }
01360 if (i>=nplane) {
01361     //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
01362     fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);

01363     fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
01364 }
01365 }
01366 fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
01367 }
01368 } // end if (nc == 7)
01369
01370 else {
01371     // forward sweep: solve unit lower matrix equation L*zz=zr
01372     fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
01373     for (i=1;i<m;++i) {
01374         ic=i*nc;
01375         //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
01376         fasp_blas_smat_mnv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
01377         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01378     }
01379     if (i>=nline-1) {
01380         //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
01381         fasp_blas_smat_mnv(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc,nc);
01382         fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01383     }
01384 }
01385 if (i>=nline) {
01386     //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
01387     fasp_blas_smat_mnv(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
01388     fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01389 }
01390 }
01391 if (i>=nplane-nline) {
01392     //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
01393     fasp_blas_smat_mnv(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc,nc);
01394     fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01395 }
01396 }
01397 if (i>=nplane-1) {
01398     // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
01399     fasp_blas_smat_mnv(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc,nc);
01400     fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01401 }
01402 }
01403 if (i>=nplane) {
01404     //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
01405     fasp_blas_smat_mnv(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
01406     fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
01407 }
01408 fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
01409 }
01410 } // end else
01411
01412 memcpy(z,zz,(size)*sizeof(REAL));
01413
01414 fasp_mem_free(zr); zr = NULL;
01415 fasp_mem_free(zz); zz = NULL;
01416 fasp_mem_free(tc); tc = NULL;
01417
01418

```

```

01419     return;
01420 }
01421
01434 void fasp_precond_dstr_ilu1_backward (REAL *r,
01435                                     REAL *z,
01436                                     void *data)
01437 {
01438     REAL *zz,*tc;
01439
01440     dSTRmat *ILU_data=(dSTRmat *)data;
01441     INT i,ic, ic2;
01442     INT m=ILU_data->ngrid;
01443     INT nc=ILU_data->nc;
01444     INT nc2=nc*nc;
01445     INT nx=ILU_data->nx;
01446     INT ny=ILU_data->ny;
01447     INT nz=ILU_data->nz;
01448     INT nxy=ILU_data->nxy;
01449     INT size=m*nc;
01450     INT nline, nplane;
01451
01452     if (nx == 1) {
01453         nline = ny;
01454         nplane = m;
01455     }
01456     else if (ny == 1) {
01457         nline = nx;
01458         nplane = m;
01459     }
01460     else if (nz == 1) {
01461         nline = nx;
01462         nplane = m;
01463     }
01464     else {
01465         nline = nx;
01466         nplane = nxy;
01467     }
01468
01469     tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));
01470
01471     zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));
01472
01473     // copy residual r to zr, to save r
01474     //for (i=0;i<size;++i) zr[i]=r[i];
01475     memcpy(zz,r,(size)*sizeof(REAL));
01476     if (nc == 1) {
01477         // backward sweep: solve upper matrix equation U*z=zz
01478
01479         z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01480         for (i=m-2;i>=0;i--) {
01481
01482             zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01483             if (i+nline-1<m)
01484                 zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline-1];
01485             if (i+nline<m)
01486                 zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nline];
01487             if (i+nplane-nline<m)
01488                 zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nplane-nline];
01489             if (i+nplane-1<m)
01490                 zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nplane-1];
01491             if (i+nplane<m)
01492                 zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nplane];
01493
01494             z[i]=ILU_data->diag[i]*zz[i];
01495         }
01496     }
01497
01498     // end if (nc == 1)
01499
01500     else if (nc == 3) {
01501         // backward sweep: solve upper matrix equation U*z=zz
01502
01503         // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01504         fasp_blas_smat_mxv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
01505
01506         for (i=m-2;i>=0;i--) {
01507             ic=i*nc;
01508             ic2=ic+nc;
01509
01510             //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01511             fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);

```

```

01512     fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01513
01514     if (i+nline-1<m) {
01515         //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
01516         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
01517         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01518     }
01519
01520     if (i+nline<m) {
01521         //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
01522         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
01523         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01524     }
01525
01526     if (i+nplane-nline<m) {
01527         //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nx-1];
01528         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
01529
01530         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01531     }
01532
01533     if (i+nplane-1<m) {
01534         //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nx-1];
01535         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
01536         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01537     }
01538
01539     if (i+nplane<m) {
01540         //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nx];
01541         fasp_blas_smat_mnv_nc3(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
01542         fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
01543     }
01544
01545     //z[i]=ILU_data->diag[i]*zz[i];
01546     fasp_blas_smat_mnv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01547 } // end for (i=m-2;i>=0;i--)
01548 } // end if (nc == 3)
01549
01550 else if (nc == 5) {
01551     // backward sweep: solve upper matrix equation U*z=zz
01552
01553     // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01554     fasp_blas_smat_mnv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(z[(m-1)*nc]),&(z[(m-1)*nc]));
01555
01556     for (i=m-2;i>=0;i--) {
01557         ic=i*nc;
01558         ic2=ic+nc;
01559
01560         //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01561         fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
01562         fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01563
01564         if (i+nline-1<m) {
01565             //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
01566             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
01567             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01568         }
01569
01570         if (i+nline<m) {
01571             //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
01572             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
01573             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01574         }
01575
01576         if (i+nplane-nline<m) {
01577             //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nx-1];
01578             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
01579
01580             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01581         }
01582
01583         if (i+nplane-1<m) {
01584             //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nx-1];
01585             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
01586             fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01587         }
01588
01589         if (i+nplane<m) {
01590             //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nx];
01591             fasp_blas_smat_mnv_nc5(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);

```

```

01591         fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
01592     }
01593
01594     //z[i]=ILU_data->diag[i]*zz[i];
01595     fasp_blas_smat_mnv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01596 } // end for (i=m-2;i>=0;i--)
01597
01598 } // end if (nc == 5)
01599
01600 else if (nc == 7) {
01601     // backward sweep: solve upper matrix equation U*z=zz
01602
01603     // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01604     fasp_blas_smat_mnv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));
01605
01606     for (i=m-2;i>=0;i--) {
01607         ic=i*nc;
01608         ic2=ic*nc;
01609
01610         //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01611         fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
01612         fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01613
01614         if (i+nline-1<m) {
01615             //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
01616             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
01617             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01618         }
01619
01620         if (i+nline<m) {
01621             //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
01622             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
01623             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01624         }
01625
01626         if (i+nplane-nline<m) {
01627             //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nx-1];
01628             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
01629
01630             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01631         }
01632
01633         if (i+nplane-1<m) {
01634             //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nx-1];
01635             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
01636             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01637         }
01638
01639         if (i+nplane<m) {
01640             //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nx];
01641             fasp_blas_smat_mnv_nc7(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
01642             fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
01643         }
01644
01645         //z[i]=ILU_data->diag[i]*zz[i];
01646         fasp_blas_smat_mnv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
01647     } // end for (i=m-2;i>=0;i--)
01648 } // end if (nc == 7)
01649
01650 else {
01651     // backward sweep: solve upper matrix equation U*z=zz
01652
01653     // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
01654     fasp_blas_smat_mnv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);
01655
01656     for (i=m-2;i>=0;i--) {
01657         ic=i*nc;
01658         ic2=ic*nc;
01659         //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
01660         fasp_blas_smat_mnv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
01661         fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
01662
01663         if (i+nline-1<m) {
01664             //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
01665             fasp_blas_smat_mnv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc,nc);
01666             fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
01667         }
01668
01669         if (i+nline<m) {
01670             //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];

```

```

01671         fasp_blas_smat_mnv(&(ILU_data->offdiag[5][ic2]), &(z[(i+nline)*nc]), tc, nc);
01672         fasp_blas_darray_axpy(nc, -1, tc, &(zz[ic]));
01673     }
01674
01675     if (i+nplane-nline<m) {
01676         //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
01677         fasp_blas_smat_mnv(&(ILU_data->offdiag[7][ic2]), &(z[(i+nplane-nline)*nc]), tc, nc);
01678         fasp_blas_darray_axpy(nc, -1, tc, &(zz[ic]));
01679     }
01680
01681     if (i+nplane-1<m) {
01682         //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
01683         fasp_blas_smat_mnv(&(ILU_data->offdiag[9][ic2]), &(z[(i+nplane-1)*nc]), tc, nc);
01684         fasp_blas_darray_axpy(nc, -1, tc, &(zz[ic]));
01685     }
01686
01687     if (i+nplane<m) {
01688         //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
01689         fasp_blas_smat_mnv(&(ILU_data->offdiag[11][ic2]), &(z[(i+nplane)*nc]), tc, nc);
01690         fasp_blas_darray_axpy(nc, -1, tc, &(zz[ic]));
01691     }
01692     //z[i]=ILU_data->diag[i]*zz[i];
01693     fasp_blas_smat_mnv(&(ILU_data->diag[ic2]), &(zz[ic]), &(z[ic]), nc);
01694 }
01695 } // end else
01696
01697 fasp_mem_free(zz); zz = NULL;
01698 fasp_mem_free(tc); tc = NULL;
01699
01700 return;
01701 }
01702
01715 void fasp_precond_dstr_blockgs (REAL *r,
01716                                REAL *z,
01717                                void *data)
01718 {
01719     precondition_data_str *predata=(precondition_data_str *)data;
01720     dSTRmat *A = predata->A_str;
01721     dvector *diainv = predata->diainv;
01722     ivector *pivot = predata->pivot;
01723     ivector *order = predata->order;
01724     ivector *neigh = predata->neigh;
01725
01726     INT i;
01727     const INT nc = A->nc;
01728     const INT ngrid = A->ngrid;
01729     const INT n = nc*ngrid; // whole size
01730
01731     dvector zz, rr;
01732     zz.row=rr.row=n; zz.val=z; rr.val=r;
01733     fasp_dvec_set(n, &zz, 0.0);
01734
01735     for (i=0; i<1; ++i)
01736         fasp_smoother_dstr_swz(A, &rr, &zz, diainv, pivot, neigh, order);
01737 }
01738
01739 /*-----*/
01740 /*--      Private Functions      --*/
01741 /*-----*/
01742
01754 static inline void fasp_darray_cp_nc3 (const REAL *x,
01755                                       REAL *y)
01756 {
01757     memcpy(y, x, 3*sizeof(REAL));
01758 }
01759
01771 static inline void fasp_darray_cp_nc5 (const REAL *x,
01772                                       REAL *y)
01773 {
01774     memcpy(y, x, 5*sizeof(REAL));
01775 }
01776
01788 static inline void fasp_darray_cp_nc7 (const REAL *x,
01789                                       REAL *y)
01790 {
01791     memcpy(y, x, 7*sizeof(REAL));
01792 }
01793
01794 /*-----*/
01795 /*--      End of File      --*/
01796 /*-----*/

```


9.175 SolAMG.c File Reference

AMG method as an iterative solver.

```
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- [INT fasp_solver_amg](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [AMG_param](#) *param)

Solve $Ax = b$ by algebraic multigrid methods.

9.175.1 Detailed Description

AMG method as an iterative solver.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSparseCheck.c](#), [BlaSparseCSR.c](#), [KrySPgmres.c](#), [PreAMGSetupRS.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreDataInit.c](#), and [PreMGSolve.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolAMG.c](#).

9.175.2 Function Documentation

9.175.2.1 fasp_solver_amg()

```
INT fasp_solver_amg (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    AMG_param * param )
```

Solve $Ax = b$ by algebraic multigrid methods.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector : the right hand side
<i>x</i>	Pointer to dvector : the unknowns
<i>param</i>	Pointer to AMG_param : AMG parameters

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

04/06/2010

Note

Refer to "Multigrid" by U. Trottenberg, C. W. Oosterlee and A. Schuller Appendix A.7 (by A. Brandt, P. Oswald and K. Stuben) Academic Press Inc., San Diego, CA, 2001.

Modified by Chensong Zhang on 07/26/2014: Add error handling for AMG setup Modified by Chensong Zhang on 02/01/2021: Add return value

Definition at line 49 of file [SolAMG.c](#).

9.176 SolAMG.c

[Go to the documentation of this file.](#)

```

00001
00016 #include <time.h>
00017
00018 #include "fasp.h"
00019 #include "fasp_functs.h"
00020
00021 /*-----*/
00022 /*--      Public Functions      --*/
00023 /*-----*/
00024
00049 INT fasp_solver_amg(dCSRmat* A, dvector* b, dvector* x, AMG_param* param)
00050 {
00051     const REAL tol      = param->tol;
00052     const SHORT max_levels = param->max_levels;
00053     const SHORT prtlvl   = param->print_level;
00054     const SHORT amg_type = param->AMG_type;
00055     const SHORT cycle_type = param->cycle_type;
00056     const INT maxit      = param->maxit;
00057     const INT nnz = A->nnz, m = A->row, n = A->col;
00058
00059     // local variables
00060     SHORT status;
00061     INT iter      = 0;
00062     AMG_data* mgl = fasp_amg_data_create(max_levels);
00063     REAL AMG_start = 0, AMG_end;
00064
00065     #if MULTI_COLOR_ORDER
00066         A->color = 0;
00067         A->IC    = NULL;
00068         A->ICMAP = NULL;
00069     #endif
00070
00071     #if DEBUG_MODE > 0
00072         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00073     #endif
00074
00075     if (prtlvl > PRINT_NONE) fasp_gettime(&AMG_start);
00076
00077     // check matrix data
00078     fasp_check_dCSRmat(A);
00079
00080     // Step 0: initialize mgl[0] with A, b and x
00081     mgl[0].A = fasp_dcsr_create(m, n, nnz);
00082     fasp_dcsr_cp(A, &mgl[0].A);
00083
00084     mgl[0].b = fasp_dvec_create(n);
00085     fasp_dvec_cp(b, &mgl[0].b);
00086
00087     mgl[0].x = fasp_dvec_create(n);
00088     fasp_dvec_cp(x, &mgl[0].x);
00089
00090     // Step 1: AMG setup phase
00091     switch (amg_type) {

```

```

00092
00093     case SA_AMG: // Smoothed Aggregation AMG setup
00094         status = fasp_amg_setup_sa(mgl, param);
00095         break;
00096
00097     case UA_AMG: // Unsmoothed Aggregation AMG setup
00098         status = fasp_amg_setup_ua(mgl, param);
00099         break;
00100
00101     default: // Classical AMG setup
00102         status = fasp_amg_setup_rs(mgl, param);
00103         break;
00104 }
00105
00106 // Step 2: AMG solve phase
00107 if (status == FASP_SUCCESS) { // call a multilevel cycle
00108
00109     switch (cycle_type) {
00110
00111         case AMLI_CYCLE: // AMLI-cycle
00112             iter = fasp_amg_solve_amli(mgl, param);
00113             break;
00114
00115         case NL_AMLI_CYCLE: // Nonlinear AMLI-cycle
00116             iter = fasp_amg_solve_namli(mgl, param);
00117             break;
00118
00119         default: // V,W-cycles or hybrid cycles (determined by param)
00120             iter = fasp_amg_solve(mgl, param);
00121             break;
00122     }
00123
00124     fasp_dvec_cp(&mgl[0].x, x);
00125
00126 }
00127
00128 else { // call a backup solver
00129
00130     if (prtlvl > PRINT_MIN) {
00131         printf("### WARNING: AMG setup failed!\n");
00132         printf("### WARNING: Use a backup solver instead!\n");
00133     }
00134     fasp_solver_dcsr_spgmres(A, b, x, NULL, tol, maxit, 20, 1, prtlvl);
00135 }
00136
00137 // clean-up memory
00138 fasp_amg_data_free(mgl, param);
00139
00140 // print out CPU time if needed
00141 if (prtlvl > PRINT_NONE) {
00142     fasp_gettime(&AMG_end);
00143     fasp_cputime("AMG totally", AMG_end - AMG_start);
00144 }
00145
00146 #if DEBUG_MODE > 0
00147     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00148 #endif
00149
00150     return iter;
00151 }
00152
00153 /*-----*/
00154 /*--           End of File           --*/
00155 /*-----*/

```

9.177 SolBLC.c File Reference

Iterative solvers for [dBLMat](#) matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_funcs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dblc_itsolver](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax = b$ by standard Krylov methods.
- [INT fasp_solver_dblc_krylov](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax = b$ by standard Krylov methods.
- [INT fasp_solver_dblc_krylov_block3](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam, [dCSRmat](#) *A_diag)
Solve $Ax = b$ by standard Krylov methods.
- [INT fasp_solver_dblc_krylov_block4](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam, [dCSRmat](#) *A_diag)
Solve $Ax = b$ by standard Krylov methods.
- [INT fasp_solver_dblc_krylov_sweeping](#) ([dBLCmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [INT](#) Num←Layers, [dBLCmat](#) *Ai, [dCSRmat](#) *local_A, [ivector](#) *local_index)
Solve $Ax = b$ by standard Krylov methods.

9.177.1 Detailed Description

Iterative solvers for [dBLCmat](#) matrices.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSparseCSR.c](#), [KryPbcgs.c](#), [KryPgmres.c](#), [KryPminres.c](#), [KryPvfgmres.c](#), [KryPvgmres.c](#), [PreAMGSetupRS.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreBLC.c](#), and [PreDataInit.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolBLC.c](#).

9.177.2 Function Documentation

9.177.2.1 fasp_solver_dblc_itsolver()

```
INT fasp_solver_dblc_itsolver (
    dBLCmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax = b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBLCmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

11/25/2010

Modified by Chunsheng Feng on 03/04/2016: add VBiCGstab solver
Definition at line 54 of file [SolBLC.c](#).

9.177.2.2 fasp_solver_dblc_krylov()

```
INT fasp_solver_dblc_krylov (
    dBLCMat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax = b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBLCMat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

07/18/2010

Definition at line 137 of file [SolBLC.c](#).

9.177.2.3 fasp_solver_dblc_krylov_block3()

```
INT fasp_solver_dblc_krylov_block3 (
    dBLCMat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam,
    dCSRmat * A_diag )
```

Solve $Ax = b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBLCMat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters for AMG solvers
<i>A_diag</i>	Digonal blocks of A

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

07/10/2014

Warning

Only works for 3X3 block problems!! – Xiaozhe Hu

Definition at line [189](#) of file [SolBLC.c](#).

9.177.2.4 fasp_solver_dblc_krylov_block4()

```

INT fasp_solver_dblc_krylov_block4 (
    dBLCMat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam,
    dCSRmat * A_diag )

```

Solve $Ax = b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBLCMat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters for AMG solvers
<i>A_diag</i>	Digonal blocks of A

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

07/06/2014

WarningOnly works for 4 by 4 block [dCSRmat](#) problems!! – Xiaozhe HuDefinition at line [379](#) of file [SolBLC.c](#).**9.177.2.5 fasp_solver_dblc_krylov_sweeping()**

```
INT fasp_solver_dblc_krylov_sweeping (
    dBLCMat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    INT NumLayers,
    dBLCMat * Ai,
    dCSRmat * local_A,
    ivector * local_index )
```

Solve $Ax = b$ by standard Krylov methods.**Parameters**

<i>A</i>	Pointer to the coeff matrix in dBLCMat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>NumLayers</i>	Number of layers used for sweeping preconditioner
<i>Ai</i>	Pointer to the coeff matrix for the preconditioner in dBLCMat format
<i>local_A</i>	Pointer to the local coeff matrices in the dCSRmat format
<i>local_index</i>	Pointer to the local index in ivector format

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/01/2014

Definition at line [501](#) of file [SolBLC.c](#).

9.178 SoIBLC.c

[Go to the documentation of this file.](#)

```

00001
00017 #include <math.h>
00018 #include <time.h>
00019
00020 #include "fasp.h"
00021 #include "fasp_block.h"
00022 #include "fasp_funcs.h"
00023
00024 /*-----*/
00025 /*--  Declare Private Functions  --*/
00026 /*-----*/
00027
00028 #include "KryUtil.inl"
00029
00030 /*-----*/
00031 /*--      Public Functions      --*/
00032 /*-----*/
00033
00054 INT fasp_solver_dblc_itsolver (dBLMat    *A,
00055                               dvector   *b,
00056                               dvector   *x,
00057                               precondition *pc,
00058                               ITS_param *itparam)
00059 {
00060     const SHORT prtlvl = itparam->print_level;
00061     const SHORT itsolver_type = itparam->itsolver_type;
00062     const SHORT stop_type = itparam->stop_type;
00063     const SHORT restart = itparam->restart;
00064     const INT   MaxIt = itparam->maxit;
00065     const REAL  tol = itparam->tol;
00066
00067     REAL solve_start, solve_end;
00068     INT  iter = ERROR_SOLVER_TYPE;
00069
00070     #if DEBUG_MODE > 0
00071         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00072         printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00073     #endif
00074
00075     fasp_gettime(&solve_start);
00076
00077     /* Safe-guard checks on parameters */
00078     ITS_CHECK ( MaxIt, tol );
00079
00080     switch (itsolver_type) {
00081
00082         case SOLVER_BiCGstab:
00083             iter=fasp_solver_dblc_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00084             break;
00085
00086         case SOLVER_MinRes:
00087             iter=fasp_solver_dblc_pminres(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00088             break;
00089
00090         case SOLVER_GMRES:
00091             iter=fasp_solver_dblc_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00092             break;
00093
00094         case SOLVER_VGMRES:
00095             iter=fasp_solver_dblc_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00096             break;
00097
00098         case SOLVER_VFGMRES:
00099             iter=fasp_solver_dblc_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00100             break;
00101
00102         default:
00103             printf("### ERROR: Unknown iterative solver type %d! [%s]\n",
00104                   itsolver_type, __FUNCTION__);
00105             return ERROR_SOLVER_TYPE;
00106     }
00107
00108     if ( (prtlvl >= PRINT_MIN) && (iter >= 0) ) {
00109         fasp_gettime(&solve_end);
00110         fasp_cputime("Iterative method", solve_end - solve_start);
00111     }
00112 }

```



```

00113
00114 #if DEBUG_MODE > 0
00115     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00116 #endif
00117
00118     return iter;
00119 }
00120
00137 INT fasp_solver_dblc_krylov (dBLMat    *A,
00138                             dvector   *b,
00139                             dvector   *x,
00140                             ITS_param *itparam)
00141 {
00142     const SHORT prtlvl = itparam->print_level;
00143
00144     INT status = FASP_SUCCESS;
00145     REAL solve_start, solve_end;
00146
00147 #if DEBUG_MODE > 0
00148     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00149 #endif
00150
00151     // solver part
00152     fasp_gettime(&solve_start);
00153
00154     status = fasp_solver_dblc_itsolver(A,b,x,NULL,itparam);
00155
00156     fasp_gettime(&solve_end);
00157
00158     if ( prtlvl >= PRINT_MIN )
00159         fasp_cputime("Krylov method totally", solve_end - solve_start);
00160
00161 #if DEBUG_MODE > 0
00162     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00163 #endif
00164
00165     return status;
00166 }
00167
00189 INT fasp_solver_dblc_krylov_block3 (dBLMat    *A,
00190                                    dvector   *b,
00191                                    dvector   *x,
00192                                    ITS_param *itparam,
00193                                    AMG_param *amgparam,
00194                                    dCSRmat    *A_diag)
00195 {
00196     const SHORT prtlvl = itparam->print_level;
00197     const SHORT precondition_type = itparam->precond_type;
00198
00199     INT status = FASP_SUCCESS;
00200     REAL setup_start, setup_end;
00201     REAL solve_start, solve_end;
00202
00203     const SHORT max_levels = amgparam->max_levels;
00204     INT m, n, nnz, i;
00205
00206     AMG_data **mgl = NULL;
00207
00208 #if WITH_UMFPACK
00209     void **LU_diag = (void **)fasp_mem_malloc(3, sizeof(void *));
00210 #endif
00211
00212 #if DEBUG_MODE > 0
00213     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00214 #endif
00215
00216     /* setup preconditioner */
00217     fasp_gettime(&solve_start);
00218     fasp_gettime(&setup_start);
00219
00220     /* diagonal blocks are solved exactly */
00221     if ( precondition_type > 20 && precondition_type < 30 ) {
00222 #if WITH_UMFPACK
00223         // Need to sort the diagonal blocks for UMFPACK format
00224         dCSRmat A_tran;
00225
00226         for (i=0; i<3; i++){
00227
00228             A_tran = fasp_dcsr_create(A_diag[i].row, A_diag[i].col, A_diag[i].nnz);
00229             fasp_dcsr_transz(&A_diag[i], NULL, &A_tran);
00230             fasp_dcsr_cp(&A_tran, &A_diag[i]);

```

```

00231
00232     printf("Factorization for %d-th diagonal: \n", i);
00233     LU_diag[i] = fasp_umfpack_factorize(&A_diag[i], prt1vl);
00234
00235 }
00236
00237     fasp_dcsr_free(&A_tran);
00238 #endif
00239 }
00240
00241 /* diagonal blocks are solved by AMG */
00242 else if ( precondition_type > 30 && precondition_type < 40 ) {
00243
00244     mgl = (AMG_data **)fasp_mem_calloc(3, sizeof(AMG_data *));
00245
00246     for (i=0; i<3; i++){
00247
00248         mgl[i] = fasp_amg_data_create(max_levels);
00249         m = A_diag[i].row; n = A_diag[i].col; nnz = A_diag[i].nnz;
00250         mgl[i][0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(&A_diag[i], &mgl[i][0].A);
00251         mgl[i][0].b=fasp_dvec_create(n); mgl[i][0].x=fasp_dvec_create(n);
00252
00253         switch (amgparam->AMG_type) {
00254             case SA_AMG: // Smoothed Aggregation AMG
00255                 status = fasp_amg_setup_sa(mgl[i], amgparam); break;
00256             case UA_AMG: // Unsmoothed Aggregation AMG
00257                 status = fasp_amg_setup_ua(mgl[i], amgparam); break;
00258             default: // Classical AMG
00259                 status = fasp_amg_setup_rs(mgl[i], amgparam); break;
00260         }
00261
00262         fasp_chkerr(status, __FUNCTION__);
00263     }
00264 }
00265
00266 else {
00267     fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
00268 }
00269
00270
00271 precdata_data_blc precdata;
00272 precdata.Ablc = A;
00273 precdata.A_diag = A_diag;
00274 precdata.r = fasp_dvec_create(b->row);
00275
00276 /* diagonal blocks are solved exactly */
00277 if ( precondition_type > 20 && precondition_type < 30 ) {
00278 #if WITH_UMFPACK
00279     precdata.LU_diag = LU_diag;
00280 #endif
00281 }
00282 /* diagonal blocks are solved by AMG */
00283 else if ( precondition_type > 30 && precondition_type < 40 ) {
00284     precdata.amgparam = amgparam;
00285     precdata.mgl = mgl;
00286 }
00287 else {
00288     fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
00289 }
00290
00291 precdata_prec; precdata_data = &precdata;
00292
00293 switch (precondition_type) {
00294     case 21:
00295         precdata_prec.fct = fasp_precond_dblc_diag_3; break;
00296
00297     case 22:
00298         precdata_prec.fct = fasp_precond_dblc_lower_3; break;
00299
00300     case 23:
00301         precdata_prec.fct = fasp_precond_dblc_upper_3; break;
00302
00303     case 24:
00304         precdata_prec.fct = fasp_precond_dblc_SGS_3; break;
00305
00306     case 31:
00307         precdata_prec.fct = fasp_precond_dblc_diag_3_amg; break;
00308
00309     case 32:
00310         precdata_prec.fct = fasp_precond_dblc_lower_3_amg; break;
00311

```

```

00312         case 33:
00313             prec.fct = fasp_precond_dblc_upper_3_amg; break;
00314
00315         case 34:
00316             prec.fct = fasp_precond_dblc_SGS_3_amg; break;
00317
00318         default:
00319             fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__); break;
00320     }
00321
00322     if ( prtlvl >= PRINT_MIN ) {
00323         fasp_gettime(&setup_end);
00324         fasp_cputime("Setup totally", setup_end - setup_start);
00325     }
00326
00327     // solve part
00328     status = fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);
00329
00330     fasp_gettime(&solve_end);
00331
00332     if ( prtlvl >= PRINT_MIN )
00333         fasp_cputime("Krylov method totally", solve_end - solve_start);
00334
00335     // clean up
00336     /* diagonal blocks are solved exactly */
00337     if ( precondition_type > 20 && precondition_type < 30 ) {
00338 #if WITH_UMFPACK
00339         for (i=0; i<3; i++) fasp_umfpack_free_numeric(LU_diag[i]);
00340 #endif
00341     }
00342     /* diagonal blocks are solved by AMG */
00343     else if ( precondition_type > 30 && precondition_type < 40 ) {
00344         for (i=0; i<3; i++) fasp_amg_data_free(mgl[i], amgparam);
00345         fasp_mem_free(mgl); mgl = NULL;
00346     }
00347     else {
00348         fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
00349     }
00350
00351 #if DEBUG_MODE > 0
00352     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00353 #endif
00354
00355     return status;
00356 }
00357
00379 INT fasp_solver_dblc_krylov_block4 (dBLCmat    *A,
00380                                   dvector    *b,
00381                                   dvector    *x,
00382                                   ITS_param  *itparam,
00383                                   AMG_param  *amgparam,
00384                                   dCSRmat    *A_diag)
00385 {
00386     const SHORT prtlvl = itparam->print_level;
00387     const SHORT precondition_type = itparam->precondition_type;
00388
00389     INT status = FASP_SUCCESS;
00390     REAL setup_start, setup_end;
00391     REAL solve_start, solve_end;
00392
00393 #if DEBUG_MODE > 0
00394     printf("### DEBUG: [-Begin-] %s ...\\n", __FUNCTION__);
00395 #endif
00396
00397     /* setup preconditioner */
00398     fasp_gettime(&solve_start);
00399     fasp_gettime(&setup_start);
00400
00401 #if WITH_UMFPACK
00402     void **LU_diag = (void **)fasp_mem_calloc(4, sizeof(void *));
00403     INT i;
00404 #endif
00405
00406     /* diagonal blocks are solved exactly */
00407     if ( precondition_type > 20 && precondition_type < 30 ) {
00408 #if WITH_UMFPACK
00409         // Need to sort the matrices local_A for UMFPACK format
00410         dCSRmat A_tran;
00411
00412         for (i=0; i<4; i++){

```

```

00414
00415     A_tran = fasp_dcsr_create(A_diag[i].row, A_diag[i].col, A_diag[i].nnz);
00416     fasp_dcsr_transz(&A_diag[i], NULL, &A_tran);
00417     fasp_dcsr_cp(&A_tran, &A_diag[i]);
00418
00419     printf("Factorization for %d-th diagonal: \n", i);
00420     LU_diag[i] = fasp_umfpack_factorize(&A_diag[i], prtlvl);
00421
00422 }
00423
00424     fasp_dcsr_free(&A_tran);
00425 #endif
00426 }
00427
00428     else {
00429         fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
00430     }
00431
00432     precondition_data_blc precddata;
00433
00434     precddata.Ablc = A;
00435     precddata.A_diag = A_diag;
00436 #if WITH_UMFPACK
00437     precddata.LU_diag = LU_diag;
00438 #endif
00439     precddata.r = fasp_dvec_create(b->row);
00440
00441     precondition prec; prec.data = &precddata;
00442
00443     switch (precond_type)
00444     {
00445         case 21:
00446             prec.fct = fasp_precond_dblc_diag_4;
00447             break;
00448
00449         case 22:
00450             prec.fct = fasp_precond_dblc_lower_4;
00451             break;
00452     }
00453
00454     if ( prtlvl >= PRINT_MIN ) {
00455         fasp_gettime(&setup_end);
00456         fasp_cputime("Setup totally", setup_end - setup_start);
00457     }
00458
00459     // solver part
00460     status=fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);
00461
00462     fasp_gettime(&solve_end);
00463
00464     if ( prtlvl >= PRINT_MIN )
00465         fasp_cputime("Krylov method totally", solve_end - solve_start);
00466
00467     // clean
00468 #if WITH_UMFPACK
00469     for (i=0; i<4; i++) fasp_umfpack_free_numeric(LU_diag[i]);
00470 #endif
00471
00472 #if DEBUG_MODE > 0
00473     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00474 #endif
00475
00476     return status;
00477 }
00478
00501 INT fasp_solver_dblc_krylov_sweeping (dBLMat    *A,
00502                                     dvector    *b,
00503                                     dvector    *x,
00504                                     ITS_param  *itparam,
00505                                     INT         NumLayers,
00506                                     dBLMat      *Ai,
00507                                     dCSRmat     *local_A,
00508                                     ivector     *local_index)
00509 {
00510     const SHORT prtlvl = itparam->print_level;
00511
00512     INT status = FASP_SUCCESS;
00513     REAL setup_start, setup_end;
00514     REAL solve_start, solve_end;
00515
00516     void **local_LU = NULL;

```

```

00517
00518 #if DEBUG_MODE > 0
00519     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00520 #endif
00521
00522     /* setup preconditioner */
00523     fasp_gettime(&solve_start);
00524     fasp_gettime(&setup_start);
00525
00526 #if WITH_UMFPACK
00527     // Need to sort the matrices local_A for UMFPACK format
00528     INT l;
00529     dCSRmat A_tran;
00530     local_LU = (void **)fasp_mem_calloc(NumLayers, sizeof(void *));
00531
00532     for ( l=0; l<NumLayers; l++ ) {
00533
00534         A_tran = fasp_dcsr_create(local_A[l].row, local_A[l].col, local_A[l].nnz);
00535         fasp_dcsr_transz(&local_A[l], NULL, &A_tran);
00536         fasp_dcsr_cp(&A_tran, &local_A[l]);
00537
00538         printf("Factorization for layer %d:  \n", l);
00539         local_LU[l] = fasp_umfpack_factorize(&local_A[l], prt1vl);
00540
00541     }
00542
00543     fasp_dcsr_free(&A_tran);
00544 #endif
00545
00546     precondition_data_sweeping precd_data;
00547     precd_data.NumLayers = NumLayers;
00548     precd_data.A = A;
00549     precd_data.Ai = Ai;
00550     precd_data.local_A = local_A;
00551     precd_data.local_LU = local_LU;
00552     precd_data.local_index = local_index;
00553     precd_data.r = fasp_dvec_create(b->row);
00554     precd_data.w = (REAL *)fasp_mem_calloc(10*b->row, sizeof(REAL));
00555
00556     precondition prec; prec.data = &precd_data;
00557     prec.fct = fasp_precond_dblc_sweeping;
00558
00559     if ( prt1vl >= PRINT_MIN ) {
00560         fasp_gettime(&setup_end);
00561         fasp_cputime("Setup totally", setup_end - setup_start);
00562     }
00563
00564     /* solver part */
00565     status = fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);
00566
00567     fasp_gettime(&solve_end);
00568
00569     if ( prt1vl >= PRINT_MIN )
00570         fasp_cputime("Krylov method totally", solve_end - solve_start);
00571
00572     // clean
00573 #if WITH_UMFPACK
00574     for (l=0; l<NumLayers; l++) fasp_umfpack_free_numeric(local_LU[l]);
00575 #endif
00576
00577 #if DEBUG_MODE > 0
00578     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00579 #endif
00580
00581     return status;
00582 }
00583
00584 /*-----*/
00585 /*--          End of File          --*/
00586 /*-----*/

```

9.179 SolBSR.c File Reference

Iterative solvers for [dBSRmat](#) matrices.

```

#include <time.h>
#include "fasp.h"

```

```
#include "fasp_funcs.h"
#include "KryUtil.inl"
```

Functions

- [INT fasp_solver_dbsr_itsolver](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax=b$ by preconditioned Krylov methods for BSR matrices.
- [INT fasp_solver_dbsr_krylov](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods for BSR matrices.
- [INT fasp_solver_dbsr_krylov_diag](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by diagonal preconditioned Krylov methods.
- [INT fasp_solver_dbsr_krylov_ilu](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [ILU_param](#) *iluparam)
Solve $Ax=b$ by ILUs preconditioned Krylov methods.
- [INT fasp_solver_dbsr_krylov_amg](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam)
Solve $Ax=b$ by AMG preconditioned Krylov methods.
- [INT fasp_solver_dbsr_krylov_amg_nk](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam, [dCSRmat](#) *A_nk, [dCSRmat](#) *P_nk, [dCSRmat](#) *R_nk)
Solve $Ax=b$ by AMG with extra near kernel solve preconditioned Krylov methods.
- [INT fasp_solver_dbsr_krylov_nk_amg](#) ([dBSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam, [const INT](#) nk_dim, [dvector](#) *nk)
Solve $Ax=b$ by AMG preconditioned Krylov methods with extra kernal space.

9.179.1 Detailed Description

Iterative solvers for [dBSRmat](#) matrices.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMemory.c](#), [AuxMessage.c](#), [AuxThreads.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSmallMatInv.c](#), [BlaLUSetupBSR.c](#), [BlaSparseBSR.c](#), [BlaSparseCheck.c](#), [KryPbcgs.c](#), [KryPcg.c](#), [KryPgmres.c](#), [KryPvfgmres.c](#), [KryPvgmres.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreBSR.c](#), and [PreDataInit.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolBSR.c](#).

9.179.2 Function Documentation

9.179.2.1 fasp_solver_dbsr_itsolver()

```
INT fasp_solver_dbsr_itsolver (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax=b$ by preconditioned Krylov methods for BSR matrices.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou, Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng on 03/04/2016: add VBiCGstab solver
Definition at line [55](#) of file [SolBSR.c](#).

9.179.2.2 fasp_solver_dbsr_krylov()

```
INT fasp_solver_dbsr_krylov (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods for BSR matrices.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou, Xiaozhe Hu

Date

10/26/2010

Definition at line [139](#) of file [SolBSR.c](#).

9.179.2.3 fasp_solver_dbsr_krylov_amg()

```
INT fasp_solver_dbsr_krylov_amg (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam )
```

Solve $Ax=b$ by AMG preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters of AMG

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

03/16/2012

parameters of iterative method

Definition at line [354](#) of file [SolBSR.c](#).

9.179.2.4 fasp_solver_dbsr_krylov_amg_nk()

```
INT fasp_solver_dbsr_krylov_amg_nk (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam,
    dCSRmat * A_nk,
    dCSRmat * P_nk,
    dCSRmat * R_nk )
```

Solve $Ax=b$ by AMG with extra near kernel solve preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters of AMG

Parameters

<i>A_nk</i>	Pointer to the coeff matrix for near kernel space in dBSRmat format
<i>P_nk</i>	Pointer to the prolongation for near kernel space in dBSRmat format
<i>R_nk</i>	Pointer to the restriction for near kernel space in dBSRmat format

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/26/2012

Definition at line [483](#) of file [SolBSR.c](#).

9.179.2.5 fasp_solver_dbsr_krylov_diag()

```
INT fasp_solver_dbsr_krylov_diag (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou, Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng, Zheng Li on 10/15/2012

Definition at line [187](#) of file [SolBSR.c](#).

9.179.2.6 fasp_solver_dbsr_krylov_ilu()

```

INT fasp_solver_dbsr_krylov_ilu (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    ILU_param * iluparam )

```

Solve $Ax=b$ by ILUs preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>iluparam</i>	Pointer to parameters of ILU

Returns

Iteration number if converges; ERROR otherwise.

Author

Shiquang Zhang, Xiaozhe Hu

Date

10/26/2010

Definition at line [289](#) of file [SolBSR.c](#).

9.179.2.7 fasp_solver_dbsr_krylov_nk_amg()

```

INT fasp_solver_dbsr_krylov_nk_amg (
    dBSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam,
    const INT nk_dim,
    dvector * nk )

```

Solve $Ax=b$ by AMG preconditioned Krylov methods with extra kernal space.

Parameters

<i>A</i>	Pointer to the coeff matrix in dBSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters of AMG
<i>nk_dim</i>	Dimension of the near kernel spaces
<i>nk</i>	Pointer to the near kernal spaces

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/27/2012

parameters of iterative method

Definition at line 640 of file SolBSR.c.

9.180 SolBSR.c

[Go to the documentation of this file.](#)

```

00001
00017 #include <time.h>
00018
00019 #ifdef _OPENMP
00020 #include <omp.h>
00021 #endif
00022
00023 #include "fasp.h"
00024 #include "fasp_funcs.h"
00025
00026 /*-----*/
00027 /*--  Declare Private Functions  --*/
00028 /*-----*/
00029
00030 #include "KryUtil.inl"
00031
00032 /*-----*/
00033 /*--      Public Functions      --*/
00034 /*-----*/
00035
00055 INT fasp_solver_dbsr_itsolver (dBSRmat    *A,
00056                               dvector    *b,
00057                               dvector    *x,
00058                               precondition *pc,
00059                               ITS_param  *itparam)
00060 {
00061     const SHORT prtlvl = itparam->print_level;
00062     const SHORT itsolver_type = itparam->itsolver_type;
00063     const SHORT stop_type = itparam->stop_type;
00064     const SHORT restart = itparam->restart;
00065     const INT   MaxIt = itparam->maxit;
00066     const REAL  tol = itparam->tol;
00067
00068     // Local variables
00069     INT iter = ERROR_SOLVER_TYPE;
00070     REAL solve_start, solve_end;
00071
00072     #if DEBUG_MODE > 0
00073         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00074         printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00075     #endif
00076
00077     fasp_gettime(&solve_start);
00078
00079     /* Safe-guard checks on parameters */
00080     ITS_CHECK ( MaxIt, tol );
00081
00082     switch (itsolver_type) {
00083
00084     case SOLVER_CG:
00085         iter = fasp_solver_dbsr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00086         break;
00087
00088     case SOLVER_BiCGstab:
00089         iter = fasp_solver_dbsr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00090         break;

```

```

00091
00092     case SOLVER_GMRES:
00093         iter = fasp_solver_dbsr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00094         break;
00095
00096     case SOLVER_VGMRES:
00097         iter = fasp_solver_dbsr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00098         break;
00099
00100     case SOLVER_VFGMRES:
00101         iter = fasp_solver_dbsr_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00102         break;
00103
00104     default:
00105         printf("### ERROR: Unknown iterative solver type %d! [%s]\n",
00106             itsolver_type, __FUNCTION__);
00107         return ERROR_SOLVER_TYPE;
00108
00109 }
00110
00111 if ( (prtlvl > PRINT_MIN) && (iter >= 0) ) {
00112     fasp_gettime(&solve_end);
00113     fasp_cputime("Iterative method", solve_end - solve_start);
00114 }
00115
00116 #if DEBUG_MODE > 0
00117     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00118 #endif
00119
00120     return iter;
00121 }
00122
00139 INT fasp_solver_dbsr_krylov (dBSRmat    *A,
00140                             dvector    *b,
00141                             dvector    *x,
00142                             ITS_param  *itparam)
00143 {
00144     const SHORT prtlvl = itparam->print_level;
00145     INT status = FASP_SUCCESS;
00146     REAL solve_start, solve_end;
00147
00148     #if DEBUG_MODE > 0
00149         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00150     #endif
00151
00152     // solver part
00153     fasp_gettime(&solve_start);
00154
00155     status=fasp_solver_dbsr_itsolver(A,b,x,NULL,itparam);
00156
00157     fasp_gettime(&solve_end);
00158
00159     if ( prtlvl > PRINT_NONE )
00160         fasp_cputime("Krylov method totally", solve_end - solve_start);
00161
00162     #if DEBUG_MODE > 0
00163         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00164     #endif
00165
00166     return status;
00167 }
00168
00187 INT fasp_solver_dbsr_krylov_diag (dBSRmat    *A,
00188                                  dvector    *b,
00189                                  dvector    *x,
00190                                  ITS_param  *itparam)
00191 {
00192     const SHORT prtlvl = itparam->print_level;
00193     INT status = FASP_SUCCESS;
00194     REAL solve_start, solve_end;
00195
00196     INT nb=A->nb,i,k;
00197     INT nb2=nb*nb;
00198     INT ROW=A->ROW;
00199
00200 #ifdef _OPENMP
00201     // variables for OpenMP
00202     INT myid, mybegin, myend;
00203     INT nthreads = fasp_get_num_threads();
00204 #endif
00205     // setup preconditioner

```

```

00206     preconditioning diag;
00207     fasp_dvec_alloc(ROW*nb2, &(diag.diag));
00208
00209 #if DEBUG_MODE > 0
00210     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00211 #endif
00212
00213     // get all the diagonal sub-blocks
00214 #ifdef _OPENMP
00215     if (ROW > OPENMP_HOLDS) {
00216 #pragma omp parallel for private(myid, mybegin, myend, i, k)
00217         for (myid=0; myid<nthreads; ++myid) {
00218             fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
00219             for (i = mybegin; i < myend; ++i) {
00220                 for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
00221                     if (A->JA[k] == i)
00222                         memcpy(diag.diag.val+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
00223                 }
00224             }
00225         }
00226     }
00227     else {
00228 #endif
00229         for (i = 0; i < ROW; ++i) {
00230             for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
00231                 if (A->JA[k] == i)
00232                     memcpy(diag.diag.val+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
00233             }
00234         }
00235 #ifdef _OPENMP
00236     }
00237 #endif
00238
00239     diag.nb=nb;
00240
00241 #ifdef _OPENMP
00242 #pragma omp parallel for if(ROW>OPENMP_HOLDS)
00243 #endif
00244     for (i=0; i<ROW; ++i){
00245         fasp_smat_inv(&(diag.diag.val[i*nb2]), nb);
00246     }
00247
00248     precondition *pc = (precondition *)fasp_mem_calloc(1, sizeof(precondition));
00249     pc->data = &diag;
00250     pc->fct = fasp_precond_dbsr_diag;
00251
00252     // solver part
00253     fasp_gettime(&solve_start);
00254
00255     status=fasp_solver_dbsr_itsolver(A,b,x,pc,itparam);
00256
00257     fasp_gettime(&solve_end);
00258
00259     if ( prtlvl > PRINT_NONE )
00260         fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);
00261
00262     // clean up
00263     fasp_dvec_free(&(diag.diag));
00264
00265 #if DEBUG_MODE > 0
00266     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00267 #endif
00268
00269     return status;
00270 }
00271
00289 INT fasp_solver_dbsr_krylov_ilu (dBSRmat *A,
00290                                dvector *b,
00291                                dvector *x,
00292                                ITS_param *itparam,
00293                                ILU_param *iluparam)
00294 {
00295     const SHORT prtlvl = itparam->print_level;
00296     REAL solve_start, solve_end;
00297     INT status = FASP_SUCCESS;
00298
00299     ILU_data LU;
00300     precondition pc;
00301
00302 #if DEBUG_MODE > 0
00303     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);

```

```

00304     printf("### DEBUG: matrix size:  %d %d %d\n", A->ROW, A->COL, A->NNZ);
00305     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00306 #endif
00307
00308     fasp_gettime(&solve_start);
00309
00310     // ILU setup for whole matrix
00311     if ( (status = fasp_ilu_dbsr_setup(A, &LU, iluparam)) < 0 ) goto FINISHED;
00312
00313     // check iludata
00314     if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;
00315
00316     // set preconditioner
00317     pc.data = &LU; pc.fct = fasp_precond_dbsr_ilu;
00318
00319     // solve
00320     status = fasp_solver_dbsr_itsolver(A, b, x, &pc, itparam);
00321
00322     fasp_gettime(&solve_end);
00323
00324     if ( prtlvl > PRINT_NONE )
00325         fasp_cputime("ILUK_Krylov method totally", solve_end - solve_start);
00326
00327 FINISHED:
00328     fasp_ilu_data_free(&LU);
00329
00330 #if DEBUG_MODE > 0
00331     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00332 #endif
00333
00334     return status;
00335 }
00336
00354 INT fasp_solver_dbsr_krylov_amg (dBSRmat      *A,
00355                                dvector      *b,
00356                                dvector      *x,
00357                                ITS_param    *itparam,
00358                                AMG_param    *amgparam)
00359 {
00360     //-----
00361     // Part 1:  prepare
00362     // -----
00363
00365     const SHORT prtlvl = itparam->print_level;
00366     const SHORT max_levels = amgparam->max_levels;
00367
00368     // return variable
00369     INT status = FASP_SUCCESS;
00370
00371     // data of AMG
00372     AMG_data_bsr *mgl = fasp_amg_data_bsr_create(max_levels);
00373
00374     // timing
00375     REAL setup_start, setup_end, solve_end;
00376
00377 #if DEBUG_MODE > 0
00378     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00379 #endif
00380
00381     //-----
00382     //Part 2:  set up the preconditioner
00383     //-----
00384     fasp_gettime(&setup_start);
00385
00386     // initialize A, b, x for mgl[0]
00387     mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
00388     mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
00389     mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);
00390
00391     fasp_dbsr_cp(A, &(mgl[0].A));
00392
00393     switch (amgparam->AMG_type) {
00394
00395         case SA_AMG: // Smoothed Aggregation AMG
00396             status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;
00397
00398         default:
00399             status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;
00400
00401     }
00402

```

```

00403     if (status < 0) goto FINISHED;
00404
00405     precondition_data_bsr precondition_data;
00406     precondition_data.print_level = amgparam->print_level;
00407     precondition_data.maxit = amgparam->maxit;
00408     precondition_data.tol = amgparam->tol;
00409     precondition_data.cycle_type = amgparam->cycle_type;
00410     precondition_data.smoother = amgparam->smoother;
00411     precondition_data.presmooth_iter = amgparam->presmooth_iter;
00412     precondition_data.postsmooth_iter = amgparam->postsmooth_iter;
00413     precondition_data.coarsening_type = amgparam->coarsening_type;
00414     precondition_data.relaxation = amgparam->relaxation;
00415     precondition_data.coarse_scaling = amgparam->coarse_scaling;
00416     precondition_data.aml_i_degree = amgparam->aml_i_degree;
00417     precondition_data.aml_i_coef = amgparam->aml_i_coef;
00418     precondition_data.tentative_smooth = amgparam->tentative_smooth;
00419     precondition_data.max_levels = mgl[0].num_levels;
00420     precondition_data.mgl_data = mgl;
00421     precondition_data.A = A;
00422
00423     precondition prec;
00424     prec.data = &precondition_data;
00425     switch (amgparam->cycle_type) {
00426     case NL_AMLI_CYCLE: // Nonlinear AMLI AMG
00427         prec.fct = fasp_precond_dbsr_namli; break;
00428     default: // V,W-Cycle AMG
00429         prec.fct = fasp_precond_dbsr_amg; break;
00430     }
00431
00432     fasp_gettime(&setup_end);
00433
00434     if ( prtlvl >= PRINT_MIN )
00435         fasp_cputime("BSR AMG setup", setup_end - setup_start);
00436
00437     //-----
00438     // Part 3: solver
00439     //-----
00440     status = fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);
00441
00442     fasp_gettime(&solve_end);
00443
00444     if ( prtlvl >= PRINT_MIN )
00445         fasp_cputime("BSR Krylov method", solve_end - setup_start);
00446
00447 FINISHED:
00448     fasp_amg_data_bsr_free(mgl);
00449
00450 #if DEBUG_MODE > 0
00451     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00452 #endif
00453
00454     if ( status == ERROR_ALLOC_MEM ) goto MEMORY_ERROR;
00455     return status;
00456
00457 MEMORY_ERROR:
00458     printf("### ERROR: Cannot allocate memory! [%s]\n", __FUNCTION__);
00459     exit(status);
00460 }
00461
00483 INT fasp_solver_dbsr_krylov_amg_nk (dBSRmat *A,
00484                                   dvector *b,
00485                                   dvector *x,
00486                                   ITS_param *itparam,
00487                                   AMG_param *amgparam,
00488                                   dCSRmat *A_nk,
00489                                   dCSRmat *P_nk,
00490                                   dCSRmat *R_nk)
00491 {
00492     //-----
00493     // Part 1: prepare
00494     //-----
00495     // parameters of iterative method
00496     const SHORT prtlvl = itparam->print_level;
00497     const SHORT max_levels = amgparam->max_levels;
00498
00499     // return variable
00500     INT status = FASP_SUCCESS;
00501
00502     // data of AMG
00503     AMG_data_bsr *mgl=fasp_amg_data_bsr_create(max_levels);
00504

```

```

00505 // timing
00506 REAL setup_start, setup_end, setup_time;
00507 REAL solve_start, solve_end, solve_time;
00508
00509 #if DEBUG_MODE > 0
00510 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00511 #endif
00512
00513 //-----
00514 //Part 2: set up the preconditioner
00515 //-----
00516 fasp_gettime(&setup_start);
00517
00518 // initialize A, b, x for mgl[0]
00519 mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
00520 fasp_dbsr_cp(A, &(mgl[0].A));
00521 mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
00522 mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);
00523
00524 // near kernel space
00525 mgl[0].A_nk = NULL;
00526 mgl[0].P_nk = P_nk;
00527 mgl[0].R_nk = R_nk;
00528
00529 switch (amgparam->AMG_type) {
00530
00531     case SA_AMG: // Smoothed Aggregation AMG
00532         status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;
00533
00534     default:
00535         status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;
00536
00537 }
00538
00539 if (status < 0) goto FINISHED;
00540
00541 precond_data_bsr precd_data;
00542 precd_data.print_level = amgparam->print_level;
00543 precd_data.maxit = amgparam->maxit;
00544 precd_data.tol = amgparam->tol;
00545 precd_data.cycle_type = amgparam->cycle_type;
00546 precd_data.smoother = amgparam->smoother;
00547 precd_data.presmooth_iter = amgparam->presmooth_iter;
00548 precd_data.postsmooth_iter = amgparam->postsmooth_iter;
00549 precd_data.coarsening_type = amgparam->coarsening_type;
00550 precd_data.relaxation = amgparam->relaxation;
00551 precd_data.coarse_scaling = amgparam->coarse_scaling;
00552 precd_data.amli_degree = amgparam->amli_degree;
00553 precd_data.amli_coef = amgparam->amli_coef;
00554 precd_data.tentative_smooth = amgparam->tentative_smooth;
00555 precd_data.max_levels = mgl[0].num_levels;
00556 precd_data.mgl_data = mgl;
00557 precd_data.A = A;
00558
00559 #if WITH_UMFPACK // use UMFPACK directly
00560 dCSRmat A_tran;
00561 A_tran = fasp_dcsr_create(A_nk->row, A_nk->col, A_nk->nnz);
00562 fasp_dcsr_transz(A_nk, NULL, &A_tran);
00563 // It is equivalent to do transpose and then sort
00564 // fasp_dcsr_trans(A_nk, &A_tran);
00565 // fasp_dcsr_sort(&A_tran);
00566 precd_data.A_nk = &A_tran;
00567 #else
00568 precd_data.A_nk = A_nk;
00569 #endif
00570
00571 precd_data.P_nk = P_nk;
00572 precd_data.R_nk = R_nk;
00573
00574 if (status < 0) goto FINISHED;
00575
00576 precond prec;
00577 prec.data = &precd_data;
00578
00579 prec.fct = fasp_precond_dbsr_amg_nk;
00580
00581 fasp_gettime(&setup_end);
00582
00583 setup_time = setup_end - setup_start;
00584
00585 if (prtlvl >= PRINT_MIN) fasp_cputime("BSR AMG setup", setup_time);

```



```

00586
00587 //-----
00588 // Part 3:  solver
00589 //-----
00590 fasp_gettime(&solve_start);
00591
00592 status=fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);
00593
00594 fasp_gettime(&solve_end);
00595
00596 solve_time = solve_end - solve_start;
00597
00598 if ( prtlvl >= PRINT_MIN ) {
00599     fasp_cputime("BSR Krylov method", setup_time+solve_time);
00600 }
00601
00602 FINISHED:
00603     fasp_amg_data_bsr_free(mgl);
00604
00605 #if DEBUG_MODE > 0
00606     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00607 #endif
00608
00609 #if WITH_UMFPACK // use UMFPACK directly
00610     fasp_dcsr_free(&A_tran);
00611 #endif
00612 if (status == ERROR_ALLOC_MEM) goto MEMORY_ERROR;
00613 return status;
00614
00615 MEMORY_ERROR:
00616     printf("### ERROR: Cannot allocate memory! [%s]\n", __FUNCTION__);
00617     exit(status);
00618 }
00619
00640 INT fasp_solver_dbsr_krylov_nk_amg (dBSRmat    *A,
00641                                   dvector    *b,
00642                                   dvector    *x,
00643                                   ITS_param  *itparam,
00644                                   AMG_param  *amgparam,
00645                                   const INT  nk_dim,
00646                                   dvector    *nk)
00647 {
00648     //-----
00649     // Part 1:  prepare
00650     // -----
00651     const SHORT prtlvl = itparam->print_level;
00652     const SHORT max_levels = amgparam->max_levels;
00653
00654     // local variable
00655     INT i;
00656
00657     // return variable
00658     INT status = FASP_SUCCESS;
00659
00660     // data of AMG
00661     AMG_data_bsr *mgl=fasp_amg_data_bsr_create(max_levels);
00662
00663     // timing
00664     REAL setup_start, setup_end, setup_time;
00665     REAL solve_start, solve_end, solve_time;
00666
00667     #if DEBUG_MODE > 0
00668         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00669     #endif
00670
00671     //-----
00672     //Part 2:  set up the preconditioner
00673     //-----
00674     fasp_gettime(&setup_start);
00675
00676     // initialize A, b, x for mgl[0]
00677     mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
00678     fasp_dbsr_cp(A, &mgl[0].A);
00679     mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
00680     mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);
00681
00682     /*-----*/
00683     /*-- setup null spaces --*/
00684     /*-----*/
00685
00686     // null space for whole Jacobian
00687

```

```

00688     mgl[0].near_kernel_dim    = nk_dim;
00689     mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));
00690
00691     for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = nk[i].val;
00692
00693     switch (amgparam->AMG_type) {
00694     case SA_AMG: // Smoothed Aggregation AMG
00695         status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;
00696
00697     default:
00698         status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;
00699
00700     }
00701
00702     if (status < 0) goto FINISHED;
00703
00704     precondition_data_bsr precondition_data;
00705     precondition_data.print_level = amgparam->print_level;
00706     precondition_data.maxit = amgparam->maxit;
00707     precondition_data.tol = amgparam->tol;
00708     precondition_data.cycle_type = amgparam->cycle_type;
00709     precondition_data.smoother = amgparam->smoother;
00710     precondition_data.presmooth_iter = amgparam->presmooth_iter;
00711     precondition_data.postsmooth_iter = amgparam->postsmooth_iter;
00712     precondition_data.coarsening_type = amgparam->coarsening_type;
00713     precondition_data.relaxation = amgparam->relaxation;
00714     precondition_data.coarse_scaling = amgparam->coarse_scaling;
00715     precondition_data.aml_i_degree = amgparam->aml_i_degree;
00716     precondition_data.aml_i_coef = amgparam->aml_i_coef;
00717     precondition_data.tentative_smooth = amgparam->tentative_smooth;
00718     precondition_data.max_levels = mgl[0].num_levels;
00719     precondition_data.mgl_data = mgl;
00720     precondition_data.A = A;
00721
00722     if (status < 0) goto FINISHED;
00723
00724     precondition prec;
00725     prec.data = &precondition_data;
00726     switch (amgparam->cycle_type) {
00727     case NL_AMLI_CYCLE: // Nonlinear AMLI AMG
00728         prec.fct = fasp_precond_dbsr_namli;
00729         break;
00730
00731     default: // V,W-Cycle AMG
00732         prec.fct = fasp_precond_dbsr_amg;
00733         break;
00734     }
00735
00736     fasp_gettime(&setup_end);
00737
00738     setup_time = setup_end - setup_start;
00739
00740     if ( prtlvl >= PRINT_MIN ) fasp_cputime("BSR AMG setup", setup_time);
00741
00742     //-----
00743     // Part 3: solver
00744     //-----
00745     fasp_gettime(&solve_start);
00746
00747     status=fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);
00748
00749     fasp_gettime(&solve_end);
00750
00751     solve_time = solve_end - solve_start;
00752
00753     if ( prtlvl >= PRINT_MIN ) {
00754         fasp_cputime("BSR Krylov method", setup_time+solve_time);
00755     }
00756
00757     FINISHED:
00758     fasp_amg_data_bsr_free(mgl);
00759
00760     #if DEBUG_MODE > 0
00761     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00762     #endif
00763
00764     if (status == ERROR_ALLOC_MEM) goto MEMORY_ERROR;
00765     return status;
00766
00767     MEMORY_ERROR:
00768     printf("### ERROR: Cannot allocate memory! [%s]\n", __FUNCTION__);

```

```

00769     exit(status);
00770 }
00771
00772 /*-----*/
00773 /*--      End of File      --*/
00774 /*-----*/

```

9.181 SolCSR.c File Reference

Iterative solvers for [dCSRmat](#) matrices.

```

#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dcsr_itsolver](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax=b$ by preconditioned Krylov methods for CSR matrices.
- [INT fasp_solver_dcsr_itsolver_s](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax=b$ by preconditioned Krylov methods with safe-net for CSR matrices.
- [INT fasp_solver_dcsr_krylov](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods for CSR matrices.
- [INT fasp_solver_dcsr_krylov_s](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods with safe-net for CSR matrices.
- [INT fasp_solver_dcsr_krylov_diag](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by diagonal preconditioned Krylov methods.
- [INT fasp_solver_dcsr_krylov_swz](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [SWZ_param](#) *schparam)
Solve $Ax=b$ by overlapping Schwarz Krylov methods.
- [INT fasp_solver_dcsr_krylov_amg](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam)
Solve $Ax=b$ by AMG preconditioned Krylov methods.
- [INT fasp_solver_dcsr_krylov_ilu](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [ILU_param](#) *iluparam)
Solve $Ax=b$ by ILUs preconditioned Krylov methods.
- [INT fasp_solver_dcsr_krylov_ilu_M](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [ILU_param](#) *iluparam, [dCSRmat](#) *M)
Solve $Ax=b$ by ILUs preconditioned Krylov methods: ILU of M as preconditioner.
- [INT fasp_solver_dcsr_krylov_amg_nk](#) ([dCSRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam, [AMG_param](#) *amgparam, [dCSRmat](#) *A_nk, [dCSRmat](#) *P_nk, [dCSRmat](#) *R_nk)
Solve $Ax=b$ by AMG preconditioned Krylov methods with an extra near kernel solve.

9.181.1 Detailed Description

Iterative solvers for [dCSRmat](#) matrices.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMemory.c](#), [AuxMessage.c](#), [AuxParam.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaILUSetupCSR.c](#), [BlaSchwarzSetup.c](#), [BlaSparseCheck.c](#), [BlaSparseCSR.c](#), [KryPbcgs.c](#), [KryPcg.c](#), [KryPgcr.c](#), [KryPgmres.c](#), [KryPminres.c](#), [KryPvfgmres.c](#), [KryPvgmres.c](#), [PreAMGSetupRS.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreCSR.c](#), and [PreDataInit.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolCSR.c](#).

9.181.2 Function Documentation

9.181.2.1 fasp_solver_dcsr_itsolver()

```
INT fasp_solver_dcsr_itsolver (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax=b$ by preconditioned Krylov methods for CSR matrices.

Note

This is an abstract interface for iterative methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

09/25/2009

Definition at line 56 of file [SolCSR.c](#).

9.181.2.2 fasp_solver_dcsr_itsolver_s()

```
INT fasp_solver_dcsr_itsolver_s (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax=b$ by preconditioned Krylov methods with safe-net for CSR matrices.

Note

This is an abstract interface for iterative methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

10/21/2017

Definition at line [158](#) of file [SolCSR.c](#).

9.181.2.3 fasp_solver_dcsr_krylov()

```
INT fasp_solver_dcsr_krylov (  
    dCSRmat * A,  
    dvector * b,  
    dvector * x,  
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods for CSR matrices.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang, Shiquan Zhang

Date

09/25/2009

Definition at line 245 of file [SolCSR.c](#).**9.181.2.4 fasp_solver_dcsr_krylov_amg()**

```

INT fasp_solver_dcsr_krylov_amg (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam )

```

Solve $Ax=b$ by AMG preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters for AMG methods

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

09/25/2009

Definition at line 483 of file [SolCSR.c](#).**9.181.2.5 fasp_solver_dcsr_krylov_amg_nk()**

```

INT fasp_solver_dcsr_krylov_amg_nk (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    AMG_param * amgparam,
    dCSRmat * A_nk,
    dCSRmat * P_nk,
    dCSRmat * R_nk )

```

Solve $Ax=b$ by AMG preconditioned Krylov methods with an extra near kernel solve.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
----------	---

Parameters

<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>amgparam</i>	Pointer to parameters for AMG methods
<i>A_nk</i>	Pointer to the coeff matrix of near kernel space in dCSRmat format
<i>P_nk</i>	Pointer to the prolongation of near kernel space in dCSRmat format
<i>R_nk</i>	Pointer to the restriction of near kernel space in dCSRmat format

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line [759](#) of file [SolCSR.c](#).

9.181.2.6 fasp_solver_dcsr_krylov_diag()

```

INT fasp_solver_dcsr_krylov_diag (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )

```

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang, Shiquan Zhang

Date

09/25/2009

Definition at line [343](#) of file [SolCSR.c](#).

9.181.2.7 fasp_solver_dcsr_krylov_ilu()

```

INT fasp_solver_dcsr_krylov_ilu (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    ILU_param * iluparam )

```

Solve $Ax=b$ by ILUs preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>iluparam</i>	Pointer to parameters for ILU

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang, Shiquan Zhang

Date

09/25/2009

Definition at line [593](#) of file [SolCSR.c](#).

9.181.2.8 fasp_solver_dcsr_krylov_ilu_M()

```

INT fasp_solver_dcsr_krylov_ilu_M (
    dCSRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    ILU_param * iluparam,
    dCSRmat * M )

```

Solve $Ax=b$ by ILUs preconditioned Krylov methods: ILU of M as preconditioner.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>iluparam</i>	Pointer to parameters for ILU
<i>M</i>	Pointer to the preconditioning matrix in dCSRmat format

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

09/25/2009

Note

This function is specially designed for reservoir simulation. Have not been tested in any other places.

Definition at line 676 of file [SolCSR.c](#).

9.181.2.9 fasp_solver_dcsr_krylov_s()

```
INT fasp_solver_dcsr_krylov_s (  
    dCSRmat * A,  
    dvector * b,  
    dvector * x,  
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods with safe-net for CSR matrices.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

10/22/2017

Definition at line 294 of file [SolCSR.c](#).

9.181.2.10 fasp_solver_dcsr_krylov_swz()

```
INT fasp_solver_dcsr_krylov_swz (  
    dCSRmat * A,  
    dvector * b,  
    dvector * x,
```

```
ITS_param * itparam,
SWZ_param * schparam )
```

Solve $Ax=b$ by overlapping Schwarz Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dCSRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>schparam</i>	Pointer to parameters for Schwarz methods

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

03/21/2011

Modified by Chensong on 07/02/2012: change interface

Definition at line [405](#) of file [SolCSR.c](#).

9.182 SolCSR.c

[Go to the documentation of this file.](#)

```
00001
00018 #include <time.h>
00019
00020 #ifdef _OPENMP
00021 #include <omp.h>
00022 #endif
00023
00024 #include "fasp.h"
00025 #include "fasp_funcs.h"
00026
00027 /*-----*/
00028 /*--  Declare Private Functions  --*/
00029 /*-----*/
00030
00031 #include "KryUtil.inl"
00032
00033 /*-----*/
00034 /*--      Public Functions      --*/
00035 /*-----*/
00036
00056 INT fasp_solver_dcsr_itsolver (dCSRmat    *A,
00057                               dvector    *b,
00058                               dvector    *x,
00059                               precondition *pc,
00060                               ITS_param  *itparam)
00061 {
00062     const SHORT prtlvl      = itparam->print_level;
00063     const SHORT itsolver_type = itparam->itsolver_type;
00064     const SHORT stop_type    = itparam->stop_type;
00065     const SHORT restart      = itparam->restart;
00066     const INT   MaxIt        = itparam->maxit;
00067     const REAL  tol          = itparam->tol;
00068
00069     /* Local Variables */
00070     REAL solve_start, solve_end;
```

```

00071     INT iter;
00072
00073 #if DEBUG_MODE > 0
00074     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00075     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00076 #endif
00077     fasp_gettime(&solve_start);
00078
00079     /* check matrix data */
00080     fasp_check_dCSRmat(A);
00081
00082     /* Safe-guard checks on parameters */
00083     ITS_CHECK ( MaxIt, tol );
00084
00085     /* Choose a desirable Krylov iterative solver */
00086     switch ( itsolver_type ) {
00087     case SOLVER_CG:
00088         iter = fasp_solver_dcsr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00089         break;
00090
00091     case SOLVER_BiCGstab:
00092         iter = fasp_solver_dcsr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00093         break;
00094
00095     case SOLVER_MinRes:
00096         iter = fasp_solver_dcsr_pminres(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00097         break;
00098
00099     case SOLVER_GMRES:
00100         iter = fasp_solver_dcsr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00101         break;
00102
00103     case SOLVER_VGMRES:
00104         iter = fasp_solver_dcsr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00105         break;
00106
00107     case SOLVER_VFGMRES:
00108         iter = fasp_solver_dcsr_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00109         break;
00110
00111     case SOLVER_GCG:
00112         iter = fasp_solver_dcsr_pgcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00113         break;
00114
00115     case SOLVER_GCR:
00116         iter = fasp_solver_dcsr_pgcr(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00117         break;
00118
00119     default:
00120         printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
00121             itsolver_type, __FUNCTION__);
00122         return ERROR_SOLVER_TYPE;
00123     }
00124
00125     if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
00126         fasp_gettime(&solve_end);
00127         fasp_cputime("Iterative method", solve_end - solve_start);
00128     }
00129
00130 #if DEBUG_MODE > 0
00131     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00132 #endif
00133     return iter;
00134 }
00135
00136 INT fasp_solver_dcsr_itsolver_s (dCSRmat      *A,
00137                                dvector      *b,
00138                                dvector      *x,
00139                                precondition *pc,
00140                                ITS_param    *itparam)
00141 {
00142     const SHORT prtlvl      = itparam->print_level;
00143     const SHORT itsolver_type = itparam->itsolver_type;
00144     const SHORT stop_type    = itparam->stop_type;
00145     const SHORT restart      = itparam->restart;
00146     const INT   MaxIt        = itparam->maxit;
00147     const REAL  tol          = itparam->tol;
00148 }

```

```

00171      /* Local Variables */
00172      REAL solve_start, solve_end;
00173      INT iter;
00174
00175      #if DEBUG_MODE > 0
00176          printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00177          printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00178      #endif
00179
00180      fasp_gettime(&solve_start);
00181
00182      /* check matrix data */
00183      fasp_check_dCSRmat(A);
00184
00185      /* Safe-guard checks on parameters */
00186      ITS_CHECK ( MaxIt, tol );
00187
00188      /* Choose a desirable Krylov iterative solver */
00189      switch ( itsolver_type ) {
00190          case SOLVER_CG:
00191              iter = fasp_solver_dcsr_spcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00192              break;
00193
00194          case SOLVER_BiCGstab:
00195              iter = fasp_solver_dcsr_spgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00196              break;
00197
00198          case SOLVER_MinRes:
00199              iter = fasp_solver_dcsr_spmr(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00200              break;
00201
00202          case SOLVER_GMRES:
00203              iter = fasp_solver_dcsr_spgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00204              break;
00205
00206          case SOLVER_VGMRES:
00207              iter = fasp_solver_dcsr_spgvmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00208              break;
00209
00210          default:
00211              printf("### ERROR: Unknown iterative solver type %d! [%s]\n",
00212                  itsolver_type, __FUNCTION__);
00213              return ERROR_SOLVER_TYPE;
00214      }
00215
00216      if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
00217          fasp_gettime(&solve_end);
00218          fasp_cputime("Iterative method", solve_end - solve_start);
00219      }
00220
00221      #if DEBUG_MODE > 0
00222          printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00223      #endif
00224      return iter;
00225  }
00226
00227  INT fasp_solver_dcsr_krylov (dCSRmat *A,
00228                             dvector *b,
00229                             dvector *x,
00230                             ITS_param *itparam)
00231  {
00232      const SHORT prtlvl = itparam->print_level;
00233
00234      /* Local Variables */
00235      INT status = FASP_SUCCESS;
00236      REAL solve_start, solve_end;
00237
00238      #if DEBUG_MODE > 0
00239          printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00240          printf("### DEBUG: matrix size: %d %d %d\n", A->row, A->col, A->nnz);
00241          printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00242      #endif
00243
00244      fasp_gettime(&solve_start);
00245
00246      status = fasp_solver_dcsr_itsolver(A,b,x,NULL,itparam);
00247
00248      if ( prtlvl >= PRINT_MIN ) {
00249          fasp_gettime(&solve_end);

```

```

00268         fasp_cputime("Krylov method totally", solve_end - solve_start);
00269     }
00270
00271 #if DEBUG_MODE > 0
00272     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00273 #endif
00274
00275     return status;
00276 }
00277
00294 INT fasp_solver_dcsr_krylov_s (dCSRmat    *A,
00295                               dvector    *b,
00296                               dvector    *x,
00297                               ITS_param  *itparam)
00298 {
00299     const SHORT prtlvl = itparam->print_level;
00300
00301     /* Local Variables */
00302     INT      status = FASP_SUCCESS;
00303     REAL     solve_start, solve_end;
00304
00305 #if DEBUG_MODE > 0
00306     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00307     printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
00308     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00309 #endif
00310
00311     fasp_gettime(&solve_start);
00312
00313     status = fasp_solver_dcsr_itsolver_s (A,b,x,NULL,itparam);
00314
00315     if ( prtlvl >= PRINT_MIN ) {
00316         fasp_gettime(&solve_end);
00317         fasp_cputime("Krylov method totally", solve_end - solve_start);
00318     }
00319
00320 #if DEBUG_MODE > 0
00321     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00322 #endif
00323
00324     return status;
00325 }
00326
00343 INT fasp_solver_dcsr_krylov_diag (dCSRmat    *A,
00344                                   dvector    *b,
00345                                   dvector    *x,
00346                                   ITS_param  *itparam)
00347 {
00348     const SHORT prtlvl = itparam->print_level;
00349
00350     /* Local Variables */
00351     INT      status = FASP_SUCCESS;
00352     REAL     solve_start, solve_end;
00353
00354 #if DEBUG_MODE > 0
00355     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00356     printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
00357     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00358 #endif
00359
00360     fasp_gettime(&solve_start);
00361
00362     // setup preconditioner
00363     dvector diag; fasp_dcsr_getdiag(0,A,&diag);
00364
00365     precondition pc;
00366     pc.data = &diag;
00367     pc.fct  = fasp_precond_diag;
00368
00369     // call iterative solver
00370     status = fasp_solver_dcsr_itsolver(A,b,x,&pc,itparam);
00371
00372     if ( prtlvl >= PRINT_MIN ) {
00373         fasp_gettime(&solve_end);
00374         fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);
00375     }
00376
00377     fasp_dvec_free(&diag);
00378
00379 #if DEBUG_MODE > 0
00380     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);

```

```

00381 #endif
00382
00383     return status;
00384 }
00385
00405 INT fasp_solver_dcsr_krylov_swz (dCSRmat    *A,
00406                                dvector     *b,
00407                                dvector     *x,
00408                                ITS_param   *itparam,
00409                                SWZ_param   *schparam)
00410 {
00411     SWZ_param swzparam;
00412     swzparam.SWZ_mmsize = schparam->SWZ_mmsize;
00413     swzparam.SWZ_maxlvl = schparam->SWZ_maxlvl;
00414     swzparam.SWZ_type = schparam->SWZ_type;
00415     swzparam.SWZ_blksolver = schparam->SWZ_blksolver;
00416
00417     const SHORT prtlvl = itparam->print_level;
00418
00419     REAL setup_start, setup_end;
00420     REAL solve_start, solve_end;
00421     INT status = FASP_SUCCESS;
00422
00423     #if DEBUG_MODE > 0
00424         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00425         printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
00426         printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00427     #endif
00428
00429     fasp_gettime(&solve_start);
00430     fasp_gettime(&setup_start);
00431
00432     // setup preconditioner
00433     SWZ_data SWZ_data;
00434
00435     // symmetrize the matrix (get rid of this later)
00436     SWZ_data.A = fasp_dcsr_sympart(A);
00437
00438     // construct Schwarz preconditioner
00439     fasp_dcsr_shift (&SWZ_data.A, 1);
00440     fasp_swz_dcsr_setup (&SWZ_data, &swzparam);
00441
00442     fasp_gettime (&setup_end);
00443     printf("SWZ-Krylov method setup %f seconds.\n", setup_end - setup_start);
00444
00445     precondition prec;
00446     prec.data = &SWZ_data;
00447     prec.fct = fasp_precond_swz;
00448
00449     // solver part
00450     status = fasp_solver_dcsr_itsolver(A,b,x,&prec,itparam);
00451
00452     if ( prtlvl > PRINT_NONE ) {
00453         fasp_gettime(&solve_end);
00454         fasp_cputime("SWZ-Krylov method totally", solve_end - solve_start);
00455     }
00456
00457     #if DEBUG_MODE > 0
00458         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00459     #endif
00460
00461     fasp_swz_data_free(&SWZ_data);
00462
00463     return status;
00464 }
00465
00483 INT fasp_solver_dcsr_krylov_amg (dCSRmat    *A,
00484                                dvector     *b,
00485                                dvector     *x,
00486                                ITS_param   *itparam,
00487                                AMG_param   *amgparam)
00488 {
00489     const SHORT prtlvl = itparam->print_level;
00490     const SHORT max_levels = amgparam->max_levels;
00491     const INT nnz = A->nnz, m = A->row, n = A->col;
00492
00493     /* Local Variables */
00494     INT status = FASP_SUCCESS;
00495     REAL solve_start, solve_end;
00496
00497     #if MULTI_COLOR_ORDER

```

```

00498     A->color = 0;
00499     A->IC = NULL;
00500     A->ICMAP = NULL;
00501 #endif
00502
00503 #if DEBUG_MODE > 0
00504     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00505     printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
00506     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00507 #endif
00508
00509     fasp_gettime(&solve_start);
00510
00511     // initialize A, b, x for mgl[0]
00512     AMG_data *mgl=fasp_amg_data_create(max_levels);
00513     mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
00514     mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);
00515
00516     // setup preconditioner
00517     switch (amgparam->AMG_type) {
00518
00519         case SA_AMG: // Smoothed Aggregation AMG
00520             status = fasp_amg_setup_sa(mgl, amgparam); break;
00521
00522         case UA_AMG: // Unsmoothed Aggregation AMG
00523             status = fasp_amg_setup_ua(mgl, amgparam); break;
00524
00525         default: // Classical AMG
00526             status = fasp_amg_setup_rs(mgl, amgparam);
00527     }
00528 }
00529
00530 #if DEBUG_MODE > 1
00531     fasp_mem_usage();
00532 #endif
00533
00534     if (status < 0) goto FINISHED;
00535
00536     // setup preconditioner
00537     precondition_data pcd;
00538     fasp_param_amg_to_prec(&pcd, amgparam);
00539     pcd.max_levels = mgl[0].num_levels;
00540     pcd.mgl_data = mgl;
00541
00542     precondition pc; pc.data = &pcd;
00543
00544     if (itparam->precond_type == PREC_FMG) {
00545         pc.fct = fasp_precond_famg; // Full AMG
00546     }
00547     else {
00548         switch (amgparam->cycle_type) {
00549             case AMLI_CYCLE: // AMLI cycle
00550                 pc.fct = fasp_precond_aml; break;
00551             case NL_AMLI_CYCLE: // Nonlinear AMLI
00552                 pc.fct = fasp_precond_naml; break;
00553             default: // V,W-cycles or hybrid cycles
00554                 pc.fct = fasp_precond_amg;
00555         }
00556     }
00557
00558     // call iterative solver
00559     status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);
00560
00561     if (prtlvl >= PRINT_MIN) {
00562         fasp_gettime(&solve_end);
00563         fasp_cputime("AMG_Krylov method totally", solve_end - solve_start);
00564     }
00565
00566 FINISHED:
00567     fasp_amg_data_free(mgl, amgparam);
00568
00569 #if DEBUG_MODE > 0
00570     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00571 #endif
00572
00573     return status;
00574 }
00575
00593 INT fasp_solver_dcsr_krylov_ilu (dCSRmat *A,
00594                                dvector *b,
00595                                dvector *x,

```

```

00596                                     ITS_param *itparam,
00597                                     ILU_param *iluparam)
00598 {
00599     const SHORT prtlvl = itparam->print_level;
00600
00601     /* Local Variables */
00602     INT status = FASP_SUCCESS;
00603     REAL solve_start, solve_end, solve_time;
00604
00605     #if DEBUG_MODE > 0
00606         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00607         printf("### DEBUG: matrix size: %d %d %d\n", A->row, A->col, A->nnz);
00608         printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00609     #endif
00610
00611     fasp_gettime(&solve_start);
00612
00613     // ILU setup for whole matrix
00614     ILU_data LU;
00615     if ( (status = fasp_ilu_dcsr_setup(A, &LU, iluparam)) < 0 ) goto FINISHED;
00616
00617     // check iludata
00618     if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;
00619
00620     // set preconditioner
00621     precondition pc;
00622     pc.data = &LU;
00623     pc.fct = fasp_precond_ilu;
00624
00625     // call iterative solver
00626     status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);
00627
00628     if ( prtlvl >= PRINT_MIN ) {
00629         fasp_gettime(&solve_end);
00630         solve_time = solve_end - solve_start;
00631
00632         switch (iluparam->ILU_type) {
00633             case ILUt:
00634                 fasp_cputime("ILUt_Krylov method totally", solve_time);
00635                 break;
00636             case ILUtp:
00637                 fasp_cputime("ILUtp_Krylov method totally", solve_time);
00638                 break;
00639             default: // ILUk
00640                 fasp_cputime("ILUk_Krylov method totally", solve_time);
00641         }
00642     }
00643
00644     FINISHED:
00645     fasp_ilu_data_free(&LU);
00646
00647     #if DEBUG_MODE > 0
00648         printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00649     #endif
00650
00651     return status;
00652 }
00653
00654 INT fasp_solver_dcsr_krylov_ilu_M (dCSRmat *A,
00655                                  dvector *b,
00656                                  dvector *x,
00657                                  ITS_param *itparam,
00658                                  ILU_param *iluparam,
00659                                  dCSRmat *M)
00660 {
00661     const SHORT prtlvl = itparam->print_level;
00662
00663     /* Local Variables */
00664     REAL solve_start, solve_end, solve_time;
00665     INT status = FASP_SUCCESS;
00666
00667     #if DEBUG_MODE > 0
00668         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00669         printf("### DEBUG: matrix size: %d %d %d\n", A->row, A->col, A->nnz);
00670         printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00671     #endif
00672
00673     fasp_gettime(&solve_start);
00674
00675     // ILU setup for M
00676     ILU_data LU;

```



```

00699     if ( (status = fasp_ilu_dcsr_setup(M, &LU, iluparam)) < 0 ) goto FINISHED;
00700
00701     // check iludata
00702     if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;
00703
00704     // set preconditioner
00705     precondition pc;
00706     pc.data = &LU;
00707     pc.fct = fasp_precond_ilu;
00708
00709     // call iterative solver
00710     status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);
00711
00712     if ( prtlvl >= PRINT_MIN ) {
00713         fasp_gettime(&solve_end);
00714         solve_time = solve_end - solve_start;
00715
00716         switch (iluparam->ILU_type) {
00717             case ILUt:
00718                 fasp_cputime("ILUt_Krylov method", solve_time);
00719                 break;
00720             case ILUtp:
00721                 fasp_cputime("ILUtp_Krylov method", solve_time);
00722                 break;
00723             default: // ILUk
00724                 fasp_cputime("ILUk_Krylov method", solve_time);
00725         }
00726     }
00727
00728 FINISHED:
00729     fasp_ilu_data_free(&LU);
00730
00731 #if DEBUG_MODE > 0
00732     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00733 #endif
00734
00735     return status;
00736 }
00737
00759 INT fasp_solver_dcsr_krylov_amg_nk (dCSRmat *A,
00760                                     dvector *b,
00761                                     dvector *x,
00762                                     ITS_param *itparam,
00763                                     AMG_param *amgparam,
00764                                     dCSRmat *A_nk,
00765                                     dCSRmat *P_nk,
00766                                     dCSRmat *R_nk)
00767 {
00768     const SHORT prtlvl = itparam->print_level;
00769     const SHORT max_levels = amgparam->max_levels;
00770     const INT nnz = A->nnz, m = A->row, n = A->col;
00771
00772     /* Local Variables */
00773     INT status = FASP_SUCCESS;
00774     REAL solve_start, solve_end, solve_time;
00775
00776 #if MULTI_COLOR_ORDER
00777     A->color = 0;
00778     A->IC = NULL;
00779     A->ICMAP = NULL;
00780 #endif
00781
00782 #if DEBUG_MODE > 0
00783     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00784     printf("### DEBUG: matrix size: %d %d %d\n", A->row, A->col, A->nnz);
00785     printf("### DEBUG: rhs/sol size: %d %d\n", b->row, x->row);
00786 #endif
00787
00788     fasp_gettime(&solve_start);
00789
00790     // initialize A, b, x for mgl[0]
00791     AMG_data *mgl=fasp_amg_data_create(max_levels);
00792     mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
00793     mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);
00794
00795     // setup preconditioner
00796     switch (amgparam->AMG_type) {
00797
00798         case SA_AMG: // Smoothed Aggregation AMG
00799             status = fasp_amg_setup_sa(mgl, amgparam); break;
00800

```

```

00801         case UA_AMG: // Unsmoothed Aggregation AMG
00802             status = fasp_amg_setup_ua(mgl, amgparam); break;
00803
00804         default: // Classical AMG
00805             status = fasp_amg_setup_rs(mgl, amgparam);
00806     }
00807 }
00808
00809 #if DEBUG_MODE > 1
00810     fasp_mem_usage();
00811 #endif
00812
00813     if (status < 0) goto FINISHED;
00814
00815     // setup preconditioner
00816     precondition_data pcd;
00817     fasp_param_amg_to_prec(&pcd, amgparam);
00818     pcd.max_levels = mgl[0].num_levels;
00819     pcd.mgl_data = mgl;
00820
00821     // near kernel space
00822 #if WITH_UMFPACK // use UMFPACK directly
00823     dCSRmat A_tran;
00824     A_tran = fasp_dcsr_create(A_nk->row, A_nk->col, A_nk->nnz);
00825     fasp_dcsr_transz(A_nk, NULL, &A_tran);
00826     // It is equivalent to do transpose and then sort
00827     // fasp_dcsr_trans(A_nk, &A_tran);
00828     // fasp_dcsr_sort(&A_tran);
00829     pcd.A_nk = &A_tran;
00830 #else
00831     pcd.A_nk = A_nk;
00832 #endif
00833
00834     pcd.P_nk = P_nk;
00835     pcd.R_nk = R_nk;
00836
00837     precondition pc; pc.data = &pcd;
00838     pc.fct = fasp_precond_amg_nk;
00839
00840     // call iterative solver
00841     status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);
00842
00843     if (prtlvl >= PRINT_MIN) {
00844         fasp_gettime(&solve_end);
00845         solve_time = solve_end - solve_start;
00846         fasp_cputime("AMG_NK_Krylov method", solve_time);
00847     }
00848
00849 FINISHED:
00850     fasp_amg_data_free(mgl, amgparam);
00851
00852 #if WITH_UMFPACK // use UMFPACK directly
00853     fasp_dcsr_free(&A_tran);
00854 #endif
00855
00856 #if DEBUG_MODE > 0
00857     printf("### DEBUG: [--End--] %s ...\\n", __FUNCTION__);
00858 #endif
00859
00860     return status;
00861 }
00862
00863 /*-----*/
00864 /*--      End of File      --*/
00865 /*-----*/

```

9.183 SoIFAMG.c File Reference

Full AMG method as an iterative solver.

```

#include <time.h>
#include "fasp.h"
#include "fasp_funcs.h"

```

Functions

- void [fasp_solver_famg](#) (const [dCSRmat](#) *A, const [dvector](#) *b, [dvector](#) *x, [AMG_param](#) *param)
Solve $Ax=b$ by full AMG.

9.183.1 Detailed Description

Full AMG method as an iterative solver.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSparseCheck.c](#), [BlaSparseCSR.c](#), [PreAMGSetupRS.c](#), [PreAMGSetupSA.c](#), [PreAMGSetupUA.c](#), [PreDataInit.c](#), and [PreMGsSolve.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolFAMG.c](#).

9.183.2 Function Documentation

9.183.2.1 fasp_solver_famg()

```
INT fasp_solver_famg (
    const dCSRmat * A,
    const dvector * b,
    dvector * x,
    AMG\_param * param )
```

Solve $Ax=b$ by full AMG.

Parameters

<i>A</i>	Pointer to dCSRmat : the coefficient matrix
<i>b</i>	Pointer to dvector : the right hand side
<i>x</i>	Pointer to dvector : the unknowns
<i>param</i>	Pointer to AMG_param : AMG parameters

Author

Xiaozhe Hu

Date

02/27/2011

Modified by Chensong Zhang on 05/05/2013: Remove error handling for AMG setup
Definition at line 41 of file [SolFAMG.c](#).

9.184 SolFAMG.c

[Go to the documentation of this file.](#)

```
00001
00016 #include <time.h>
```

```

00017
00018 #include "fasp.h"
00019 #include "fasp_functs.h"
00020
00021 /*-----*/
00022 /*--      Public Functions      --*/
00023 /*-----*/
00024
00041 void fasp_solver_famg (const dCSRmat *A,
00042                      const dvector *b,
00043                      dvector *x,
00044                      AMG_param *param)
00045 {
00046     const SHORT max_levels = param->max_levels;
00047     const SHORT prtlvl = param->print_level;
00048     const SHORT amg_type = param->AMG_type;
00049     const INT nnz = A->nnz, m = A->row, n = A->col;
00050
00051     // local variables
00052     AMG_data * mgl = fasp_amg_data_create(max_levels);
00053     REAL FMG_start = 0, FMG_end;
00054
00055     #if DEBUG_MODE > 0
00056     printf("###DEBUG: %s ..... [begin]\n", __FUNCTION__);
00057     printf("###DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
00058     #endif
00059
00060     if ( prtlvl > PRINT_NONE ) fasp_gettime(&FMG_start);
00061
00062     // check matrix data
00063     fasp_check_dCSRmat (A);
00064
00065     // Step 0: initialize mgl[0] with A, b and x
00066     mgl[0].A = fasp_dcsr_create(m,n,nnz);
00067     fasp_dcsr_cp(A,&mgl[0].A);
00068
00069     mgl[0].b = fasp_dvec_create(n);
00070     fasp_dvec_cp(b,&mgl[0].b);
00071
00072     mgl[0].x = fasp_dvec_create(n);
00073     fasp_dvec_cp(x,&mgl[0].x);
00074
00075     // Step 1: AMG setup phase
00076     switch (amg_type) {
00077
00078     case SA_AMG:
00079         // Smoothed Aggregation AMG setup phase
00080         fasp_amg_setup_sa(mgl, param); break;
00081
00082     case UA_AMG:
00083         // Unsmoothed Aggregation AMG setup phase
00084         fasp_amg_setup_ua(mgl, param); break;
00085
00086     default:
00087         // Classical AMG setup phase
00088         fasp_amg_setup_rs(mgl, param); break;
00089
00090     }
00091
00092     // Step 2: FAMG solve phase
00093     fasp_famg_solve(mgl, param);
00094
00095     // Step 3: Save solution vector and return
00096     fasp_dvec_cp(&mgl[0].x, x);
00097
00098     // clean-up memory
00099     fasp_amg_data_free(mgl, param);
00100
00101     // print out CPU time if needed
00102     if ( prtlvl > PRINT_NONE ) {
00103         fasp_gettime(&FMG_end);
00104         fasp_cputime("FAMG totally", FMG_end - FMG_start);
00105     }
00106
00107     #if DEBUG_MODE > 0
00108     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00109     #endif
00110
00111     return;
00112 }
00113

```

```

00114  /*-----*/
00115  /*--      End of File      --*/
00116  /*-----*/

```

9.185 SolGMGPoisson.c File Reference

GMG method as an iterative solver for Poisson Problem.

```

#include <time.h>
#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "PreGMG.inl"

```

Functions

- **INT fasp_poisson_gmg1d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method.
- **INT fasp_poisson_gmg2d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method.
- **INT fasp_poisson_gmg3d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** nz, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method.
- void **fasp_poisson_fgmg1d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method (FMG)
- void **fasp_poisson_fgmg2d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method (FMG)
- void **fasp_poisson_fgmg3d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** nz, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method (FMG)
- **INT fasp_poisson_gmgcg1d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)
- **INT fasp_poisson_gmgcg2d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)
- **INT fasp_poisson_gmgcg3d** (**REAL** *u, **REAL** *b, const **INT** nx, const **INT** ny, const **INT** nz, const **INT** maxlevel, const **REAL** rtol, const **SHORT** prtlvl)
Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)

9.185.1 Detailed Description

GMG method as an iterative solver for Poisson Problem.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxArray.c](#), [AuxMessage.c](#), and [AuxTiming.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolGMGPoisson.c](#).

9.185.2 Function Documentation

9.185.2.1 fasp_poisson_fgm1d()

```
void fasp_poisson_fgm1d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method (FMG)

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 442 of file [SolGMGPoisson.c](#).

9.185.2.2 fasp_poisson_fgm2d()

```
void fasp_poisson_fgm2d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT ny,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method (FMG)

Parameters

<i>u</i>	Pointer to the vector of dofs
----------	-------------------------------

Parameters

<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in Y direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 536 of file SolGMGPoisson.c.

9.185.2.3 fasp_poisson_fgmg3d()

```
void fasp_poisson_fgmg3d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT ny,
    const INT nz,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method (FMG)

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	NUmber of grids in y direction
<i>nz</i>	NUmber of grids in z direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 644 of file [SolGMGPoisson.c](#).

9.185.2.4 fasp_poisson_gmg1d()

```
INT fasp_poisson_gmg1d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method.

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 48 of file [SolGMGPoisson.c](#).

9.185.2.5 fasp_poisson_gmg2d()

```
INT fasp_poisson_gmg2d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT ny,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method.

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 172 of file [SolGMGPoisson.c](#).

9.185.2.6 fasp_poisson_gmg3d()

```
INT fasp_poisson_gmg3d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT ny,
    const INT nz,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method.

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>nz</i>	Number of grids in z direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 308 of file [SolGMGPoisson.c](#).

9.185.2.7 fasp_poisson_gmgcg1d()

```
INT fasp_poisson_gmgcg1d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )
```

Solve $Ax=b$ of Poisson 1D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 754 of file [SolGMGPoisson.c](#).

9.185.2.8 fasp_poisson_gmgcg2d()

```
INT fasp_poisson_gmgcg2d (
    REAL * u,
    REAL * b,
```

```

const INT nx,
const INT ny,
const INT maxlevel,
const REAL rtol,
const SHORT prtlvl )

```

Solve $Ax=b$ of Poisson 2D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 849 of file [SolGMGPoisson.c](#).

9.185.2.9 fasp_poisson_gmgcg3d()

```

INT fasp_poisson_gmgcg3d (
    REAL * u,
    REAL * b,
    const INT nx,
    const INT ny,
    const INT nz,
    const INT maxlevel,
    const REAL rtol,
    const SHORT prtlvl )

```

Solve $Ax=b$ of Poisson 3D equation by Geometric Multigrid Method (GMG preconditioned Conjugate Gradient method)

Parameters

<i>u</i>	Pointer to the vector of dofs
<i>b</i>	Pointer to the vector of right hand side
<i>nx</i>	Number of grids in x direction
<i>ny</i>	Number of grids in y direction
<i>nz</i>	Number of grids in z direction
<i>maxlevel</i>	Maximum levels of the multigrid
<i>rtol</i>	Relative tolerance to judge convergence
<i>prtlvl</i>	Print level for output

Returns

Iteration number if converges; ERROR otherwise.

Author

Ziteng Wang, Chensong Zhang

Date

06/07/2013

Definition at line 959 of file [SolGMGPoisson.c](#).

9.186 SolGMGPoisson.c

[Go to the documentation of this file.](#)

```

00001
00014 #include <time.h>
00015 #include <math.h>
00016
00017 #include "fasp.h"
00018 #include "fasp_funcs.h"
00019
00020 /*-----*/
00021 /*--  Declare Private Functions  --*/
00022 /*-----*/
00023
00024 #include "PreGMG.inl"
00025
00026 /*-----*/
00027 /*--      Public Functions      --*/
00028 /*-----*/
00029
00048 INT fasp_poisson_gmgld (REAL      *u,
00049                          REAL      *b,
00050                          const INT  nx,
00051                          const INT  maxlevel,
00052                          const REAL  rtol,
00053                          const SHORT prtlvl)
00054 {
00055     const REAL atol = 1.0E-15;
00056     const INT  max_itr_num = 100;
00057
00058     REAL      *u0, *r0, *b0;
00059     REAL      norm_r, norm_r0, norm_r1, factor, error = BIGREAL;
00060     INT        i, *level, count = 0;
00061     REAL      AMG_start = 0, AMG_end;
00062
00063     #if DEBUG_MODE > 0
00064         printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00065         printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
00066     #endif
00067
00068     if ( prtlvl > PRINT_NONE ) {
00069         fasp_gettime(&AMG_start);
00070         printf("Num of DOF's:  %d\n", nx+1);
00071     }
00072
00073     // set level
00074     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00075     level[0] = 0; level[1] = nx+1;
00076     for (i = 1; i < maxlevel; i++) {
00077         level[i+1] = level[i] + (level[i] - level[i-1] + 1) / 2;
00078     }
00079     level[maxlevel+1] = level[maxlevel] + 1;
00080
00081     // set u0, b0, r0
00082     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00083     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00084     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00085
00086     fasp_darray_set(level[maxlevel], u0, 0.0);
00087     fasp_darray_set(level[maxlevel], b0, 0.0);

```

```

00088     fasp_darray_set(level[maxlevel], r0, 0.0);
00089
00090     fasp_darray_cp(nx, u, u0);
00091     fasp_darray_cp(nx, b, b0);
00092
00093     // compute initial l2 norm of residue
00094     fasp_darray_set(level[1], r0, 0.0);
00095     residual1d(u0, b0, r0, 0, level);
00096     norm_r0 = l2norm(r0, level, 0);
00097     norm_r1 = norm_r0;
00098     if (norm_r0 < atol) goto FINISHED;
00099
00100     if ( prtlvl > PRINT_SOME ){
00101         printf("-----\n");
00102         printf("It Num | ||r||/||b|| | ||r|| | Conv. Factor\n");
00103         printf("-----\n");
00104     }
00105
00106     // GMG solver of V-cycle
00107     while (count < max_itr_num) {
00108         count++;
00109         mg1d(u0, b0, level, 0, maxlevel);
00110         residual1d(u0, b0, r0, 0, level);
00111         norm_r = l2norm(r0, level, 0);
00112         factor = norm_r/norm_r1;
00113         error = norm_r / norm_r0;
00114         norm_r1 = norm_r;
00115         if ( prtlvl > PRINT_SOME ){
00116             printf("%6d | %13.6e | %13.6e | %10.4f\n",count,error,norm_r,factor);
00117         }
00118         if (error < rtol || norm_r < atol) break;
00119     }
00120
00121     if ( prtlvl > PRINT_NONE ){
00122         if (count >= max_itr_num) {
00123             printf("### WARNING: V-cycle failed to converge.\n");
00124         }
00125         else {
00126             printf("Num of V-cycle's:  %d, Relative Residual = %e.\n", count, error);
00127         }
00128     }
00129
00130     // Update u
00131     fasp_darray_cp(level[1], u0, u);
00132
00133     // print out CPU time if needed
00134     if ( prtlvl > PRINT_NONE ) {
00135         fasp_gettime(&AMG_end);
00136         fasp_cputime("GMG totally", AMG_end - AMG_start);
00137     }
00138
00139 #if DEBUG_MODE > 0
00140     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00141 #endif
00142
00143 FINISHED:
00144     free(level);
00145     free(r0);
00146     free(u0);
00147     free(b0);
00148
00149     return count;
00150 }
00151
00172 INT fasp_poisson_gmg2d (REAL      *u,
00173                        REAL      *b,
00174                        const INT  nx,
00175                        const INT  ny,
00176                        const INT  maxlevel,
00177                        const REAL  rtol,
00178                        const SHORT prtlvl)
00179 {
00180     const REAL atol = 1.0E-15;
00181     const INT  max_itr_num = 100;
00182
00183     REAL *u0, *b0, *r0;
00184     REAL norm_r, norm_r0, norm_r1, factor, error = BIGREAL;
00185     INT i, k, count = 0, *nxk, *nyk, *level;
00186     REAL AMG_start = 0, AMG_end;
00187
00188 #if DEBUG_MODE > 0

```

```

00189     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00190     printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
00191 #endif
00192
00193     if ( prtlvl > PRINT_NONE ) {
00194         fasp_gettime(&AMG_start);
00195         printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
00196     }
00197
00198     // set nxk, nyk
00199     nxk = (INT *)malloc(maxlevel*sizeof(INT));
00200     nyk = (INT *)malloc(maxlevel*sizeof(INT));
00201     nxk[0] = nx+1; nyk[0] = ny+1;
00202     for (k=1; k<maxlevel; k++) {
00203         nxk[k] = (int) (nxk[k-1]+1)/2;
00204         nyk[k] = (int) (nyk[k-1]+1)/2;
00205     }
00206
00207     // set level
00208     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00209     level[0] = 0; level[1] = (nx+1)*(ny+1);
00210     for (i = 1; i < maxlevel; i++) {
00211         level[i+1] = level[i] + (nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
00212     }
00213     level[maxlevel+1] = level[maxlevel]+1;
00214
00215     // set u0, b0
00216     u0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));
00217     b0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));
00218     r0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));
00219
00220     fasp_darray_set(level[maxlevel], u0, 0.0);
00221     fasp_darray_set(level[maxlevel], b0, 0.0);
00222     fasp_darray_set(level[maxlevel], r0, 0.0);
00223
00224     fasp_darray_cp(level[1], u, u0);
00225     fasp_darray_cp(level[1], b, b0);
00226
00227     // compute initial l2 norm of residue
00228     residual2d(u0, b0, r0, 0, level, nxk, nyk);
00229     norm_r0 = l2norm(r0, level, 0);
00230     norm_r1 = norm_r0;
00231     if (norm_r0 < atol) goto FINISHED;
00232
00233     if ( prtlvl > PRINT_SOME ){
00234         printf("-----\n");
00235         printf("It Num | ||r||/||b|| | ||r|| | Conv. Factor\n");
00236         printf("-----\n");
00237     }
00238
00239     // GMG solver of V-cycle
00240     while ( count < max_itr_num ) {
00241         count++;
00242         mg2d(u0, b0, level, 0, maxlevel, nxk, nyk);
00243         residual2d(u0, b0, r0, 0, level, nxk, nyk);
00244         norm_r = l2norm(r0, level, 0);
00245         error = norm_r / norm_r0;
00246         factor = norm_r/norm_r1;
00247         norm_r1 = norm_r;
00248         if ( prtlvl > PRINT_SOME ){
00249             printf("%6d | %13.6e | %13.6e | %10.4f\n", count, error, norm_r, factor);
00250         }
00251         if ( error < rtol || norm_r < atol ) break;
00252     }
00253
00254     if ( prtlvl > PRINT_NONE ){
00255         if (count >= max_itr_num) {
00256             printf("### WARNING: V-cycle failed to converge.\n");
00257         }
00258         else {
00259             printf("Num of V-cycle's:  %d, Relative Residual = %e.\n", count, error);
00260         }
00261     }
00262
00263     // update u
00264     fasp_darray_cp(level[1], u0, u);
00265
00266     // print out CPU time if needed
00267     if ( prtlvl > PRINT_NONE ) {
00268         fasp_gettime(&AMG_end);
00269         fasp_cputime("GMG totally", AMG_end - AMG_start);

```

```

00270     }
00271
00272 #if DEBUG_MODE > 0
00273     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00274 #endif
00275
00276 FINISHED:
00277     free(level);
00278     free(nxk);
00279     free(nyk);
00280     free(u0);
00281     free(b0);
00282     free(r0);
00283
00284     return count;
00285 }
00286
00308 INT fasp_poisson_gmg3d (REAL      *u,
00309                        REAL      *b,
00310                        const INT  nx,
00311                        const INT  ny,
00312                        const INT  nz,
00313                        const INT  maxlevel,
00314                        const REAL rtol,
00315                        const SHORT prtlvl)
00316 {
00317     const REAL atol = 1.0E-15;
00318     const INT  max_itr_num = 100;
00319
00320     REAL      *u0, *r0, *b0;
00321     REAL      norm_r, norm_r0, norm_r1, factor, error = BIGREAL;
00322     INT       i, k, count = 0, *nxk, *nyk, *nz, *level;
00323     REAL      AMG_start = 0, AMG_end;
00324
00325 #if DEBUG_MODE > 0
00326     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00327     printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
00328           nx, ny, nz, maxlevel);
00329 #endif
00330
00331     if ( prtlvl > PRINT_NONE ) {
00332         fasp_gettime(&AMG_start);
00333         printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
00334     }
00335
00336     // set nxk, nyk, nz
00337     nxk = (INT *)malloc(maxlevel*sizeof(INT));
00338     nyk = (INT *)malloc(maxlevel*sizeof(INT));
00339     nzk = (INT *)malloc(maxlevel*sizeof(INT));
00340     nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
00341     for(k=1; k<maxlevel; k++){
00342         nxk[k] = (int) (nxk[k-1]+1)/2;
00343         nyk[k] = (int) (nyk[k-1]+1)/2;
00344         nzk[k] = (int) (nyk[k-1]+1)/2;
00345     }
00346
00347     // set level
00348     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00349     level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
00350     for (i = 1; i < maxlevel; i++) {
00351         level[i+1] = level[i] + (nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
00352     }
00353     level[maxlevel+1] = level[maxlevel]+1;
00354
00355     // set u0, b0, r0
00356     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00357     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00358     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00359     fasp_darray_set(level[maxlevel], u0, 0.0);
00360     fasp_darray_set(level[maxlevel], b0, 0.0);
00361     fasp_darray_set(level[maxlevel], r0, 0.0);
00362     fasp_darray_cp(level[1], u, u0);
00363     fasp_darray_cp(level[1], b, b0);
00364
00365     // compute initial l2 norm of residue
00366     residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
00367     norm_r0 = l2norm(r0, level, 0);
00368     norm_r1 = norm_r0;
00369     if (norm_r0 < atol) goto FINISHED;
00370
00371     if ( prtlvl > PRINT_SOME ) {

```

```

00372         printf("-----\n");
00373         printf("It Num | ||r||/||b|| | ||r|| | Conv. Factor\n");
00374         printf("-----\n");
00375     }
00376
00377     // GMG solver of V-cycle
00378     while (count < max_itr_num) {
00379         count++;
00380         mg3d(u0, b0, level, 0, maxlevel, nxk, nyk, nzk);
00381         residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
00382         norm_r = l2norm(r0, level, 0);
00383         factor = norm_r/norm_rl;
00384         error = norm_r / norm_r0;
00385         norm_rl = norm_r;
00386         if ( prtlvl > PRINT_SOME ){
00387             printf("%6d | %13.6e | %13.6e | %10.4f\n",count,error,norm_r,factor);
00388         }
00389         if (error < rtol || norm_r < atol) break;
00390     }
00391
00392     if ( prtlvl > PRINT_NONE ){
00393         if (count >= max_itr_num) {
00394             printf("### WARNING: V-cycle failed to converge.\n");
00395         }
00396         else {
00397             printf("Num of V-cycle's: %d, Relative Residual = %e.\n", count, error);
00398         }
00399     }
00400
00401     // update u
00402     fasp_darray_cp(level[1], u0, u);
00403
00404     // print out CPU time if needed
00405     if ( prtlvl > PRINT_NONE ) {
00406         fasp_gettime(&AMG_end);
00407         fasp_cputime("GMG totally", AMG_end - AMG_start);
00408     }
00409
00410     FINISHED:
00411     free(level);
00412     free(nxk);
00413     free(nyk);
00414     free(nzk);
00415     free(r0);
00416     free(u0);
00417     free(b0);
00418
00419     #if DEBUG_MODE > 0
00420     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00421     #endif
00422
00423     return count;
00424 }
00425
00442 void fasp_poisson_fgmglc (REAL          *u,
00443                          REAL          *b,
00444                          const INT      nx,
00445                          const INT      maxlevel,
00446                          const REAL     rtol,
00447                          const SHORT    prtlvl)
00448 {
00449     const REAL atol = 1.0E-15;
00450     REAL      *u0, *r0, *b0;
00451     REAL      norm_r0, norm_r;
00452     INT       *level;
00453     REAL      AMG_start = 0, AMG_end;
00454     int       i;
00455
00456     #if DEBUG_MODE > 0
00457     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00458     printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
00459     #endif
00460
00461     if ( prtlvl > PRINT_NONE ) {
00462         fasp_gettime(&AMG_start);
00463         printf("Num of DOF's: %d\n", (nx+1));
00464     }
00465
00466     // set level
00467     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00468     level[0] = 0; level[1] = nx+1;

```



```

00469     for (i = 1; i < maxlevel; i++) {
00470         level[i+1] = level[i] + (level[i] - level[i-1] + 1) / 2;
00471     }
00472     level[maxlevel+1] = level[maxlevel] + 1;
00473
00474     // set u0, b0, r0
00475     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00476     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00477     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00478     fasp_darray_set(level[maxlevel], u0, 0.0);
00479     fasp_darray_set(level[maxlevel], b0, 0.0);
00480     fasp_darray_set(level[maxlevel], r0, 0.0);
00481     fasp_darray_cp(nx, u, u0);
00482     fasp_darray_cp(nx, b, b0);
00483
00484     // compute initial l2 norm of residue
00485     fasp_darray_set(level[1], r0, 0.0);
00486     residualld(u0, b0, r0, 0, level);
00487     norm_r0 = l2norm(r0, level, 0);
00488     if (norm_r0 < atol) goto FINISHED;
00489
00490     // Full GMG solver
00491     fmgld(u0, b0, level, maxlevel, nx);
00492
00493     // update u
00494     fasp_darray_cp(level[1], u0, u);
00495
00496     // print out Relative Residual and CPU time if needed
00497     if ( prtlvl > PRINT_NONE ) {
00498         fasp_gettime(&AMG_end);
00499         fasp_cputime("FGMG totally", AMG_end - AMG_start);
00500         residualld(u0, b0, r0, 0, level);
00501         norm_r = l2norm(r0, level, 0);
00502         printf("Relative Residual = %e.\n", norm_r/norm_r0);
00503     }
00504
00505 FINISHED:
00506     free(level);
00507     free(r0);
00508     free(u0);
00509     free(b0);
00510
00511 #if DEBUG_MODE > 0
00512     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00513 #endif
00514
00515     return;
00516 }
00517
00536 void fasp_poisson_fmg2d (REAL      *u,
00537                          REAL      *b,
00538                          const INT  nx,
00539                          const INT  ny,
00540                          const INT  maxlevel,
00541                          const REAL  rtol,
00542                          const SHORT prtlvl)
00543 {
00544     const REAL atol = 1.0E-15;
00545     REAL      *u0, *r0, *b0;
00546     REAL      norm_r0, norm_r;
00547     INT        *nxk, *nyk, *level;
00548     int         i, k;
00549     REAL      AMG_start = 0, AMG_end;
00550
00551 #if DEBUG_MODE > 0
00552     printf("### DEBUG: [--Begin--] %s ...\n", __FUNCTION__);
00553     printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
00554 #endif
00555
00556     if ( prtlvl > PRINT_NONE ) {
00557         fasp_gettime(&AMG_start);
00558         printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
00559     }
00560
00561     // set nxk, nyk
00562     nxk = (INT *)malloc(maxlevel*sizeof(INT));
00563     nyk = (INT *)malloc(maxlevel*sizeof(INT));
00564
00565     nxk[0] = nx+1; nyk[0] = ny+1;
00566     for(k=1; k<maxlevel; k++) {
00567         nxk[k] = (int) (nxk[k-1] + 1) / 2;

```

```

00568     nyk[k] = (int) (nyk[k-1]+1)/2;
00569 }
00570
00571 // set level
00572 level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00573 level[0] = 0; level[1] = (nx+1)*(ny+1);
00574 for (i = 1; i < maxlevel; i++) {
00575     level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
00576 }
00577 level[maxlevel+1] = level[maxlevel] + 1;
00578
00579 // set u0, b0, r0
00580 u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00581 b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00582 r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00583 fasp_darray_set(level[maxlevel], u0, 0.0);
00584 fasp_darray_set(level[maxlevel], b0, 0.0);
00585 fasp_darray_set(level[maxlevel], r0, 0.0);
00586 fasp_darray_cp(level[1], u, u0);
00587 fasp_darray_cp(level[1], b, b0);
00588
00589 // compute initial l2 norm of residue
00590 fasp_darray_set(level[1], r0, 0.0);
00591 residual2d(u0, b0, r0, 0, level, nxk, nyk);
00592 norm_r0 = l2norm(r0, level, 0);
00593 if (norm_r0 < atol) goto FINISHED;
00594
00595 // FMG solver
00596 fmg2d(u0, b0, level, maxlevel, nxk, nyk);
00597
00598 // update u
00599 fasp_darray_cp(level[1], u0, u);
00600
00601 // print out Relative Residual and CPU time if needed
00602 if ( prtlvl > PRINT_NONE ) {
00603     fasp_gettime(&AMG_end);
00604     fasp_cputime("FGMG totally", AMG_end - AMG_start);
00605     residual2d(u0, b0, r0, 0, level, nxk, nyk);
00606     norm_r = l2norm(r0, level, 0);
00607     printf("Relative Residual = %e.\n", norm_r/norm_r0);
00608 }
00609
00610 FINISHED:
00611 free(level);
00612 free(nxk);
00613 free(nyk);
00614 free(r0);
00615 free(u0);
00616 free(b0);
00617
00618 #if DEBUG_MODE > 0
00619 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00620 #endif
00621
00622 return;
00623 }
00624
00644 void fasp_poisson_fgm3d (REAL      *u,
00645                          REAL      *b,
00646                          const INT  nx,
00647                          const INT  ny,
00648                          const INT  nz,
00649                          const INT  maxlevel,
00650                          const REAL  rtol,
00651                          const SHORT prtlvl)
00652 {
00653     const REAL  atol = 1.0E-15;
00654     REAL        *u0, *r0, *b0;
00655     REAL        norm_r0, norm_r;
00656     INT         *nxk, *nyk, *nz, *level;
00657     int         i, k;
00658     REAL        AMG_start = 0, AMG_end;
00659
00660 #if DEBUG_MODE > 0
00661 printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00662 printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
00663        nx, ny, nz, maxlevel);
00664 #endif
00665
00666 if ( prtlvl > PRINT_NONE ) {
00667     fasp_gettime(&AMG_start);

```

```

00668     printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
00669 }
00670 // set nxk, nyk, nzk
00671 nxk = (INT *)malloc(maxlevel*sizeof(INT));
00672 nyk = (INT *)malloc(maxlevel*sizeof(INT));
00673 nzk = (INT *)malloc(maxlevel*sizeof(INT));
00674
00675 nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
00676 for(k=1;k<maxlevel;k++){
00677     nxk[k] = (int) (nxk[k-1]+1)/2;
00678     nyk[k] = (int) (nyk[k-1]+1)/2;
00679     nzk[k] = (int) (nyk[k-1]+1)/2;
00680 }
00681
00682 // set level
00683 level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00684 level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
00685 for (i = 1; i < maxlevel; i++) {
00686     level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
00687 }
00688 level[maxlevel+1] = level[maxlevel]+1;
00689
00690 // set u0, b0, r0
00691 u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00692 b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00693 r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00694 fasp_darray_set(level[maxlevel], u0, 0.0);
00695 fasp_darray_set(level[maxlevel], b0, 0.0);
00696 fasp_darray_set(level[maxlevel], r0, 0.0);
00697 fasp_darray_cp(level[1], u, u0);
00698 fasp_darray_cp(level[1], b, b0);
00699
00700 // compute initial l2 norm of residue
00701 residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
00702 norm_r0 = l2norm(r0, level, 0);
00703 if (norm_r0 < atol) goto FINISHED;
00704
00705 // FMG
00706 fmg3d(u0, b0, level, maxlevel, nxk, nyk, nzk);
00707
00708 // update u
00709 fasp_darray_cp(level[1], u0, u);
00710
00711 if ( prtlvl > PRINT_NONE ) {
00712     fasp_gettime(&AMG_end);
00713     fasp_cputime("FGMG totally", AMG_end - AMG_start);
00714     residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
00715     norm_r = l2norm(r0, level, 0);
00716     printf("Relative Residual = %e.\n", norm_r/norm_r0);
00717 }
00718
00719 FINISHED:
00720 free(level);
00721 free(nxk);
00722 free(nyk);
00723 free(nzk);
00724 free(r0);
00725 free(u0);
00726 free(b0);
00727
00728 #if DEBUG_MODE > 0
00729 printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00730 #endif
00731
00732 return;
00733 }
00734
00754 INT fasp_poisson_gmgcgl (REAL      *u,
00755                          REAL      *b,
00756                          const INT  nx,
00757                          const INT  maxlevel,
00758                          const REAL  rtol,
00759                          const SHORT prtlvl)
00760 {
00761     const REAL atol = 1.0E-15;
00762     const INT  max_itr_num = 100;
00763
00764     REAL      *u0, *r0, *b0;
00765     REAL      norm_r0;
00766     INT        *level;
00767     int         i, iter = 0;

```

```

00768     REAL      AMG_start = 0, AMG_end;
00769
00770 #if DEBUG_MODE > 0
00771     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00772     printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
00773 #endif
00774
00775     if ( prtlvl > PRINT_NONE ) {
00776         fasp_gettime(&AMG_start);
00777         printf("Num of DOF's:  %d\n", (nx+1));
00778     }
00779     // set level
00780     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00781     level[0] = 0; level[1] = nx+1;
00782     for (i = 1; i < maxlevel; i++) {
00783         level[i+1] = level[i]+(level[i]-level[i-1]+1)/2;
00784     }
00785     level[maxlevel+1] = level[maxlevel]+1;
00786
00787     // set u0, b0
00788     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00789     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00790     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00791     fasp_darray_set(level[maxlevel], u0, 0.0);
00792     fasp_darray_set(level[maxlevel], b0, 0.0);
00793     fasp_darray_set(level[maxlevel], r0, 0.0);
00794     fasp_darray_cp(nx, u, u0);
00795     fasp_darray_cp(nx, b, b0);
00796
00797     // compute initial l2 norm of residue
00798     fasp_darray_set(level[1], r0, 0.0);
00799     residuall2(u, b, r0, 0, level);
00800     norm_r0 = l2norm(r0, level, 0);
00801     if (norm_r0 < atol) goto FINISHED;
00802
00803     // Preconditioned CG method
00804     iter = pcgld(u0, b0, level, maxlevel, nx, rtol, max_itr_num, prtlvl);
00805
00806     // Update u
00807     fasp_darray_cp(level[1], u0, u);
00808
00809     // print out CPU time if needed
00810     if ( prtlvl > PRINT_NONE ) {
00811         fasp_gettime(&AMG_end);
00812         fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
00813     }
00814
00815 FINISHED:
00816     free(level);
00817     free(r0);
00818     free(u0);
00819     free(b0);
00820
00821 #if DEBUG_MODE > 0
00822     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00823 #endif
00824
00825     return iter;
00826 }
00827
00849 INT fasp_poisson_gmgcg2d (REAL      *u,
00850                          REAL      *b,
00851                          const INT  nx,
00852                          const INT  ny,
00853                          const INT  maxlevel,
00854                          const REAL  rtol,
00855                          const SHORT prtlvl)
00856 {
00857     const REAL atol = 1.0E-15;
00858     const INT  max_itr_num = 100;
00859
00860     REAL      *u0,*r0,*b0;
00861     REAL      norm_r0;
00862     INT      *nxk, *nyk, *level;
00863     int      i, k, iter = 0;
00864     REAL      AMG_start = 0, AMG_end;
00865
00866 #if DEBUG_MODE > 0
00867     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00868     printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
00869 #endif

```

```

00870
00871     if ( prtlvl > PRINT_NONE ) {
00872         fasp_gettime(&AMG_start);
00873         printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
00874     }
00875     // set nxk, nyk
00876     nxk = (INT *)malloc(maxlevel*sizeof(INT));
00877     nyk = (INT *)malloc(maxlevel*sizeof(INT));
00878
00879     nxk[0] = nx+1; nyk[0] = ny+1;
00880     for (k=1;k<maxlevel;k++) {
00881         nxk[k] = (int) (nxk[k-1]+1)/2;
00882         nyk[k] = (int) (nyk[k-1]+1)/2;
00883     }
00884
00885     // set level
00886     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
00887     level[0] = 0; level[1] = (nx+1)*(ny+1);
00888     for (i = 1; i < maxlevel; i++) {
00889         level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
00890     }
00891     level[maxlevel+1] = level[maxlevel]+1;
00892
00893     // set u0, b0, r0
00894     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00895     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00896     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
00897     fasp_darray_set(level[maxlevel], u0, 0.0);
00898     fasp_darray_set(level[maxlevel], b0, 0.0);
00899     fasp_darray_set(level[maxlevel], r0, 0.0);
00900     fasp_darray_cp(level[1], u, u0);
00901     fasp_darray_cp(level[1], b, b0);
00902
00903     // compute initial l2 norm of residue
00904     fasp_darray_set(level[1], r0, 0.0);
00905     residual2d(u0, b0, r0, 0, level, nxk, nyk);
00906     norm_r0 = l2norm(r0, level, 0);
00907     if (norm_r0 < atol) goto FINISHED;
00908
00909     // Preconditioned CG method
00910     iter = pcg2d(u0, b0, level, maxlevel, nxk,
00911                 nyk, rtol, max_itr_num, prtlvl);
00912
00913     // update u
00914     fasp_darray_cp(level[1], u0, u);
00915
00916     // print out CPU time if needed
00917     if ( prtlvl > PRINT_NONE ) {
00918         fasp_gettime(&AMG_end);
00919         fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
00920     }
00921
00922 FINISHED:
00923     free(level);
00924     free(nxk);
00925     free(nyk);
00926     free(r0);
00927     free(u0);
00928     free(b0);
00929
00930 #if DEBUG_MODE > 0
00931     printf("### DEBUG: [--End--] %s ... \n", __FUNCTION__);
00932 #endif
00933
00934     return iter;
00935 }
00936
00959 INT fasp_poisson_gmgcg3d (REAL          *u,
00960                          REAL          *b,
00961                          const INT     nx,
00962                          const INT     ny,
00963                          const INT     nz,
00964                          const INT     maxlevel,
00965                          const REAL    rtol,
00966                          const SHORT   prtlvl)
00967 {
00968     const REAL atol = 1.0E-15;
00969     const INT  max_itr_num = 100;
00970
00971     REAL      *u0,*r0,*b0;
00972     REAL      norm_r0;

```

```

00973     INT          *nxk, *nyk, *nzk, *level;
00974     int          i, k, iter = 0;
00975     REAL         AMG_start = 0, AMG_end;
00976
00977 #if DEBUG_MODE > 0
00978     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00979     printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
00980           nx, ny, nz, maxlevel);
00981 #endif
00982
00983     if ( prtlvl > PRINT_NONE ) {
00984         fasp_gettime(&AMG_start);
00985         printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
00986     }
00987
00988     // set nxk, nyk, nzk
00989     nxk = (INT *)malloc(maxlevel*sizeof(INT));
00990     nyk = (INT *)malloc(maxlevel*sizeof(INT));
00991     nzk = (INT *)malloc(maxlevel*sizeof(INT));
00992
00993     nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
00994     for (k = 1; k < maxlevel; k++) {
00995         nxk[k] = (int) (nxk[k-1]+1)/2;
00996         nyk[k] = (int) (nyk[k-1]+1)/2;
00997         nzk[k] = (int) (nyk[k-1]+1)/2;
00998     }
00999
01000     // set level
01001     level = (INT *)malloc((maxlevel+2)*sizeof(INT));
01002     level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
01003     for (i = 1; i < maxlevel; i++) {
01004         level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
01005     }
01006     level[maxlevel+1] = level[maxlevel]+1;
01007
01008     // set u0, b0
01009     u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
01010     b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
01011     r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
01012     fasp_darray_set(level[maxlevel], u0, 0.0);
01013     fasp_darray_set(level[maxlevel], b0, 0.0);
01014     fasp_darray_set(level[maxlevel], r0, 0.0);
01015     fasp_darray_cp(level[1], u, u0);
01016     fasp_darray_cp(level[1], b, b0);
01017
01018     // compute initial l2 norm of residue
01019     residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
01020     norm_r0 = l2norm(r0, level, 0);
01021     if (norm_r0 < atol) goto FINISHED;
01022
01023     // Preconditioned CG method
01024     iter = pcg3d(u0, b0, level, maxlevel, nxk,
01025               nyk, nzk, rtol, max_itr_num, prtlvl);
01026
01027     // update u
01028     fasp_darray_cp(level[1], u0, u);
01029
01030     // print out CPU time if needed
01031     if ( prtlvl > PRINT_NONE ) {
01032         fasp_gettime(&AMG_end);
01033         fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
01034     }
01035
01036 FINISHED:
01037     free(level);
01038     free(nxk);
01039     free(nyk);
01040     free(nzk);
01041     free(r0);
01042     free(u0);
01043     free(b0);
01044
01045 #if DEBUG_MODE > 0
01046     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
01047 #endif
01048
01049     return iter;
01050 }
01051
01052 /*-----*/
01053 /*--      End of File      --*/

```

01054 / *-----*/

9.187 SolMatFree.c File Reference

Iterative solvers using MatFree spmv operations.

```
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "fasp_block.h"
#include "KryUtil.inl"
#include "BlaSpmvMatFree.inl"
```

Functions

- [INT fasp_solver_itsolver](#) ([mxv_matfree](#) *mf, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax=b$ by preconditioned Krylov methods for CSR matrices.
- [INT fasp_solver_krylov](#) ([mxv_matfree](#) *mf, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods – without preconditioner.
- void [fasp_solver_matfree_init](#) ([INT](#) matrix_format, [mxv_matfree](#) *mf, void *A)
Initialize MatFree (or non-specified format) itsolvers.

9.187.1 Detailed Description

Iterative solvers using MatFree spmv operations.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxMessage.c](#), [AuxTiming.c](#), [BlaSpmvBLC.c](#), [BlaSpmvBSR.c](#), [BlaSpmvCSR.c](#), [BlaSpmvCSRL.c](#), [BlaSpmvSTR.c](#), [KryPbcgs.c](#), [KryPcg.c](#), [KryPgcg.c](#), [KryPgmres.c](#), [KryPminres.c](#), [KryPvfgmres.c](#), and [KryPvgmres.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolMatFree.c](#).

9.187.2 Function Documentation

9.187.2.1 fasp_solver_itsolver()

```
INT fasp_solver_itsolver (
    mxv_matfree * mf,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax=b$ by preconditioned Krylov methods for CSR matrices.

Note

This is an abstract interface for iterative methods.

Parameters

<i>mf</i>	Pointer to mxv_matfree MatFree spmv operation
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

09/25/2009

Modified by Feiteng Huang on 09/19/2012: matrix free
Definition at line [58](#) of file [SolMatFree.c](#).

9.187.2.2 fasp_solver_krylov()

```
INT fasp_solver_krylov (
    mxv_matfree * mf,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods – without preconditioner.

Parameters

<i>mf</i>	Pointer to mxv_matfree MatFree spmv operation
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Number of iterations if succeed

Author

Chensong Zhang, Shiquan Zhang

Date

09/25/2009

Modified by Feiteng Huang on 09/20/2012: matrix free
Definition at line [154](#) of file [SolMatFree.c](#).

9.187.2.3 fasp_solver_matfree_init()

```
void fasp_solver_matfree_init (
    INT matrix_format,
    mxv_matfree * mf,
    void * A )
```

Initialize MatFree (or non-specified format) itsolvers.

Parameters

<i>matrix_format</i>	matrix format
<i>mf</i>	Pointer to mxv_matfree MatFree spmv operation
<i>A</i>	void pointer to the coefficient matrix

Author

Feiteng Huang

Date

09/18/2012

Modified by Chensong Zhang on 05/10/2013: Change interface of mat-free mv
Definition at line 201 of file [SolMatFree.c](#).

9.188 SolMatFree.c

[Go to the documentation of this file.](#)

```
00001
00016 #include <time.h>
00017
00018 #ifdef _OPENMP
00019 #include <omp.h>
00020 #endif
00021
00022 #include "fasp.h"
00023 #include "fasp_funcs.h"
00024 #include "fasp_block.h"
00025
00026 /*-----*/
00027 /*--  Declare Private Functions  --*/
00028 /*-----*/
00029
00030 #include "KryUtil.inl"
00031 #include "BlaSpmvMatFree.inl"
00032
00033 /*-----*/
00034 /*--      Public Functions      --*/
00035 /*-----*/
00036
00058 INT fasp_solver_itsolver (mxv_matfree *mf,
00059                          dvector *b,
00060                          dvector *x,
00061                          precondition *pc,
00062                          ITS_param *itparam)
00063 {
00064     const SHORT prtlvl = itparam->print_level;
00065     const SHORT itsolver_type = itparam->itsolver_type;
00066     const SHORT stop_type = itparam->stop_type;
00067     const INT restart = itparam->restart;
00068     const INT MaxIt = itparam->maxit;
00069     const REAL tol = itparam->tol;
00070
00071     /* Local Variables */
00072     REAL solve_start, solve_end, solve_time;
00073     INT iter = ERROR_SOLVER_TYPE;
00074
```

```

00075     fasp_gettime(&solve_start);
00076
00077 #if DEBUG_MODE > 0
00078     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00079     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00080 #endif
00081
00082     /* Safe-guard checks on parameters */
00083     ITS_CHECK ( MaxIt, tol );
00084
00085     /* Choose a desirable Krylov iterative solver */
00086     switch ( itsolver_type ) {
00087
00088     case SOLVER_CG:
00089         iter = fasp_solver_pcg(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00090         break;
00091
00092     case SOLVER_BiCGstab:
00093         iter = fasp_solver_pbcgs(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00094         break;
00095
00096     case SOLVER_MinRes:
00097         iter = fasp_solver_pminres(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00098         break;
00099
00100     case SOLVER_GMRES:
00101         iter = fasp_solver_pgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00102         break;
00103
00104     case SOLVER_VGMRES:
00105         iter = fasp_solver_pvgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00106         break;
00107
00108     case SOLVER_VFGMRES:
00109         iter = fasp_solver_pvfgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00110         break;
00111
00112     case SOLVER_GCG:
00113         iter = fasp_solver_pgcg(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00114         break;
00115
00116     default:
00117         printf("### ERROR: Unknown iterative solver type %d! [%s]\n",
00118             itsolver_type, __FUNCTION__);
00119         return ERROR_SOLVER_TYPE;
00120     }
00121 }
00122
00123 if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
00124     fasp_gettime(&solve_end);
00125     solve_time = solve_end - solve_start;
00126     fasp_cputime("Iterative method", solve_time);
00127 }
00128
00129 #if DEBUG_MODE > 0
00130     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00131 #endif
00132
00133     return iter;
00134 }
00135
00154 INT fasp_solver_krylov (mxv_matfree *mf,
00155     dvector *b,
00156     dvector *x,
00157     ITS_param *itparam)
00158 {
00159     const SHORT prtlvl = itparam->print_level;
00160
00161     /* Local Variables */
00162     INT status = FASP_SUCCESS;
00163     REAL solve_start, solve_end, solve_time;
00164
00165 #if DEBUG_MODE > 0
00166     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00167     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00168 #endif
00169
00170     fasp_gettime(&solve_start);
00171
00172     status = fasp_solver_itsolver(mf,b,x,NULL,itparam);
00173

```

```

00174     if ( prtlvl >= PRINT_MIN ) {
00175         fasp_gettime(&solve_end);
00176         solve_time = solve_end - solve_start;
00177         fasp_cputime("Krylov method totally", solve_time);
00178     }
00179
00180 #if DEBUG_MODE > 0
00181     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00182 #endif
00183
00184     return status;
00185 }
00186
00201 void fasp_solver_matfree_init (INT          matrix_format,
00202                               mxv_matfree *mf,
00203                               void         *A)
00204 {
00205     switch ( matrix_format ) {
00206
00207     case MAT_CSR:
00208         mf->fct = fasp_blas_m xv_csr;
00209         break;
00210
00211     case MAT_BSR:
00212         mf->fct = fasp_blas_m xv_bsr;
00213         break;
00214
00215     case MAT_STR:
00216         mf->fct = fasp_blas_m xv_str;
00217         break;
00218
00219     case MAT_BLC:
00220         mf->fct = fasp_blas_m xv_blc;
00221         break;
00222
00223     case MAT_CSRL:
00224         mf->fct = fasp_blas_m xv_csrl;
00225         break;
00226
00227     default:
00228         printf("### ERROR: Unknown matrix format %d!\n", matrix_format);
00229         exit(ERROR_DATA_STRUCTURE);
00230     }
00231 }
00232
00233 mf->data = A;
00234 }
00235
00236 /*-----*/
00237 /*--      End of File      --*/
00238 /*-----*/

```

9.189 SolSTR.c File Reference

Iterative solvers for [dSTRmat](#) matrices.

```

#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"

```

Functions

- [INT fasp_solver_dstr_itsolver](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [precond](#) *pc, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods.
- [INT fasp_solver_dstr_krylov](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)
Solve $Ax=b$ by standard Krylov methods.
- [INT fasp_solver_dstr_krylov_diag](#) ([dSTRmat](#) *A, [dvector](#) *b, [dvector](#) *x, [ITS_param](#) *itparam)

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

- `INT fasp_solver_dstr_krylov_ilu` (`dSTRmat *A`, `dvector *b`, `dvector *x`, `ITS_param *itparam`, `ILU_param *iluparam`)

Solve $Ax=b$ by structured ILU preconditioned Krylov methods.

- `INT fasp_solver_dstr_krylov_blockgs` (`dSTRmat *A`, `dvector *b`, `dvector *x`, `ITS_param *itparam`, `ivector *neigh`, `ivector *order`)

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

9.189.1 Detailed Description

Iterative solvers for `dSTRmat` matrices.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxArray.c](#), [AuxMemory.c](#), [AuxMessage.c](#), [AuxTiming.c](#), [AuxVector.c](#), [BlaSmallMatInv.c](#), [BlaLUSetupSTR.c](#), [BlaSparseSTR.c](#), [ItrSmootherSTR.c](#), [KryPbcgs.c](#), [KryPcg.c](#), [KryPgmres.c](#), [KryPvgmres.c](#), and [PreSTR.c](#)

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolSTR.c](#).

9.189.2 Function Documentation

9.189.2.1 `fasp_solver_dstr_itsolver()`

```
INT fasp_solver_dstr_itsolver (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    precondition * pc,
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in <code>dSTRmat</code> format
<i>b</i>	Pointer to the right hand side in <code>dvector</code> format
<i>x</i>	Pointer to the approx solution in <code>dvector</code> format
<i>pc</i>	Pointer to the preconditioning action
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Chensong Zhang

Date

09/25/2009

Modified by Chunsheng Feng on 03/04/2016: add VBiCGstab solver
Definition at line 51 of file [SolSTR.c](#).

9.189.2.2 fasp_solver_dstr_krylov()

```
INT fasp_solver_dstr_krylov (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax=b$ by standard Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dSTRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

04/25/2010

Definition at line 131 of file [SolSTR.c](#).

9.189.2.3 fasp_solver_dstr_krylov_blockgs()

```
INT fasp_solver_dstr_krylov_blockgs (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    ivector * neigh,
    ivector * order )
```

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dSTRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format

Parameters

<i>itparam</i>	Pointer to parameters for iterative solvers
<i>neigh</i>	Pointer to neighbor vector
<i>order</i>	Pointer to solver ordering

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

10/10/2010

Definition at line 334 of file [SolSTR.c](#).

9.189.2.4 fasp_solver_dstr_krylov_diag()

```
INT fasp_solver_dstr_krylov_diag (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam )
```

Solve $Ax=b$ by diagonal preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dSTRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers

Returns

Iteration number if converges; ERROR otherwise.

Author

Zhiyang Zhou

Date

4/23/2010

Definition at line 177 of file [SolSTR.c](#).

9.189.2.5 fasp_solver_dstr_krylov_ilu()

```

INT fasp_solver_dstr_krylov_ilu (
    dSTRmat * A,
    dvector * b,
    dvector * x,
    ITS_param * itparam,
    ILU_param * iluparam )

```

Solve $Ax=b$ by structured ILU preconditioned Krylov methods.

Parameters

<i>A</i>	Pointer to the coeff matrix in dSTRmat format
<i>b</i>	Pointer to the right hand side in dvector format
<i>x</i>	Pointer to the approx solution in dvector format
<i>itparam</i>	Pointer to parameters for iterative solvers
<i>iluparam</i>	Pointer to parameters for ILU

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

05/01/2010

Definition at line [241](#) of file [SolSTR.c](#).

9.190 SolSTR.c

[Go to the documentation of this file.](#)

```

00001
00016 #include <math.h>
00017 #include <time.h>
00018
00019 #include "fasp.h"
00020 #include "fasp_funcs.h"
00021
00022 /*-----*/
00023 /*--  Declare Private Functions  --*/
00024 /*-----*/
00025
00026 #include "KryUtil.inl"
00027
00028 /*-----*/
00029 /*--      Public Functions      --*/
00030 /*-----*/
00031
00051 INT fasp_solver_dstr_itsolver (dSTRmat    *A,
00052                               dvector    *b,
00053                               dvector    *x,
00054                               precondition *pc,
00055                               ITS_param  *itparam)
00056 {
00057     const SHORT prtlvl = itparam->print_level;
00058     const SHORT itsolver_type = itparam->itsolver_type;
00059     const SHORT stop_type = itparam->stop_type;
00060     const SHORT restart = itparam->restart;
00061     const INT   MaxIt = itparam->maxit;

```

```

00062     const REAL  tol = itparam->tol;
00063
00064     // local variables
00065     INT iter = ERROR_SOLVER_TYPE;
00066     REAL solve_start, solve_end;
00067
00068     #if DEBUG_MODE > 0
00069     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00070     printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
00071     #endif
00072
00073     fasp_gettime(&solve_start);
00074
00075     /* Safe-guard checks on parameters */
00076     ITS_CHECK ( MaxIt, tol );
00077
00078     switch (itsolver_type) {
00079
00080         case SOLVER_CG:
00081             iter=fasp_solver_dstr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00082             break;
00083
00084         case SOLVER_BiCGstab:
00085             iter=fasp_solver_dstr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
00086             break;
00087
00088         case SOLVER_GMRES:
00089             iter=fasp_solver_dstr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00090             break;
00091
00092         case SOLVER_VGMRES:
00093             iter=fasp_solver_dstr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
00094             break;
00095
00096         default:
00097             printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
00098                 itsolver_type, __FUNCTION__);
00099             return ERROR_SOLVER_TYPE;
00100
00101     }
00102
00103     if ( (prtlvl > PRINT_MIN) && (iter >= 0) ) {
00104         fasp_gettime(&solve_end);
00105         fasp_cputime("Iterative method", solve_end - solve_start);
00106     }
00107
00108     #if DEBUG_MODE > 0
00109     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00110     #endif
00111
00112     return iter;
00113 }
00114
00131 INT fasp_solver_dstr_krylov (dSTRmat    *A,
00132                             dvector     *b,
00133                             dvector     *x,
00134                             ITS_param   *itparam)
00135 {
00136     const SHORT prtlvl = itparam->print_level;
00137     INT status = FASP_SUCCESS;
00138     REAL solve_start, solve_end;
00139
00140     #if DEBUG_MODE > 0
00141     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00142     #endif
00143
00144     // solver part
00145     fasp_gettime(&solve_start);
00146
00147     status=fasp_solver_dstr_itsolver(A,b,x,NULL,itparam);
00148
00149     fasp_gettime(&solve_end);
00150
00151     if ( prtlvl >= PRINT_MIN )
00152         fasp_cputime("Krylov method totally", solve_end - solve_start);
00153
00154     #if DEBUG_MODE > 0
00155     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00156     #endif
00157
00158     return status;

```



```

00159 }
00160
00177 INT fasp_solver_dstr_krylov_diag (dSTRmat    *A,
00178                                dvector     *b,
00179                                dvector     *x,
00180                                ITS_param   *itparam)
00181 {
00182     const SHORT prtlvl = itparam->print_level;
00183     const INT ngrid = A->ngrid;
00184
00185     INT status = FASP_SUCCESS;
00186     REAL solve_start, solve_end;
00187     INT nc = A->nc, nc2 = nc*nc, i;
00188
00189     fasp_gettime(&solve_start);
00190
00191     // setup preconditioner
00192     precond_diag_str diag;
00193     fasp_dvec_alloc(ngrid*nc2, &(diag.diag));
00194     fasp_darray_cp(ngrid*nc2, A->diag, diag.diag.val);
00195
00196     diag.nc = nc;
00197
00198     for (i=0;i<ngrid;++i) fasp_smat_inv(&(diag.diag.val[i*nc2]),nc);
00199
00200     precond *pc = (precond *)fasp_mem_calloc(1,sizeof(precond));
00201
00202     pc->data = &diag;
00203     pc->fct = fasp_precond_dstr_diag;
00204
00205     #if DEBUG_MODE > 0
00206     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00207     #endif
00208
00209     // solver part
00210     status=fasp_solver_dstr_itsolver(A,b,x,pc,itparam);
00211
00212     fasp_gettime(&solve_end);
00213
00214     if ( prtlvl >= PRINT_MIN )
00215         fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);
00216
00217     #if DEBUG_MODE > 0
00218     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00219     #endif
00220
00221     return status;
00222 }
00223
00241 INT fasp_solver_dstr_krylov_ilu (dSTRmat    *A,
00242                                dvector     *b,
00243                                dvector     *x,
00244                                ITS_param   *itparam,
00245                                ILU_param   *iluparam)
00246 {
00247     const SHORT prtlvl = itparam->print_level;
00248     const INT ILU_lfil = iluparam->ILU_lfil;
00249
00250     INT status = FASP_SUCCESS;
00251     REAL setup_start, setup_end, setup_time;
00252     REAL solve_start, solve_end, solve_time;
00253
00254     //set up
00255     dSTRmat LU;
00256
00257     #if DEBUG_MODE > 0
00258     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00259     #endif
00260
00261     fasp_gettime(&setup_start);
00262
00263     if (ILU_lfil == 0) {
00264         fasp_ilu_dstr_setup0(A,&LU);
00265     }
00266     else if (ILU_lfil == 1) {
00267         fasp_ilu_dstr_setup1(A,&LU);
00268     }
00269     else {
00270         printf("### ERROR: Illegal level of ILU fill-in (>1)! [%s]\n", __FUNCTION__);
00271         return ERROR_MISC;
00272     }

```

```

00273
00274     fasp_gettime(&setup_end);
00275
00276     setup_time = setup_end - setup_start;
00277
00278     if ( prtlvl > PRINT_NONE )
00279         printf("Structrued ILU(%d) setup costs %f seconds.\n", ILU_lfil, setup_time);
00280
00281     precondition pc; pc.data=&LU;
00282     if (ILU_lfil == 0) {
00283         pc.fct = fasp_precond_dstr_ilu0;
00284     }
00285     else if (ILU_lfil == 1) {
00286         pc.fct = fasp_precond_dstr_ilu1;
00287     }
00288     else {
00289         printf("### ERROR: Illegal level of ILU fill-in (>1)! [%s]\n", __FUNCTION__);
00290         return ERROR_MISC;
00291     }
00292
00293     // solver part
00294     fasp_gettime(&solve_start);
00295
00296     status=fasp_solver_dstr_itsolver(A,b,x,&pc,itparam);
00297
00298     fasp_gettime(&solve_end);
00299
00300     if ( prtlvl >= PRINT_MIN ) {
00301         solve_time = solve_end - solve_start;
00302         printf("Iterative solver costs %f seconds.\n", solve_time);
00303         fasp_cputime("ILU-Krylov method totally", setup_time+solve_time);
00304     }
00305
00306     fasp_dstr_free(&LU);
00307
00308     #if DEBUG_MODE > 0
00309     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00310     #endif
00311
00312     return status;
00313 }
00314
00334 INT fasp_solver_dstr_krylov_blockgs (dSTRmat      *A,
00335                                     dvector      *b,
00336                                     dvector      *x,
00337                                     ITS_param    *itparam,
00338                                     ivector      *neigh,
00339                                     ivector      *order)
00340 {
00341     // Parameter for iterative method
00342     const SHORT prtlvl = itparam->print_level;
00343
00344     // Information about matrices
00345     INT ngrid = A->ngrid;
00346
00347     // return parameter
00348     INT status = FASP_SUCCESS;
00349
00350     // local parameter
00351     REAL setup_start, setup_end, setup_time = 0;
00352     REAL solve_start, solve_end, solve_time = 0;
00353
00354     dvector *diaginv;
00355     ivector *pivot;
00356
00357     #if DEBUG_MODE > 0
00358     printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
00359     #endif
00360
00361     // setup preconditioner
00362     fasp_gettime(&setup_start);
00363
00364     diaginv = (dvector *)fasp_mem_calloc(ngrid, sizeof(dvector));
00365     pivot = (ivector *)fasp_mem_calloc(ngrid, sizeof(ivector));
00366     fasp_generate_diaginv_block(A, neigh, diaginv, pivot);
00367
00368     precondition_data_str pcddata;
00369     pcddata.A_str = A;
00370     pcddata.diaginv = diaginv;
00371     pcddata.pivot = pivot;
00372     pcddata.order = order;

```

```

00373     pcddata.neigh = neigh;
00374
00375     precondition pc; pc.data = &pcdata; pc.fct = fasp_precond_dstr_blockgs;
00376
00377     fasp_gettime(&setup_end);
00378
00379     if ( prtlvl > PRINT_NONE ) {
00380         setup_time = setup_end - setup_start;
00381         printf("Preconditioner setup costs %f seconds.\n", setup_time);
00382     }
00383
00384     // solver part
00385     fasp_gettime(&solve_start);
00386
00387     status = fasp_solver_dstr_itsolver(A,b,x,&pc,itparam);
00388
00389     fasp_gettime(&solve_end);
00390     solve_time = solve_end - solve_start;
00391
00392     if ( prtlvl >= PRINT_MIN ) {
00393         fasp_cputime("Iterative solver", solve_time);
00394         fasp_cputime("BlockGS-Krylov method totally", setup_time + solve_time);
00395     }
00396
00397 #if DEBUG_MODE > 0
00398     printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
00399 #endif
00400
00401     return status;
00402 }
00403
00404 /*-----*/
00405 /*--          End of File          --*/
00406 /*-----*/

```

9.191 SolWrapper.c File Reference

Wrappers for accessing functions by advanced users.

```

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_funcs.h"

```

Functions

- void `fasp_fwrapper_dcsr_pardiso_` (INT *n, INT *nnz, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, INT *ptrlvl)
Solve $Ax=b$ by the Pardiso direct solver.
- void `fasp_fwrapper_dcsr_amg_` (INT *n, INT *nnz, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, REAL *tol, INT *maxit, INT *ptrlvl)
Solve $Ax=b$ by Ruge and Stuben's classic AMG.
- void `fasp_fwrapper_dcsr_krylov_ilu_` (INT *n, INT *nnz, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, REAL *tol, INT *maxit, INT *ptrlvl)
Solve $Ax=b$ by Krylov method preconditioned by ILUk.
- void `fasp_fwrapper_dcsr_krylov_amg_` (INT *n, INT *nnz, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, REAL *tol, INT *maxit, INT *ptrlvl)
Solve $Ax=b$ by Krylov method preconditioned by classic AMG.
- void `fasp_fwrapper_dbsr_krylov_ilu_` (INT *n, INT *nnz, INT *nb, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, REAL *tol, INT *maxit, INT *ptrlvl)
Solve $Ax=b$ by Krylov method preconditioned by block ILU in BSR format.
- void `fasp_fwrapper_dbsr_krylov_amg_` (INT *n, INT *nnz, INT *nb, INT *ia, INT *ja, REAL *a, REAL *b, REAL *u, REAL *tol, INT *maxit, INT *ptrlvl)
Solve $Ax=b$ by Krylov method preconditioned by block AMG in BSR format.

9.191.1 Detailed Description

Wrappers for accessing functions by advanced users.

Note

This file contains Level-5 (Sol) functions. It requires: [AuxParam.c](#), [BlaFormat.c](#), [BlaSparseBSR.c](#), [BlaSparseCSR.c](#), [SolAMG.c](#), [SolBSR.c](#), and [SolCSR.c](#)

IMPORTANT: The wrappers DO NOT change the original matrix data. Users should shift the matrix indices in order to make the IA and JA to start from 0 instead of 1.

Copyright (C) 2009–Present by the FASP team. All rights reserved.

Released under the terms of the GNU Lesser General Public License 3.0 or later.

Definition in file [SolWrapper.c](#).

9.191.2 Function Documentation

9.191.2.1 fasp_fwrapper_dbsr_krylov_amg_()

```
void fasp_fwrapper_dbsr_krylov_amg_ (
    INT * n,
    INT * nnz,
    INT * nb,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    REAL * u,
    REAL * tol,
    INT * maxit,
    INT * ptrlvl )
```

Solve $Ax=b$ by Krylov method preconditioned by block AMG in BSR format.

Parameters

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A
<i>nb</i>	Size of each small block
<i>ia</i>	IA of A in CSR format
<i>ja</i>	JA of A in CSR format
<i>a</i>	VAL of A in CSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>tol</i>	Tolerance for iterative solvers
<i>maxit</i>	Max number of iterations
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

04/05/2018

Definition at line 321 of file [SolWrapper.c](#).**9.191.2.2 fasp_fwrapper_dbsr_krylov_ilu_()**

```
void fasp_fwrapper_dbsr_krylov_ilu_ (
    INT * n,
    INT * nnz,
    INT * nb,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    REAL * u,
    REAL * tol,
    INT * maxit,
    INT * ptrlvl )
```

Solve $Ax=b$ by Krylov method preconditioned by block ILU in BSR format.

Parameters

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A
<i>nb</i>	Size of each small block
<i>ia</i>	IA of A in BSR format
<i>ja</i>	JA of A in BSR format
<i>a</i>	VAL of A in BSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>tol</i>	Tolerance for iterative solvers
<i>maxit</i>	Max number of iterations
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

03/25/2018

Definition at line 258 of file [SolWrapper.c](#).**9.191.2.3 fasp_fwrapper_dcsr_amg_()**

```
void fasp_fwrapper_dcsr_amg_ (
    INT * n,
```

```

    INT * nnz,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    REAL * u,
    REAL * tol,
    INT * maxit,
    INT * ptrlvl )

```

Solve $Ax=b$ by Ruge and Stuben's classic AMG.

Parameters

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A
<i>ia</i>	IA of A in CSR format
<i>ja</i>	JA of A in CSR format
<i>a</i>	VAL of A in CSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>tol</i>	Tolerance for iterative solvers
<i>maxit</i>	Max number of iterations
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

09/16/2010

Definition at line 90 of file [SolWrapper.c](#).

9.191.2.4 fasp_fwrapper_dcsr_krylov_amg_()

```

void fasp_fwrapper_dcsr_krylov_amg_ (
    INT * n,
    INT * nnz,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    REAL * u,
    REAL * tol,
    INT * maxit,
    INT * ptrlvl )

```

Solve $Ax=b$ by Krylov method preconditioned by classic AMG.

Parameters

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A

Parameters

<i>ia</i>	IA of A in CSR format
<i>ja</i>	JA of A in CSR format
<i>a</i>	VAL of A in CSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>tol</i>	Tolerance for iterative solvers
<i>maxit</i>	Max number of iterations
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

09/16/2010

Step 0. Read input parameters

Definition at line 200 of file [SolWrapper.c](#).

9.191.2.5 fasp_fwrapper_dcsr_krylov_ilu_()

```
void fasp_fwrapper_dcsr_krylov_ilu_ (  
    INT * n,  
    INT * nnz,  
    INT * ia,  
    INT * ja,  
    REAL * a,  
    REAL * b,  
    REAL * u,  
    REAL * tol,  
    INT * maxit,  
    INT * ptrlvl )
```

Solve $Ax=b$ by Krylov method preconditioned by ILUk.

Parameters

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A
<i>ia</i>	IA of A in CSR format
<i>ja</i>	JA of A in CSR format
<i>a</i>	VAL of A in CSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>tol</i>	Tolerance for iterative solvers
<i>maxit</i>	Max number of iterations
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

03/24/2018

Definition at line 141 of file [SolWrapper.c](#).**9.191.2.6 fasp_fwrapper_dcsr_pardiso_()**

```
void fasp_fwrapper_dcsr_pardiso_ (
    INT * n,
    INT * nnz,
    INT * ia,
    INT * ja,
    REAL * a,
    REAL * b,
    REAL * u,
    INT * ptrlvl )
```

Solve $Ax=b$ by the Pardiso direct solver.**Parameters**

<i>n</i>	Number of cols of A
<i>nnz</i>	Number of nonzeros of A
<i>ia</i>	IA of A in CSR format
<i>ja</i>	JA of A in CSR format
<i>a</i>	VAL of A in CSR format
<i>b</i>	RHS vector
<i>u</i>	Solution vector
<i>ptrlvl</i>	Print level for iterative solvers

Author

Chensong Zhang

Date

01/09/2020

Definition at line 45 of file [SolWrapper.c](#).**9.192 SolWrapper.c**[Go to the documentation of this file.](#)

```
00001
00019 #include "fasp.h"
00020 #include "fasp_block.h"
00021 #include "fasp_funcs.h"
00022
00023 /*-----*/
00024 /*--      Public Functions      --*/
00025 /*-----*/
00026
```



```

00045 void fasp_fwrapper_dcsr_pardiso_(INT* n, INT* nnz, INT* ia, INT* ja, REAL* a, REAL* b,
00046                                     REAL* u, INT* ptrlvl)
00047 {
00048     dCSRmat mat;      // coefficient matrix
00049     dvector rhs, sol; // right-hand-side, solution
00050
00051     // set up coefficient matrix
00052     mat.row = *n;
00053     mat.col = *n;
00054     mat.nnz = *nnz;
00055     mat.IA = ia;
00056     mat.JA = ja;
00057     mat.val = a;
00058
00059     rhs.row = *n;
00060     rhs.val = b;
00061     sol.row = *n;
00062     sol.val = u;
00063
00064     fasp_dcsr_sort(&mat);
00065
00066     fasp_solver_pardiso(&mat, &rhs, &sol, *ptrlvl);
00067 }
00068
00090 void fasp_fwrapper_dcsr_amg_(INT* n, INT* nnz, INT* ia, INT* ja, REAL* a, REAL* b,
00091                             REAL* u, REAL* tol, INT* maxit, INT* ptrlvl)
00092 {
00093     dCSRmat mat;      // coefficient matrix
00094     dvector rhs, sol; // right-hand-side, solution
00095     AMG_param amgparam; // parameters for AMG
00096
00097     // setup AMG parameters
00098     fasp_param_amg_init(&amgparam);
00099
00100     amgparam.tol = *tol;
00101     amgparam.print_level = *ptrlvl;
00102     amgparam.maxit = *maxit;
00103
00104     // set up coefficient matrix
00105     mat.row = *n;
00106     mat.col = *n;
00107     mat.nnz = *nnz;
00108     mat.IA = ia;
00109     mat.JA = ja;
00110     mat.val = a;
00111
00112     rhs.row = *n;
00113     rhs.val = b;
00114     sol.row = *n;
00115     sol.val = u;
00116
00117     fasp_solver_amg(&mat, &rhs, &sol, &amgparam);
00118 }
00119
00141 void fasp_fwrapper_dcsr_krylov_ilu_(INT* n, INT* nnz, INT* ia, INT* ja, REAL* a,
00142                                    REAL* b, REAL* u, REAL* tol, INT* maxit,
00143                                    INT* ptrlvl)
00144 {
00145     dCSRmat mat;      // coefficient matrix
00146     dvector rhs, sol; // right-hand-side, solution
00147     ILU_param iluparam; // parameters for ILU
00148     ITS_param itsparam; // parameters for itsolver
00149
00150     // setup ILU parameters
00151     fasp_param_ilu_init(&iluparam);
00152
00153     iluparam.print_level = *ptrlvl;
00154
00155     // setup Krylov method parameters
00156     fasp_param_solver_init(&itsparam);
00157
00158     itsparam.itsolver_type = SOLVER_VFGMRES;
00159     itsparam.tol = *tol;
00160     itsparam.maxit = *maxit;
00161     itsparam.print_level = *ptrlvl;
00162
00163     // set up coefficient matrix
00164     mat.row = *n;
00165     mat.col = *n;
00166     mat.nnz = *nnz;
00167     mat.IA = ia;

```

```

00168     mat.JA = ja;
00169     mat.val = a;
00170
00171     rhs.row = *n;
00172     rhs.val = b;
00173     sol.row = *n;
00174     sol.val = u;
00175
00176     fasp_solver_dcsr_krylov_ilu(&mat, &rhs, &sol, &itsparam, &iluparam);
00177 }
00178
00200 void fasp_fwapper_dcsr_krylov_amg_(INT* n, INT* nnz, INT* ia, INT* ja, REAL* a,
00201     REAL* b, REAL* u, REAL* tol, INT* maxit,
00202     INT* ptrlvl)
00203 {
00204     dCSRmat mat; // coefficient matrix
00205     dvector rhs, sol; // right-hand-side, solution
00206     input_param inparam; // parameters from input files
00207     AMG_param amgparam; // parameters for AMG
00208     ITS_param itsparam; // parameters for itsolver
00209     ILU_param iluparam; // parameters for ILU
00210
00211     char* inputfile = "ini/amg.dat"; // Added for fasp4ns 2022.04.08 --zcs
00212     fasp_param_input(inputfile, &inparam);
00213     fasp_param_init(&inparam, &itsparam, &amgparam, &iluparam, NULL);
00214
00215     itsparam.tol = *tol;
00216     itsparam.maxit = *maxit;
00217     itsparam.print_level = *ptrlvl;
00218
00219     // set up coefficient matrix
00220     mat.row = *n;
00221     mat.col = *n;
00222     mat.nnz = *nnz;
00223     mat.IA = ia;
00224     mat.JA = ja;
00225     mat.val = a;
00226
00227     rhs.row = *n;
00228     rhs.val = b;
00229     sol.row = *n;
00230     sol.val = u;
00231
00232     fasp_solver_dcsr_krylov_amg(&mat, &rhs, &sol, &itsparam, &amgparam);
00233 }
00234
00258 void fasp_fwapper_dbsr_krylov_ilu_(INT* n, INT* nnz, INT* nb, INT* ia, INT* ja,
00259     REAL* a, REAL* b, REAL* u, REAL* tol, INT* maxit,
00260     INT* ptrlvl)
00261 {
00262     dBSRmat mat; // coefficient matrix in BSR format
00263     dvector rhs, sol; // right-hand-side, solution
00264
00265     ILU_param iluparam; // parameters for ILU
00266     ITS_param itsparam; // parameters for itsolver
00267
00268     // setup ILU parameters
00269     fasp_param_ilu_init(&iluparam);
00270     iluparam.ILU_lfil = 0;
00271     iluparam.print_level = *ptrlvl;
00272
00273     // setup Krylov method parameters
00274     fasp_param_solver_init(&itsparam);
00275
00276     itsparam.itsolver_type = SOLVER_VFGMRES;
00277     itsparam.tol = *tol;
00278     itsparam.maxit = *maxit;
00279     itsparam.print_level = *ptrlvl;
00280
00281     // set up coefficient matrix
00282     mat.ROW = *n;
00283     mat.COL = *n;
00284     mat.NNZ = *nnz;
00285     mat.nb = *nb;
00286     mat.IA = ia;
00287     mat.JA = ja;
00288     mat.val = a;
00289
00290     rhs.row = *n * *nb;
00291     rhs.val = b;
00292     sol.row = *n * *nb;

```

```

00293     sol.val = u;
00294
00295     // solve
00296     fasp_solver_dbsr_krylov_ilu(&mat, &rhs, &sol, &itsparam, &iluparam);
00297 }
00298
00321 void fasp_fwrapper_dbsr_krylov_amg_(INT* n, INT* nnz, INT* nb, INT* ia, INT* ja,
00322                                     REAL* a, REAL* b, REAL* u, REAL* tol, INT* maxit,
00323                                     INT* ptrlvl)
00324 {
00325     dBSRmat mat; // coefficient matrix in CSR format
00326     dvector rhs, sol; // right-hand-side, solution
00327
00328     AMG_param amgparam; // parameters for AMG
00329     ITS_param itsparam; // parameters for itsolver
00330
00331     // setup AMG parameters
00332     fasp_param_amg_init(&amgparam);
00333     amgparam.AMG_type = UA_AMG;
00334     amgparam.print_level = *ptrlvl;
00335
00336     // setup Krylov method parameters
00337     fasp_param_solver_init(&itsparam);
00338     itsparam.tol = *tol;
00339     itsparam.print_level = *ptrlvl;
00340     itsparam.maxit = *maxit;
00341     itsparam.itsolver_type = SOLVER_VFGMR5;
00342
00343     // set up coefficient matrix
00344     mat.ROW = *n;
00345     mat.COL = *n;
00346     mat.NNZ = *nnz;
00347     mat.nb = *nb;
00348     mat.IA = ia;
00349     mat.JA = ja;
00350     mat.val = a;
00351
00352     rhs.row = *n * *nb;
00353     rhs.val = b;
00354     sol.row = *n * *nb;
00355     sol.val = u;
00356
00357     // solve
00358     fasp_solver_dbsr_krylov_amg(&mat, &rhs, &sol, &itsparam, &amgparam);
00359 }
00360
00361 /*-----*/
00362 /*--      End of File      --*/
00363 /*-----*/

```


Index

__FASPBLOCK_HEADER__
 fasp_block.h, [173](#)
__FASPGRID_HEADER__
 fasp_grid.h, [202](#)
__FASP_HEADER__
 fasp.h, [158](#)

A

 AMG_data, [18](#)
 AMG_data_bsr, [25](#)
 ILU_data, [73](#)
 precond_data, [100](#)
 precond_data_bsr, [109](#)
 precond_data_str, [116](#)
 precond_data_sweeping, [122](#)
 SWZ_data, [128](#)

A_diag

 precond_data_blc, [106](#)

A_nk

 AMG_data_bsr, [25](#)
 precond_data, [100](#)
 precond_data_bsr, [109](#)

A_str

 precond_data_str, [116](#)

Ablc

 precond_data_blc, [106](#)

ABS

 fasp.h, [159](#)

Ac

 AMG_data_bsr, [25](#)

aggregation_type

 AMG_param, [33](#)

aggressive_level

 AMG_param, [33](#)

aggressive_path

 AMG_param, [33](#)

Ai

 precond_data_sweeping, [122](#)

al

 SWZ_data, [128](#)

AMG_aggregation_type

 input_param, [81](#)

AMG_aggressive_level

 input_param, [81](#)

AMG_aggressive_path

 input_param, [81](#)

AMG_amli_degree

 input_param, [81](#)

AMG_coarse_dof

 input_param, [81](#)

AMG_coarse_scaling

 input_param, [82](#)

AMG_coarse_solver

 input_param, [82](#)

AMG_COARSEN_CR

 PreAMGCoarsenCR.c, [1101](#)

AMG_coarsening_type

 input_param, [82](#)

AMG_cycle_type

 input_param, [82](#)

AMG_data, [17](#)

 A, [18](#)

 b, [18](#)

 cfmark, [19](#)

 colors, [19](#)

 cycle_type, [19](#)

 ic, [19](#)

 icmap, [19](#)

 ILU_levels, [20](#)

 LU, [20](#)

 max_levels, [20](#)

 mumps, [20](#)

 near_kernel_basis, [20](#)

 near_kernel_dim, [21](#)

 num_levels, [21](#)

 Numeric, [21](#)

 P, [21](#)

 pdata, [21](#)

 R, [22](#)

 Schwarz, [22](#)

 SWZ_levels, [22](#)

 w, [22](#)

 weight, [22](#)

 x, [23](#)

AMG_data_bsr, [23](#)

 A, [25](#)

 A_nk, [25](#)

 Ac, [25](#)

 b, [25](#)

 cfmark, [25](#)

 diaginv, [26](#)

- diaginv_SS, 26
- ILU_levels, 26
- LU, 26
- max_levels, 26
- mumps, 27
- near_kernel_basis, 27
- near_kernel_dim, 27
- num_levels, 27
- Numeric, 27
- P, 28
- P_nk, 28
- pdata, 28
- PP, 28
- PP_LU, 28
- pw, 29
- R, 29
- R_nk, 29
- SS, 29
- sw, 29
- w, 30
- x, 30
- AMG_ILU_levels
 - input_param, 82
- AMG_interpolation_type
 - input_param, 83
- AMG_levels
 - input_param, 83
- AMG_max_aggregation
 - input_param, 83
- AMG_max_row_sum
 - input_param, 83
- AMG_maxit
 - input_param, 83
- AMG_nl_amli_krylov_type
 - input_param, 84
- AMG_pair_number
 - input_param, 84
- AMG_param, 30
 - aggregation_type, 33
 - aggressive_level, 33
 - aggressive_path, 33
 - AMG_type, 33
 - amli_coef, 34
 - amli_degree, 34
 - coarse_dof, 34
 - coarse_scaling, 34
 - coarse_solver, 34
 - coarsening_type, 35
 - cycle_type, 35
 - ILU_droptol, 35
 - ILU_levels, 35
 - ILU_lfil, 35
 - ILU_permtol, 36
 - ILU_relax, 36
 - ILU_type, 36
 - interpolation_type, 36
 - max_aggregation, 36
 - max_levels, 37
 - max_row_sum, 37
 - maxit, 37
 - nl_amli_krylov_type, 37
 - pair_number, 37
 - polynomial_degree, 38
 - postsmooth_iter, 38
 - presmooth_iter, 38
 - print_level, 38
 - quality_bound, 38
 - relaxation, 39
 - smooth_filter, 39
 - smooth_order, 39
 - smooth_restriction, 39
 - smoother, 39
 - strong_coupled, 40
 - strong_threshold, 40
 - SWZ_blksolver, 40
 - SWZ_levels, 40
 - SWZ_maxlvl, 40
 - SWZ_mmsize, 41
 - SWZ_type, 41
 - tentative_smooth, 41
 - tol, 41
 - truncation_threshold, 41
- AMG_polynomial_degree
 - input_param, 84
- AMG_postsmooth_iter
 - input_param, 84
- AMG_presmooth_iter
 - input_param, 84
- AMG_quality_bound
 - input_param, 85
- AMG_relaxation
 - input_param, 85
- AMG_smooth_filter
 - input_param, 85
- AMG_smooth_order
 - input_param, 85
- AMG_smooth_restriction
 - input_param, 85
- AMG_smoother
 - input_param, 86
- AMG_strong_coupled
 - input_param, 86
- AMG_strong_threshold
 - input_param, 86
- AMG_SWZ_levels
 - input_param, 86
- AMG_tentative_smooth
 - input_param, 86

- AMG_tol
 - input_param, 87
- AMG_truncation_threshold
 - input_param, 87
- AMG_type
 - AMG_param, 33
 - input_param, 87
 - precond_data, 100
 - precond_data_bsr, 109
 - precond_data_str, 116
- amgparam
 - precond_data_blc, 106
- amli_coef
 - AMG_param, 34
 - precond_data, 101
 - precond_data_bsr, 110
- AMLI_CYCLE
 - fasp_const.h, 181
- amli_degree
 - AMG_param, 34
 - precond_data, 101
 - precond_data_bsr, 110
- ASCEND
 - fasp_const.h, 181
- au
 - SWZ_data, 128
- AuxArray.c, 203, 207
 - fasp_darray_cp, 204
 - fasp_darray_set, 204
 - fasp_iarray_cp, 205
 - fasp_iarray_set, 205
- AuxConvert.c, 209, 211
 - fasp_aux_bbyteToldouble, 209
 - fasp_aux_change_endian4, 210
 - fasp_aux_change_endian8, 210
- AuxGivens.c, 211, 212
 - fasp_aux_givens, 212
- AuxGraphics.c, 213, 217
 - fasp_dbsr_plot, 214
 - fasp_dbsr_subplot, 214
 - fasp_dcsr_plot, 215
 - fasp_dcsr_subplot, 216
 - fasp_grid2d_plot, 216
- AuxInput.c, 223, 226
 - fasp_param_check, 224
 - fasp_param_input, 224
- AuxMemory.c, 236, 239
 - fasp_mem_calloc, 237
 - fasp_mem_free, 237
 - fasp_mem_iludata_check, 237
 - fasp_mem_realloc, 238
 - fasp_mem_usage, 238
 - Million, 239
- AuxMessage.c, 241, 245
 - fasp_amgcomplexity, 242
 - fasp_amgcomplexity_bsr, 242
 - fasp_chkerr, 242
 - fasp_cputime, 244
 - fasp_itinfo, 244
 - fasp_message, 245
- AuxParam.c, 248, 257
 - fasp_param_amg_init, 249
 - fasp_param_amg_print, 250
 - fasp_param_amg_set, 250
 - fasp_param_amg_to_prec, 251
 - fasp_param_amg_to_precbsr, 251
 - fasp_param_ilu_init, 251
 - fasp_param_ilu_print, 252
 - fasp_param_ilu_set, 252
 - fasp_param_init, 253
 - fasp_param_input_init, 253
 - fasp_param_prec_to_amg, 254
 - fasp_param_precbsr_to_amg, 254
 - fasp_param_set, 254
 - fasp_param_solver_init, 255
 - fasp_param_solver_print, 255
 - fasp_param_solver_set, 256
 - fasp_param_swz_init, 256
 - fasp_param_swz_print, 256
 - fasp_param_swz_set, 257
- AuxSort.c, 267, 273
 - fasp_aux_BiSearch, 268
 - fasp_aux_dQuickSort, 269
 - fasp_aux_dQuickSortIndex, 269
 - fasp_aux_iQuickSort, 270
 - fasp_aux_iQuickSortIndex, 270
 - fasp_aux_merge, 271
 - fasp_aux_msort, 271
 - fasp_aux_unique, 272
- AuxThreads.c, 276, 278
 - fasp_get_start_end, 276
 - fasp_set_gs_threads, 277
 - THDs_AMG_GS, 277
 - THDs_CPR_gGS, 277
 - THDs_CPR_IGS, 278
- AuxTiming.c, 279, 280
 - fasp_gettime, 280
- AuxVector.c, 281, 288
 - fasp_dvec_alloc, 282
 - fasp_dvec_cp, 282
 - fasp_dvec_create, 282
 - fasp_dvec_free, 283
 - fasp_dvec_isnan, 283
 - fasp_dvec_maxdiff, 284
 - fasp_dvec_rand, 284
 - fasp_dvec_set, 285
 - fasp_dvec_symdiagscale, 285
 - fasp_ivec_alloc, 286

- fasp_ivec_create, 286
 - fasp_ivec_free, 287
 - fasp_ivec_set, 287
- b
 - AMG_data, 18
 - AMG_data_bsr, 25
- bcol
 - dBLMat, 45
 - iBLMat, 65
- BIGREAL
 - fasp_const.h, 181
- BlaArray.c, 291, 300
 - fasp_blas_darray_ax, 292
 - fasp_blas_darray_axpby, 293
 - fasp_blas_darray_axpy, 293
 - fasp_blas_darray_axpy_nc2, 294
 - fasp_blas_darray_axpy_nc3, 294
 - fasp_blas_darray_axpy_nc5, 295
 - fasp_blas_darray_axpy_nc7, 295
 - fasp_blas_darray_axpyz, 296
 - fasp_blas_darray_axpyz_nc2, 296
 - fasp_blas_darray_axpyz_nc3, 297
 - fasp_blas_darray_axpyz_nc5, 297
 - fasp_blas_darray_axpyz_nc7, 298
 - fasp_blas_darray_dotprod, 298
 - fasp_blas_darray_norm1, 299
 - fasp_blas_darray_norm2, 299
 - fasp_blas_darray_norminf, 300
- BlaEigen.c, 307, 308
 - fasp_dcsr_maxeig, 307
- BlaFormat.c, 309, 314
 - fasp_format_dblc_dcsr, 310
 - fasp_format_dbsr_dcoo, 310
 - fasp_format_dbsr_dcsr, 310
 - fasp_format_dcoo_dcsr, 311
 - fasp_format_dcsr_dbsr, 311
 - fasp_format_dcsr_dcoo, 312
 - fasp_format_dcsr_dcsr, 312
 - fasp_format_dstr_dbsr, 313
 - fasp_format_dstr_dcsr, 313
- BlaILU.c, 325, 331
 - fasp_iluk, 326
 - fasp_ilut, 326
 - fasp_ilutp, 327
 - fasp_symbfactor, 329
- BlaILUSetupBSR.c, 353, 358
 - fasp_ilu_dbsr_setup, 354
 - fasp_ilu_dbsr_setup_levsch_omp, 354
 - fasp_ilu_dbsr_setup_levsch_step, 355
 - fasp_ilu_dbsr_setup_mc_omp, 355
 - fasp_ilu_dbsr_setup_omp, 356
 - fasp_ilu_dbsr_setup_step, 357
 - mulcol_independ_set, 357
- topologic_sort_ILU, 358
- BlaILUSetupCSR.c, 379, 380
 - fasp_ilu_dcsr_setup, 380
- BlaILUSetupSTR.c, 382, 384
 - fasp_ilu_dstr_setup0, 383
 - fasp_ilu_dstr_setup1, 383
- BlaIO.c, 399, 424
 - dlength, 424
 - fasp_dbsr_print, 401
 - fasp_dbsr_read, 402
 - fasp_dbsr_write, 402
 - fasp_dbsr_write_coo, 403
 - fasp_dcoo_print, 403
 - fasp_dcoo_read, 404
 - fasp_dcoo_read1, 404
 - fasp_dcoo_shift_read, 405
 - fasp_dcoo_write, 405
 - fasp_dcoovec_bin_read, 406
 - fasp_dcsr_print, 407
 - fasp_dcsr_read, 407
 - fasp_dcsr_write_coo, 408
 - fasp_dcsr_write_mtx, 408
 - fasp_dcsrvec_read1, 409
 - fasp_dcsrvec_read2, 409
 - fasp_dcsrvec_write1, 410
 - fasp_dcsrvec_write2, 411
 - fasp_dmtx_read, 412
 - fasp_dmtxsym_read, 412
 - fasp_dstr_print, 413
 - fasp_dstr_read, 413
 - fasp_dstr_write, 414
 - fasp_dvec_print, 414
 - fasp_dvec_read, 415
 - fasp_dvec_write, 415
 - fasp_dvecind_read, 416
 - fasp_dvecind_write, 416
 - fasp_hb_read, 417
 - fasp_ivec_print, 417
 - fasp_ivec_read, 418
 - fasp_ivec_write, 418
 - fasp_ivecind_read, 419
 - fasp_matrix_read, 419
 - fasp_matrix_read_bin, 420
 - fasp_matrix_write, 421
 - fasp_vector_read, 421
 - fasp_vector_write, 423
 - ilength, 424
- BlaOrderingCSR.c, 446, 447
 - fasp_dcsr_CMK_order, 446
 - fasp_dcsr_RCMK_order, 446
- BlaSchwarzSetup.c, 449, 451
 - fasp_dcsr_swz_backward, 449
 - fasp_dcsr_swz_forward, 450
 - fasp_swz_dcsr_setup, 450

- BlaSmallMat.c, [457](#), [476](#)
 - [fasp_blas_smat_aAxpby](#), [459](#)
 - [fasp_blas_smat_add](#), [460](#)
 - [fasp_blas_smat_axm](#), [460](#)
 - [fasp_blas_smat_mul](#), [461](#)
 - [fasp_blas_smat_mul_nc2](#), [461](#)
 - [fasp_blas_smat_mul_nc3](#), [462](#)
 - [fasp_blas_smat_mul_nc4](#), [462](#)
 - [fasp_blas_smat_mul_nc5](#), [463](#)
 - [fasp_blas_smat_mul_nc7](#), [463](#)
 - [fasp_blas_smat_mxv](#), [464](#)
 - [fasp_blas_smat_mxv_nc2](#), [464](#)
 - [fasp_blas_smat_mxv_nc3](#), [465](#)
 - [fasp_blas_smat_mxv_nc4](#), [465](#)
 - [fasp_blas_smat_mxv_nc5](#), [466](#)
 - [fasp_blas_smat_mxv_nc7](#), [466](#)
 - [fasp_blas_smat_yMAx](#), [467](#)
 - [fasp_blas_smat_yMAx_nc2](#), [467](#)
 - [fasp_blas_smat_yMAx_nc3](#), [468](#)
 - [fasp_blas_smat_yMAx_nc4](#), [468](#)
 - [fasp_blas_smat_yMAx_nc5](#), [470](#)
 - [fasp_blas_smat_yMAx_nc7](#), [470](#)
 - [fasp_blas_smat_ypAx](#), [472](#)
 - [fasp_blas_smat_ypAx_nc2](#), [472](#)
 - [fasp_blas_smat_ypAx_nc3](#), [474](#)
 - [fasp_blas_smat_ypAx_nc4](#), [474](#)
 - [fasp_blas_smat_ypAx_nc5](#), [475](#)
 - [fasp_blas_smat_ypAx_nc7](#), [475](#)
- BlaSmallMatInv.c, [485](#), [492](#)
 - [fasp_smat_identity](#), [486](#)
 - [fasp_smat_identity_nc2](#), [486](#)
 - [fasp_smat_identity_nc3](#), [487](#)
 - [fasp_smat_identity_nc5](#), [487](#)
 - [fasp_smat_identity_nc7](#), [488](#)
 - [fasp_smat_inv](#), [488](#)
 - [fasp_smat_inv_nc](#), [488](#)
 - [fasp_smat_inv_nc2](#), [489](#)
 - [fasp_smat_inv_nc3](#), [489](#)
 - [fasp_smat_inv_nc4](#), [490](#)
 - [fasp_smat_inv_nc5](#), [490](#)
 - [fasp_smat_inv_nc7](#), [490](#)
 - [fasp_smat_invp_nc](#), [491](#)
 - [fasp_smat_Linf](#), [491](#)
 - [SWAP](#), [486](#)
- BlaSmallMatLU.c, [501](#), [503](#)
 - [fasp_smat_lu_decomp](#), [502](#)
 - [fasp_smat_lu_solve](#), [502](#)
- BlaSparseBLC.c, [505](#), [506](#)
 - [fasp_dblc_free](#), [505](#)
- BlaSparseBSR.c, [506](#), [518](#)
 - [fasp_dbsr_alloc](#), [507](#)
 - [fasp_dbsr_cp](#), [508](#)
 - [fasp_dbsr_create](#), [508](#)
 - [fasp_dbsr_diaginv](#), [509](#)
 - [fasp_dbsr_diaginv2](#), [510](#)
 - [fasp_dbsr_diaginv3](#), [510](#)
 - [fasp_dbsr_diaginv4](#), [511](#)
 - [fasp_dbsr_diagLU](#), [511](#)
 - [fasp_dbsr_diagLU2](#), [512](#)
 - [fasp_dbsr_diagpref](#), [512](#)
 - [fasp_dbsr_free](#), [514](#)
 - [fasp_dbsr_getblk](#), [514](#)
 - [fasp_dbsr_getdiag](#), [515](#)
 - [fasp_dbsr_getdiaginv](#), [516](#)
 - [fasp_dbsr_merge_col](#), [516](#)
 - [fasp_dbsr_perm](#), [516](#)
 - [fasp_dbsr_trans](#), [517](#)
- BlaSparseCheck.c, [542](#), [547](#)
 - [fasp_check_dCSRmat](#), [543](#)
 - [fasp_check_diagdom](#), [543](#)
 - [fasp_check_diagpos](#), [544](#)
 - [fasp_check_diagzero](#), [544](#)
 - [fasp_check_iCSRmat](#), [545](#)
 - [fasp_check_ordering](#), [545](#)
 - [fasp_check_symm](#), [545](#)
- BlaSparseCOO.c, [551](#), [553](#)
 - [fasp_dcoo_alloc](#), [551](#)
 - [fasp_dcoo_create](#), [552](#)
 - [fasp_dcoo_free](#), [552](#)
 - [fasp_dcoo_shift](#), [553](#)
- BlaSparseCSR.c, [554](#), [571](#)
 - [dCSRmat_Multicoloring](#), [556](#)
 - [dCSRmat_Multicoloring_Strong_Coupled](#), [557](#)
 - [dCSRmat_Multicoloring_Theta](#), [557](#)
 - [fasp_dcsr_alloc](#), [558](#)
 - [fasp_dcsr_bandwidth](#), [558](#)
 - [fasp_dcsr_compress](#), [559](#)
 - [fasp_dcsr_compress_inplace](#), [559](#)
 - [fasp_dcsr_cp](#), [560](#)
 - [fasp_dcsr_create](#), [560](#)
 - [fasp_dcsr_diagpref](#), [561](#)
 - [fasp_dcsr_free](#), [561](#)
 - [fasp_dcsr_getblk](#), [562](#)
 - [fasp_dcsr_getcol](#), [562](#)
 - [fasp_dcsr_getdiag](#), [563](#)
 - [fasp_dcsr_multicoloring](#), [563](#)
 - [fasp_dcsr_perm](#), [564](#)
 - [fasp_dcsr_permz](#), [564](#)
 - [fasp_dcsr_regdiag](#), [565](#)
 - [fasp_dcsr_shift](#), [565](#)
 - [fasp_dcsr_sort](#), [566](#)
 - [fasp_dcsr_sortz](#), [566](#)
 - [fasp_dcsr_symdiagscale](#), [567](#)
 - [fasp_dcsr_sympart](#), [567](#)
 - [fasp_dcsr_trans](#), [568](#)
 - [fasp_dcsr_transpose](#), [568](#)
 - [fasp_dcsr_transz](#), [569](#)
 - [fasp_icsr_cp](#), [569](#)

- fasp_icsr_create, 570
 - fasp_icsr_free, 570
 - fasp_icsr_trans, 571
 - fasp_smoother_dcsr_gs_multicolor, 571
- BlaSparseCSRL.c, 593, 594
 - fasp_dcsr_create, 594
 - fasp_dcsr_free, 594
- BlaSparseSTR.c, 595, 598
 - fasp_dstr_alloc, 596
 - fasp_dstr_cp, 596
 - fasp_dstr_create, 597
 - fasp_dstr_free, 597
- BlaSparseUtil.c, 599, 609
 - fasp_sparse_aat_, 601
 - fasp_sparse_abyb_, 601
 - fasp_sparse_abybms_, 602
 - fasp_sparse_aplbms_, 602
 - fasp_sparse_aplusb_, 603
 - fasp_sparse_iit_, 604
 - fasp_sparse_mis, 604
 - fasp_sparse_rapcmp_, 605
 - fasp_sparse_rapms_, 606
 - fasp_sparse_wta_, 606
 - fasp_sparse_wtams_, 607
 - fasp_sparse_ytx_, 608
 - fasp_sparse_ytxbig_, 608
- BlaSpmvBLC.c, 618, 620
 - fasp_blas_dbldc_aAxy, 618
 - fasp_blas_dbldc_mnv, 619
 - fasp_blas_ldbldc_aAxy, 619
- BlaSpmvBSR.c, 624, 631
 - fasp_blas_dbsr_aAxyby, 625
 - fasp_blas_dbsr_aAxy, 626
 - fasp_blas_dbsr_aAxy_agg, 626
 - fasp_blas_dbsr_axm, 627
 - fasp_blas_dbsr_mnm, 627
 - fasp_blas_dbsr_mnv, 628
 - fasp_blas_dbsr_mnv_agg, 628
 - fasp_blas_dbsr_rap, 629
 - fasp_blas_dbsr_rap1, 629
 - fasp_blas_dbsr_rap_agg, 630
- BlaSpmvCSR.c, 696, 706
 - fasp_blas_dcsr_aAxy, 698
 - fasp_blas_dcsr_aAxy_agg, 698
 - fasp_blas_dcsr_add, 699
 - fasp_blas_dcsr_axm, 699
 - fasp_blas_dcsr_mnm, 700
 - fasp_blas_dcsr_mnv, 700
 - fasp_blas_dcsr_mnv_agg, 701
 - fasp_blas_dcsr_ptap, 701
 - fasp_blas_dcsr_rap, 702
 - fasp_blas_dcsr_rap2, 702
 - fasp_blas_dcsr_rap4, 703
 - fasp_blas_dcsr_rap_agg, 704
- fasp_blas_dcsr_rap_agg1, 704
 - fasp_blas_dcsr_vmv, 705
 - fasp_blas_ldcsr_aAxy, 705
 - total_alloc_count, 706
 - total_alloc_mem, 706
- BlaSpmvCSRL.c, 730, 731
 - fasp_blas_dcsr_mnv, 731
- BlaSpmvSTR.c, 733, 735
 - fasp_blas_dstr_aAxy, 733
 - fasp_blas_dstr_diagscale, 734
 - fasp_blas_dstr_mnv, 734
- BlaVector.c, 754, 758
 - fasp_blas_dvec_axpy, 755
 - fasp_blas_dvec_axpyz, 755
 - fasp_blas_dvec_dotprod, 756
 - fasp_blas_dvec_norm1, 756
 - fasp_blas_dvec_norm2, 757
 - fasp_blas_dvec_norminf, 757
 - fasp_blas_dvec_reterr, 758
- blk_data
 - SWZ_data, 128
- blk_solver
 - SWZ_data, 128
- block_dvector, 42
 - blocks, 42
 - brow, 42
 - fasp_block.h, 174
- block_ivector, 43
 - blocks, 43
 - brow, 44
 - fasp_block.h, 174
- blocks
 - block_dvector, 42
 - block_ivector, 43
 - dBLCmat, 45
 - iBLCmat, 65
- brow
 - block_dvector, 42
 - block_ivector, 44
 - dBLCmat, 45
 - iBLCmat, 65
- CF_ORDER
 - fasp_const.h, 181
- cfmark
 - AMG_data, 19
 - AMG_data_bsr, 25
- CGPT
 - fasp_const.h, 181
- CLASSIC_AMG
 - fasp_const.h, 181
- COARSE_AC
 - fasp_const.h, 181
- COARSE_CR

- fasp_const.h, [182](#)
- coarse_dof
 - AMG_param, [34](#)
- COARSE_MIS
 - fasp_const.h, [182](#)
- COARSE_RS
 - fasp_const.h, [182](#)
- COARSE_RSP
 - fasp_const.h, [182](#)
- coarse_scaling
 - AMG_param, [34](#)
 - precond_data, [101](#)
 - precond_data_bsr, [110](#)
 - precond_data_str, [117](#)
- coarse_solver
 - AMG_param, [34](#)
 - precond_data, [101](#)
 - precond_data_bsr, [110](#)
- coarsening_type
 - AMG_param, [35](#)
 - precond_data, [101](#)
 - precond_data_bsr, [110](#)
 - precond_data_str, [117](#)
- COL
 - dBSRmat, [46](#)
- col
 - dCOOmat, [49](#)
 - dCSRLmat, [51](#)
 - dCSRmat, [54](#)
 - ddenmat, [56](#)
 - iCOOmat, [67](#)
 - iCSRmat, [69](#)
 - idenmat, [71](#)
 - ILU_data, [73](#)
- colind
 - dCOOmat, [49](#)
- colors
 - AMG_data, [19](#)
- CPFIRST
 - fasp_const.h, [182](#)
- cycle_type
 - AMG_data, [19](#)
 - AMG_param, [35](#)
 - precond_data, [102](#)
 - precond_data_bsr, [111](#)
 - precond_data_str, [117](#)
- data
 - mxv_matfree, [96](#)
 - precond, [98](#)
- dBLCmat, [44](#)
 - bcol, [45](#)
 - blocks, [45](#)
 - brow, [45](#)
- fasp_block.h, [174](#)
- dBSRmat, [45](#)
 - COL, [46](#)
 - fasp_block.h, [174](#)
 - IA, [46](#)
 - JA, [47](#)
 - nb, [47](#)
 - NNZ, [47](#)
 - ROW, [47](#)
 - storage_manner, [47](#)
 - val, [48](#)
- dCOOmat, [48](#)
 - col, [49](#)
 - colind, [49](#)
 - fasp.h, [162](#)
 - nnz, [49](#)
 - row, [49](#)
 - rowind, [50](#)
 - val, [50](#)
- dCSRLmat, [50](#)
 - col, [51](#)
 - dif, [51](#)
 - fasp.h, [162](#)
 - index, [52](#)
 - ja, [52](#)
 - nnz, [52](#)
 - nz_diff, [52](#)
 - row, [52](#)
 - start, [53](#)
 - val, [53](#)
- dCSRmat, [53](#)
 - col, [54](#)
 - fasp.h, [162](#)
 - IA, [54](#)
 - JA, [55](#)
 - nnz, [55](#)
 - row, [55](#)
 - val, [55](#)
- dCSRmat2SAMGInput
 - XtrSamg.c, [148](#)
- dCSRmat_Multicoloring
 - BlaSparseCSR.c, [556](#)
- dCSRmat_Multicoloring_Strong_Coupled
 - BlaSparseCSR.c, [557](#)
- dCSRmat_Multicoloring_Theta
 - BlaSparseCSR.c, [557](#)
- ddenmat, [56](#)
 - col, [56](#)
 - fasp.h, [162](#)
 - row, [56](#)
 - val, [56](#)
- decoup_type
 - input_param, [87](#)
 - ITS_param, [92](#)

DESCEND
 fasp_const.h, 182
 diag
 dSTRmat, 58
 precond_diag_bsr, 125
 precond_diag_str, 126
 diaginv
 AMG_data_bsr, 26
 precond_data_str, 117
 diaginv_SS
 AMG_data_bsr, 26
 diaginvS
 precond_data_str, 117
 DIAGONAL_PREF
 fasp.h, 159
 dif
 dCSRmat, 51
 dlength
 BlalO.c, 424
 DLMALLOC
 fasp.h, 159
 doxygen.h, 135
 dSTRmat, 57
 diag, 58
 fasp.h, 162
 nband, 58
 nc, 58
 ngrid, 58
 nx, 59
 nxy, 59
 ny, 59
 nz, 59
 offdiag, 59
 offsets, 60
 dvector, 60
 fasp.h, 162
 row, 60
 val, 61
 dvector2SAMGInput
 XtrSamg.c, 148

 e
 grid2d, 62
 edges
 grid2d, 62
 ediri
 grid2d, 62
 efather
 grid2d, 62
 ERROR_ALLOC_MEM
 fasp_const.h, 182
 ERROR_AMG_COARSE_TYPE
 fasp_const.h, 182
 ERROR_AMG_COARSEING
 fasp_const.h, 183
 ERROR_AMG_INTERP_TYPE
 fasp_const.h, 183
 ERROR_AMG_SETUP
 fasp_const.h, 183
 ERROR_AMG_SMOOTH_TYPE
 fasp_const.h, 183
 ERROR_DATA_STRUCTURE
 fasp_const.h, 183
 ERROR_DATA_ZERODIAG
 fasp_const.h, 183
 ERROR_DUMMY_VAR
 fasp_const.h, 183
 ERROR_INPUT_PAR
 fasp_const.h, 183
 ERROR_LIC_TYPE
 fasp_const.h, 184
 ERROR_MAT_SIZE
 fasp_const.h, 184
 ERROR_MISC
 fasp_const.h, 184
 ERROR_NUM_BLOCKS
 fasp_const.h, 184
 ERROR_OPEN_FILE
 fasp_const.h, 184
 ERROR_QUAD_DIM
 fasp_const.h, 184
 ERROR_QUAD_TYPE
 fasp_const.h, 184
 ERROR_READ_FILE
 fasp_const.h, 184
 ERROR_REGRESS
 fasp_const.h, 185
 ERROR_SOLVER_EXIT
 fasp_const.h, 185
 ERROR_SOLVER_ILUSETUP
 fasp_const.h, 185
 ERROR_SOLVER_MAXIT
 fasp_const.h, 185
 ERROR_SOLVER_MISC
 fasp_const.h, 185
 ERROR_SOLVER_PRECTYPE
 fasp_const.h, 185
 ERROR_SOLVER_SOLSTAG
 fasp_const.h, 185
 ERROR_SOLVER_STAG
 fasp_const.h, 185
 ERROR_SOLVER_TOLSMALL
 fasp_const.h, 186
 ERROR_SOLVER_TYPE
 fasp_const.h, 186
 ERROR_UNKNOWN
 fasp_const.h, 186
 ERROR_WRONG_FILE

fasp_const.h, [186](#)

FALSE

 fasp_const.h, [186](#)

fasp.h, [156](#), [163](#)

 __FASP_HEADER__, [158](#)

 ABS, [159](#)

 dCOOmat, [162](#)

 dCSRmat, [162](#)

 dCSRmat, [162](#)

 ddenmat, [162](#)

 DIAGONAL_PREF, [159](#)

 DLMALLOC, [159](#)

 dSTRmat, [162](#)

 dvector, [162](#)

 FASP_VERSION, [159](#)

 GE, [159](#)

 GT, [159](#)

 iCOOmat, [163](#)

 iCSRmat, [163](#)

 idenmat, [163](#)

 INT, [159](#)

 ISNAN, [160](#)

 ivector, [163](#)

 LE, [160](#)

 LONG, [160](#)

 LONGLONG, [160](#)

 LONGREAL, [160](#)

 LS, [160](#)

 MAX, [160](#)

 MIN, [161](#)

 MULTI_COLOR_ORDER, [161](#)

 NEDMALLOC, [161](#)

 PUT_INT, [161](#)

 PUT_REAL, [161](#)

 REAL, [161](#)

 RS_C1, [162](#)

 SHORT, [162](#)

 STRLEN, [162](#)

fasp_amg_amli_coef

 PreMGRecurAMLI.c, [1263](#)

fasp_amg_coarsening_cr

 PreAMGCoarsenCR.c, [1101](#)

fasp_amg_coarsening_rs

 PreAMGCoarsenRS.c, [1107](#)

fasp_amg_data_bsr_create

 PreDataInit.c, [1240](#)

fasp_amg_data_bsr_free

 PreDataInit.c, [1240](#)

fasp_amg_data_create

 PreDataInit.c, [1241](#)

fasp_amg_data_free

 PreDataInit.c, [1241](#)

fasp_amg_interp

 PreAMGInterp.c, [1131](#)

fasp_amg_interp_em

 PreAMGInterpEM.c, [1142](#)

fasp_amg_setup_cr

 PreAMGSetupCR.c, [1151](#)

fasp_amg_setup_rs

 PreAMGSetupRS.c, [1154](#)

fasp_amg_setup_sa

 PreAMGSetupSA.c, [1159](#)

fasp_amg_setup_sa_bsr

 PreAMGSetupSABSR.c, [1169](#)

fasp_amg_setup_ua

 PreAMGSetupUA.c, [1175](#)

fasp_amg_setup_ua_bsr

 PreAMGSetupUABSR.c, [1181](#)

fasp_amg_solve

 PreMGSolve.c, [1275](#)

fasp_amg_solve_amli

 PreMGSolve.c, [1276](#)

fasp_amg_solve_namli

 PreMGSolve.c, [1276](#)

fasp_amgcomplexity

 AuxMessage.c, [242](#)

fasp_amgcomplexity_bsr

 AuxMessage.c, [242](#)

fasp_aux_bbyteToldouble

 AuxConvert.c, [209](#)

fasp_aux_BiSearch

 AuxSort.c, [268](#)

fasp_aux_change_endian4

 AuxConvert.c, [210](#)

fasp_aux_change_endian8

 AuxConvert.c, [210](#)

fasp_aux_dQuickSort

 AuxSort.c, [269](#)

fasp_aux_dQuickSortIndex

 AuxSort.c, [269](#)

fasp_aux_givens

 AuxGivens.c, [212](#)

fasp_aux_iQuickSort

 AuxSort.c, [270](#)

fasp_aux_iQuickSortIndex

 AuxSort.c, [270](#)

fasp_aux_merge

 AuxSort.c, [271](#)

fasp_aux_msort

 AuxSort.c, [271](#)

fasp_aux_unique

 AuxSort.c, [272](#)

fasp_blas_darray_ax

 BlaArray.c, [292](#)

fasp_blas_darray_axpby

 BlaArray.c, [293](#)

fasp_blas_darray_axpy

BlaArray.c, [293](#)
fasp_blas_darray_axpy_nc2
 BlaArray.c, [294](#)
fasp_blas_darray_axpy_nc3
 BlaArray.c, [294](#)
fasp_blas_darray_axpy_nc5
 BlaArray.c, [295](#)
fasp_blas_darray_axpy_nc7
 BlaArray.c, [295](#)
fasp_blas_darray_axpyz
 BlaArray.c, [296](#)
fasp_blas_darray_axpyz_nc2
 BlaArray.c, [296](#)
fasp_blas_darray_axpyz_nc3
 BlaArray.c, [297](#)
fasp_blas_darray_axpyz_nc5
 BlaArray.c, [297](#)
fasp_blas_darray_axpyz_nc7
 BlaArray.c, [298](#)
fasp_blas_darray_dotprod
 BlaArray.c, [298](#)
fasp_blas_darray_norm1
 BlaArray.c, [299](#)
fasp_blas_darray_norm2
 BlaArray.c, [299](#)
fasp_blas_darray_norminf
 BlaArray.c, [300](#)
fasp_blas_dblc_aAxy
 BlaSpmvBLC.c, [618](#)
fasp_blas_dblc_mxv
 BlaSpmvBLC.c, [619](#)
fasp_blas_dbsr_aAxpby
 BlaSpmvBSR.c, [625](#)
fasp_blas_dbsr_aAxy
 BlaSpmvBSR.c, [626](#)
fasp_blas_dbsr_aAxy_agg
 BlaSpmvBSR.c, [626](#)
fasp_blas_dbsr_axm
 BlaSpmvBSR.c, [627](#)
fasp_blas_dbsr_mxm
 BlaSpmvBSR.c, [627](#)
fasp_blas_dbsr_mxv
 BlaSpmvBSR.c, [628](#)
fasp_blas_dbsr_mxv_agg
 BlaSpmvBSR.c, [628](#)
fasp_blas_dbsr_rap
 BlaSpmvBSR.c, [629](#)
fasp_blas_dbsr_rap1
 BlaSpmvBSR.c, [629](#)
fasp_blas_dbsr_rap_agg
 BlaSpmvBSR.c, [630](#)
fasp_blas_dcsr_aAxy
 BlaSpmvCSR.c, [698](#)
fasp_blas_dcsr_aAxy_agg
 BlaSpmvCSR.c, [698](#)
fasp_blas_dcsr_add
 BlaSpmvCSR.c, [699](#)
fasp_blas_dcsr_axm
 BlaSpmvCSR.c, [699](#)
fasp_blas_dcsr_mxm
 BlaSpmvCSR.c, [700](#)
fasp_blas_dcsr_mxv
 BlaSpmvCSR.c, [700](#)
fasp_blas_dcsr_mxv_agg
 BlaSpmvCSR.c, [701](#)
fasp_blas_dcsr_ptap
 BlaSpmvCSR.c, [701](#)
fasp_blas_dcsr_rap
 BlaSpmvCSR.c, [702](#)
fasp_blas_dcsr_rap2
 BlaSpmvCSR.c, [702](#)
fasp_blas_dcsr_rap4
 BlaSpmvCSR.c, [703](#)
fasp_blas_dcsr_rap_agg
 BlaSpmvCSR.c, [704](#)
fasp_blas_dcsr_rap_agg1
 BlaSpmvCSR.c, [704](#)
fasp_blas_dcsr_vmv
 BlaSpmvCSR.c, [705](#)
fasp_blas_dcsr_mxv
 BlaSpmvCSR.c, [731](#)
fasp_blas_dstr_aAxy
 BlaSpmvSTR.c, [733](#)
fasp_blas_dstr_diagscale
 BlaSpmvSTR.c, [734](#)
fasp_blas_dstr_mxv
 BlaSpmvSTR.c, [734](#)
fasp_blas_dvec_axpy
 BlaVector.c, [755](#)
fasp_blas_dvec_axpyz
 BlaVector.c, [755](#)
fasp_blas_dvec_dotprod
 BlaVector.c, [756](#)
fasp_blas_dvec_norm1
 BlaVector.c, [756](#)
fasp_blas_dvec_norm2
 BlaVector.c, [757](#)
fasp_blas_dvec_norminf
 BlaVector.c, [757](#)
fasp_blas_dvec_relerr
 BlaVector.c, [758](#)
fasp_blas_ldblc_aAxy
 BlaSpmvBLC.c, [619](#)
fasp_blas_ldcsr_aAxy
 BlaSpmvCSR.c, [705](#)
fasp_blas_smat_aAxpby
 BlaSmallMat.c, [459](#)
fasp_blas_smat_add

- BlaSmallMat.c, [460](#)
- fasp_blas_smat_axm
 - BlaSmallMat.c, [460](#)
- fasp_blas_smat_mul
 - BlaSmallMat.c, [461](#)
- fasp_blas_smat_mul_nc2
 - BlaSmallMat.c, [461](#)
- fasp_blas_smat_mul_nc3
 - BlaSmallMat.c, [462](#)
- fasp_blas_smat_mul_nc4
 - BlaSmallMat.c, [462](#)
- fasp_blas_smat_mul_nc5
 - BlaSmallMat.c, [463](#)
- fasp_blas_smat_mul_nc7
 - BlaSmallMat.c, [463](#)
- fasp_blas_smat_mxv
 - BlaSmallMat.c, [464](#)
- fasp_blas_smat_mxv_nc2
 - BlaSmallMat.c, [464](#)
- fasp_blas_smat_mxv_nc3
 - BlaSmallMat.c, [465](#)
- fasp_blas_smat_mxv_nc4
 - BlaSmallMat.c, [465](#)
- fasp_blas_smat_mxv_nc5
 - BlaSmallMat.c, [466](#)
- fasp_blas_smat_mxv_nc7
 - BlaSmallMat.c, [466](#)
- fasp_blas_smat_ymAxBlaSmallMat.c, [467](#)
- fasp_blas_smat_ymAxBnc2BlaSmallMat.c, [467](#)
- fasp_blas_smat_ymAxBnc3BlaSmallMat.c, [468](#)
- fasp_blas_smat_ymAxBnc4BlaSmallMat.c, [468](#)
- fasp_blas_smat_ymAxBnc5BlaSmallMat.c, [470](#)
- fasp_blas_smat_ymAxBnc7BlaSmallMat.c, [470](#)
- fasp_blas_smat_ypAxBlaSmallMat.c, [472](#)
- fasp_blas_smat_ypAxBnc2BlaSmallMat.c, [472](#)
- fasp_blas_smat_ypAxBnc3BlaSmallMat.c, [474](#)
- fasp_blas_smat_ypAxBnc4BlaSmallMat.c, [474](#)
- fasp_blas_smat_ypAxBnc5BlaSmallMat.c, [475](#)
- fasp_blas_smat_ypAxBnc7BlaSmallMat.c, [475](#)
- fasp_block.h, [172](#), [174](#)
 - __FASPBLOCK_HEADER__, [173](#)
 - block_dvector, [174](#)
 - block_ivector, [174](#)
 - dBLCmat, [174](#)
 - dBSRmat, [174](#)
 - iBLCmat, [174](#)
- fasp_check_dCSRmat
 - BlaSparseCheck.c, [543](#)
- fasp_check_diagdom
 - BlaSparseCheck.c, [543](#)
- fasp_check_diagpos
 - BlaSparseCheck.c, [544](#)
- fasp_check_diagzero
 - BlaSparseCheck.c, [544](#)
- fasp_check_iCSRmat
 - BlaSparseCheck.c, [545](#)
- fasp_check_ordering
 - BlaSparseCheck.c, [545](#)
- fasp_check_symm
 - BlaSparseCheck.c, [545](#)
- fasp_chkerr
 - AuxMessage.c, [242](#)
- fasp_const.h, [177](#), [200](#)
 - AMLI_CYCLE, [181](#)
 - ASCEND, [181](#)
 - BIGREAL, [181](#)
 - CF_ORDER, [181](#)
 - CGPT, [181](#)
 - CLASSIC_AMG, [181](#)
 - COARSE_AC, [181](#)
 - COARSE_CR, [182](#)
 - COARSE_MIS, [182](#)
 - COARSE_RS, [182](#)
 - COARSE_RSP, [182](#)
 - CPFIRST, [182](#)
 - DESCEND, [182](#)
 - ERROR_ALLOC_MEM, [182](#)
 - ERROR_AMG_COARSE_TYPE, [182](#)
 - ERROR_AMG_COARSEING, [183](#)
 - ERROR_AMG_INTERP_TYPE, [183](#)
 - ERROR_AMG_SETUP, [183](#)
 - ERROR_AMG_SMOOTH_TYPE, [183](#)
 - ERROR_DATA_STRUCTURE, [183](#)
 - ERROR_DATA_ZERODIAG, [183](#)
 - ERROR_DUMMY_VAR, [183](#)
 - ERROR_INPUT_PAR, [183](#)
 - ERROR_LIC_TYPE, [184](#)
 - ERROR_MAT_SIZE, [184](#)
 - ERROR_MISC, [184](#)
 - ERROR_NUM_BLOCKS, [184](#)
 - ERROR_OPEN_FILE, [184](#)
 - ERROR_QUAD_DIM, [184](#)
 - ERROR_QUAD_TYPE, [184](#)
 - ERROR_READ_FILE, [184](#)
 - ERROR_REGRESS, [185](#)
 - ERROR_SOLVER_EXIT, [185](#)

ERROR_SOLVER_ILUSETUP, 185
ERROR_SOLVER_MAXIT, 185
ERROR_SOLVER_MISC, 185
ERROR_SOLVER_PRECTYPE, 185
ERROR_SOLVER_SOLSTAG, 185
ERROR_SOLVER_STAG, 185
ERROR_SOLVER_TOLSMALL, 186
ERROR_SOLVER_TYPE, 186
ERROR_UNKNOWN, 186
ERROR_WRONG_FILE, 186
FALSE, 186
FASP_SUCCESS, 186
FGPT, 186
FPFIRST, 187
FPNA_RATIO, 187
G0PT, 187
ILUk, 187
ILUt, 187
ILUtp, 187
INTERP_DIR, 187
INTERP_ENG, 187
INTERP_EXT, 188
INTERP_STD, 188
ISPT, 188
MAT_bBSR, 188
MAT_bCSR, 188
MAT_BLC, 188
MAT_BSR, 188
MAT_bSTR, 188
MAT_CSR, 189
MAT_CSRL, 189
MAT_FREE, 189
MAT_STR, 189
MAT_SymCSR, 189
MAX_AMG_LVL, 189
MAX_CRATE, 189
MAX_REFINE_LVL, 189
MAX_RESTART, 190
MAX_STAG, 190
MIN_CDOF, 190
MIN_CRATE, 190
NL_AMLI_CYCLE, 190
NO_ORDER, 190
NPAIR, 190
OFF, 190
ON, 191
OPENMP_HOLDS, 191
PAIRWISE, 191
PREC_AMG, 191
PREC_DIAG, 191
PREC_FMG, 191
PREC_ILU, 191
PREC_NULL, 192
PREC_SCHWARZ, 192
PRINT_ALL, 192
PRINT_MIN, 192
PRINT_MORE, 192
PRINT_MOST, 192
PRINT_NONE, 192
PRINT_SOME, 192
SA_AMG, 193
SCHWARZ_BACKWARD, 193
SCHWARZ_FORWARD, 193
SCHWARZ_SYMMETRIC, 193
SMALLREAL, 193
SMALLREAL2, 193
SMOOTHER_BLKOil, 193
SMOOTHER_CG, 194
SMOOTHER_GS, 194
SMOOTHER_GSOR, 194
SMOOTHER_JACOBI, 194
SMOOTHER_L1DIAG, 194
SMOOTHER_POLY, 194
SMOOTHER_SGS, 194
SMOOTHER_SGSOR, 194
SMOOTHER_SOR, 195
SMOOTHER_SPETEN, 195
SMOOTHER_SSOR, 195
SOLVER_AMG, 195
SOLVER_BiCGstab, 195
SOLVER_CG, 195
SOLVER_DEFAULT, 195
SOLVER_FMG, 195
SOLVER_GCG, 196
SOLVER_GCR, 196
SOLVER_GMRES, 196
SOLVER_MinRes, 196
SOLVER_MUMPS, 196
SOLVER_PARDISO, 196
SOLVER_SBiCGstab, 196
SOLVER_SCG, 196
SOLVER_SGCG, 197
SOLVER_SGMRES, 197
SOLVER_SMinRes, 197
SOLVER_SUPERLU, 197
SOLVER_SVFGMRES, 197
SOLVER_SVGMRES, 197
SOLVER_UMFPACK, 197
SOLVER_VFGMRES, 197
SOLVER_VGMRES, 198
SPAIR, 198
STAG_RATIO, 198
STOP_MOD_REL_RES, 198
STOP_REL_PRECRES, 198
STOP_REL_RES, 198
TRUE, 198
UA_AMG, 199
UNPT, 199

- USERDEFINED, [199](#)
- V_CYCLE, [199](#)
- VMB, [199](#)
- VW_CYCLE, [199](#)
- W_CYCLE, [199](#)
- WV_CYCLE, [199](#)
- fasp_cputime
 - AuxMessage.c, [244](#)
- fasp_darray_cp
 - AuxArray.c, [204](#)
- fasp_darray_set
 - AuxArray.c, [204](#)
- fasp_dblc_free
 - BlaSparseBLC.c, [505](#)
- fasp_dbsr_alloc
 - BlaSparseBSR.c, [507](#)
- fasp_dbsr_cp
 - BlaSparseBSR.c, [508](#)
- fasp_dbsr_create
 - BlaSparseBSR.c, [508](#)
- fasp_dbsr_diaginv
 - BlaSparseBSR.c, [509](#)
- fasp_dbsr_diaginv2
 - BlaSparseBSR.c, [510](#)
- fasp_dbsr_diaginv3
 - BlaSparseBSR.c, [510](#)
- fasp_dbsr_diaginv4
 - BlaSparseBSR.c, [511](#)
- fasp_dbsr_diagLU
 - BlaSparseBSR.c, [511](#)
- fasp_dbsr_diagLU2
 - BlaSparseBSR.c, [512](#)
- fasp_dbsr_diagpref
 - BlaSparseBSR.c, [512](#)
- fasp_dbsr_free
 - BlaSparseBSR.c, [514](#)
- fasp_dbsr_getblk
 - BlaSparseBSR.c, [514](#)
- fasp_dbsr_getdiag
 - BlaSparseBSR.c, [515](#)
- fasp_dbsr_getdiaginv
 - BlaSparseBSR.c, [516](#)
- fasp_dbsr_merge_col
 - BlaSparseBSR.c, [516](#)
- fasp_dbsr_perm
 - BlaSparseBSR.c, [516](#)
- fasp_dbsr_plot
 - AuxGraphics.c, [214](#)
- fasp_dbsr_print
 - BlaIO.c, [401](#)
- fasp_dbsr_read
 - BlaIO.c, [402](#)
- fasp_dbsr_subplot
 - AuxGraphics.c, [214](#)
- fasp_dbsr_trans
 - BlaSparseBSR.c, [517](#)
- fasp_dbsr_write
 - BlaIO.c, [402](#)
- fasp_dbsr_write_coo
 - BlaIO.c, [403](#)
- fasp_dcoo_alloc
 - BlaSparseCOO.c, [551](#)
- fasp_dcoo_create
 - BlaSparseCOO.c, [552](#)
- fasp_dcoo_free
 - BlaSparseCOO.c, [552](#)
- fasp_dcoo_print
 - BlaIO.c, [403](#)
- fasp_dcoo_read
 - BlaIO.c, [404](#)
- fasp_dcoo_read1
 - BlaIO.c, [404](#)
- fasp_dcoo_shift
 - BlaSparseCOO.c, [553](#)
- fasp_dcoo_shift_read
 - BlaIO.c, [405](#)
- fasp_dcoo_write
 - BlaIO.c, [405](#)
- fasp_dcoovec_bin_read
 - BlaIO.c, [406](#)
- fasp_dcsr_alloc
 - BlaSparseCSR.c, [558](#)
- fasp_dcsr_bandwidth
 - BlaSparseCSR.c, [558](#)
- fasp_dcsr_CMK_order
 - BlaOrderingCSR.c, [446](#)
- fasp_dcsr_compress
 - BlaSparseCSR.c, [559](#)
- fasp_dcsr_compress_inplace
 - BlaSparseCSR.c, [559](#)
- fasp_dcsr_cp
 - BlaSparseCSR.c, [560](#)
- fasp_dcsr_create
 - BlaSparseCSR.c, [560](#)
- fasp_dcsr_diagpref
 - BlaSparseCSR.c, [561](#)
- fasp_dcsr_free
 - BlaSparseCSR.c, [561](#)
- fasp_dcsr_getblk
 - BlaSparseCSR.c, [562](#)
- fasp_dcsr_getcol
 - BlaSparseCSR.c, [562](#)
- fasp_dcsr_getdiag
 - BlaSparseCSR.c, [563](#)
- fasp_dcsr_maxeig
 - BlaEigen.c, [307](#)
- fasp_dcsr_multicoloring
 - BlaSparseCSR.c, [563](#)

`fasp_dcsr_perm`
 `BlaSparseCSR.c`, 564

`fasp_dcsr_permz`
 `BlaSparseCSR.c`, 564

`fasp_dcsr_plot`
 `AuxGraphics.c`, 215

`fasp_dcsr_print`
 `BlalO.c`, 407

`fasp_dcsr_RCMK_order`
 `BlaOrderingCSR.c`, 446

`fasp_dcsr_read`
 `BlalO.c`, 407

`fasp_dcsr_regdiag`
 `BlaSparseCSR.c`, 565

`fasp_dcsr_shift`
 `BlaSparseCSR.c`, 565

`fasp_dcsr_sort`
 `BlaSparseCSR.c`, 566

`fasp_dcsr_sortz`
 `BlaSparseCSR.c`, 566

`fasp_dcsr_subplot`
 `AuxGraphics.c`, 216

`fasp_dcsr_swz_backward`
 `BlaSchwarzSetup.c`, 449

`fasp_dcsr_swz_forward`
 `BlaSchwarzSetup.c`, 450

`fasp_dcsr_symdiagscale`
 `BlaSparseCSR.c`, 567

`fasp_dcsr_sympart`
 `BlaSparseCSR.c`, 567

`fasp_dcsr_trans`
 `BlaSparseCSR.c`, 568

`fasp_dcsr_transpose`
 `BlaSparseCSR.c`, 568

`fasp_dcsr_transz`
 `BlaSparseCSR.c`, 569

`fasp_dcsr_write_coo`
 `BlalO.c`, 408

`fasp_dcsr_write_mtx`
 `BlalO.c`, 408

`fasp_dcsr_create`
 `BlaSparseCSRL.c`, 594

`fasp_dcsr_free`
 `BlaSparseCSRL.c`, 594

`fasp_dcsrvec_read1`
 `BlalO.c`, 409

`fasp_dcsrvec_read2`
 `BlalO.c`, 409

`fasp_dcsrvec_write1`
 `BlalO.c`, 410

`fasp_dcsrvec_write2`
 `BlalO.c`, 411

`fasp_dmtx_read`
 `BlalO.c`, 412

`fasp_dmtxsym_read`
 `BlalO.c`, 412

`fasp_dstr_alloc`
 `BlaSparseSTR.c`, 596

`fasp_dstr_cp`
 `BlaSparseSTR.c`, 596

`fasp_dstr_create`
 `BlaSparseSTR.c`, 597

`fasp_dstr_free`
 `BlaSparseSTR.c`, 597

`fasp_dstr_print`
 `BlalO.c`, 413

`fasp_dstr_read`
 `BlalO.c`, 413

`fasp_dstr_write`
 `BlalO.c`, 414

`fasp_dvec_alloc`
 `AuxVector.c`, 282

`fasp_dvec_cp`
 `AuxVector.c`, 282

`fasp_dvec_create`
 `AuxVector.c`, 282

`fasp_dvec_free`
 `AuxVector.c`, 283

`fasp_dvec_isnan`
 `AuxVector.c`, 283

`fasp_dvec_maxdiff`
 `AuxVector.c`, 284

`fasp_dvec_print`
 `BlalO.c`, 414

`fasp_dvec_rand`
 `AuxVector.c`, 284

`fasp_dvec_read`
 `BlalO.c`, 415

`fasp_dvec_set`
 `AuxVector.c`, 285

`fasp_dvec_symdiagscale`
 `AuxVector.c`, 285

`fasp_dvec_write`
 `BlalO.c`, 415

`fasp_dvecind_read`
 `BlalO.c`, 416

`fasp_dvecind_write`
 `BlalO.c`, 416

`fasp_famg_solve`
 `PreMGSolve.c`, 1277

`fasp_format_dblc_dcsr`
 `BlaFormat.c`, 310

`fasp_format_dbsr_dcoo`
 `BlaFormat.c`, 310

`fasp_format_dbsr_dcsr`
 `BlaFormat.c`, 310

`fasp_format_dcoo_dcsr`
 `BlaFormat.c`, 311

`fasp_format_dcsr_dbsr`
 `BlaFormat.c`, 311

`fasp_format_dcsr_dcoo`
 `BlaFormat.c`, 312

`fasp_format_dcsr_dcsr`
 `BlaFormat.c`, 312

`fasp_format_dstr_dbsr`
 `BlaFormat.c`, 313

`fasp_format_dstr_dcsr`
 `BlaFormat.c`, 313

`fasp_fwrapper_dbsr_krylov_amg_`
 `SolWrapper.c`, 1382

`fasp_fwrapper_dbsr_krylov_ilu_`
 `SolWrapper.c`, 1383

`fasp_fwrapper_dcsr_amg_`
 `SolWrapper.c`, 1383

`fasp_fwrapper_dcsr_krylov_amg_`
 `SolWrapper.c`, 1384

`fasp_fwrapper_dcsr_krylov_ilu_`
 `SolWrapper.c`, 1385

`fasp_fwrapper_dcsr_pardiso_`
 `SolWrapper.c`, 1386

`fasp_generate_diaginv_block`
 `ltrSmootherSTR.c`, 824

`fasp_get_start_end`
 `AuxThreads.c`, 276

`fasp_gettime`
 `AuxTiming.c`, 280

`fasp_grid.h`, 202, 203
 `__FASPGRID_HEADER__`, 202
 `grid2d`, 203
 `pcgrid2d`, 203
 `pgrid2d`, 203

`fasp_grid2d_plot`
 `AuxGraphics.c`, 216

`fasp_hb_read`
 `BlalO.c`, 417

`fasp_iarray_cp`
 `AuxArray.c`, 205

`fasp_iarray_set`
 `AuxArray.c`, 205

`fasp_icsr_cp`
 `BlaSparseCSR.c`, 569

`fasp_icsr_create`
 `BlaSparseCSR.c`, 570

`fasp_icsr_free`
 `BlaSparseCSR.c`, 570

`fasp_icsr_trans`
 `BlaSparseCSR.c`, 571

`fasp_ilu_data_create`
 `PreDataInit.c`, 1242

`fasp_ilu_data_free`
 `PreDataInit.c`, 1242

`fasp_ilu_dbsr_setup`
 `BlalLUSetupBSR.c`, 354

`fasp_ilu_dbsr_setup_levsch_omp`
 `BlalLUSetupBSR.c`, 354

`fasp_ilu_dbsr_setup_levsch_step`
 `BlalLUSetupBSR.c`, 355

`fasp_ilu_dbsr_setup_mc_omp`
 `BlalLUSetupBSR.c`, 355

`fasp_ilu_dbsr_setup_omp`
 `BlalLUSetupBSR.c`, 356

`fasp_ilu_dbsr_setup_step`
 `BlalLUSetupBSR.c`, 357

`fasp_ilu_dcsr_setup`
 `BlalLUSetupCSR.c`, 380

`fasp_ilu_dstr_setup0`
 `BlalLUSetupSTR.c`, 383

`fasp_ilu_dstr_setup1`
 `BlalLUSetupSTR.c`, 383

`fasp_iluk`
 `BlalLU.c`, 326

`fasp_ilut`
 `BlalLU.c`, 326

`fasp_ilutp`
 `BlalLU.c`, 327

`fasp_itinfo`
 `AuxMessage.c`, 244

`fasp_ivec_alloc`
 `AuxVector.c`, 286

`fasp_ivec_create`
 `AuxVector.c`, 286

`fasp_ivec_free`
 `AuxVector.c`, 287

`fasp_ivec_print`
 `BlalO.c`, 417

`fasp_ivec_read`
 `BlalO.c`, 418

`fasp_ivec_set`
 `AuxVector.c`, 287

`fasp_ivec_write`
 `BlalO.c`, 418

`fasp_ivecind_read`
 `BlalO.c`, 419

`fasp_matrix_read`
 `BlalO.c`, 419

`fasp_matrix_read_bin`
 `BlalO.c`, 420

`fasp_matrix_write`
 `BlalO.c`, 421

`fasp_mem_calloc`
 `AuxMemory.c`, 237

`fasp_mem_free`
 `AuxMemory.c`, 237

`fasp_mem_iludata_check`
 `AuxMemory.c`, 237

`fasp_mem_realloc`

AuxMemory.c, [238](#)
fasp_mem_usage
 AuxMemory.c, [238](#)
fasp_message
 AuxMessage.c, [245](#)
fasp_param_amg_init
 AuxParam.c, [249](#)
fasp_param_amg_print
 AuxParam.c, [250](#)
fasp_param_amg_set
 AuxParam.c, [250](#)
fasp_param_amg_to_prec
 AuxParam.c, [251](#)
fasp_param_amg_to_precbsr
 AuxParam.c, [251](#)
fasp_param_check
 AuxInput.c, [224](#)
fasp_param_ilu_init
 AuxParam.c, [251](#)
fasp_param_ilu_print
 AuxParam.c, [252](#)
fasp_param_ilu_set
 AuxParam.c, [252](#)
fasp_param_init
 AuxParam.c, [253](#)
fasp_param_input
 AuxInput.c, [224](#)
fasp_param_input_init
 AuxParam.c, [253](#)
fasp_param_prec_to_amg
 AuxParam.c, [254](#)
fasp_param_precbsr_to_amg
 AuxParam.c, [254](#)
fasp_param_set
 AuxParam.c, [254](#)
fasp_param_solver_init
 AuxParam.c, [255](#)
fasp_param_solver_print
 AuxParam.c, [255](#)
fasp_param_solver_set
 AuxParam.c, [256](#)
fasp_param_swz_init
 AuxParam.c, [256](#)
fasp_param_swz_print
 AuxParam.c, [256](#)
fasp_param_swz_set
 AuxParam.c, [257](#)
fasp_poisson_fgm1d
 SolGMGPoisson.c, [1352](#)
fasp_poisson_fgm2d
 SolGMGPoisson.c, [1352](#)
fasp_poisson_fgm3d
 SolGMGPoisson.c, [1353](#)
fasp_poisson_gmg1d
 SolGMGPoisson.c, [1354](#)
fasp_poisson_gmg2d
 SolGMGPoisson.c, [1354](#)
fasp_poisson_gmg3d
 SolGMGPoisson.c, [1355](#)
fasp_poisson_gmgcg1d
 SolGMGPoisson.c, [1356](#)
fasp_poisson_gmgcg2d
 SolGMGPoisson.c, [1356](#)
fasp_poisson_gmgcg3d
 SolGMGPoisson.c, [1357](#)
fasp_precond_amg
 PreCSR.c, [1228](#)
fasp_precond_amg_nk
 PreCSR.c, [1228](#)
fasp_precond_amli
 PreCSR.c, [1229](#)
fasp_precond_data_init
 PreDataInit.c, [1242](#)
fasp_precond_dblc_diag_3
 PreBLC.c, [1187](#)
fasp_precond_dblc_diag_3_amg
 PreBLC.c, [1187](#)
fasp_precond_dblc_diag_4
 PreBLC.c, [1188](#)
fasp_precond_dblc_lower_3
 PreBLC.c, [1188](#)
fasp_precond_dblc_lower_3_amg
 PreBLC.c, [1190](#)
fasp_precond_dblc_lower_4
 PreBLC.c, [1190](#)
fasp_precond_dblc_SGS_3
 PreBLC.c, [1192](#)
fasp_precond_dblc_SGS_3_amg
 PreBLC.c, [1192](#)
fasp_precond_dblc_sweeping
 PreBLC.c, [1194](#)
fasp_precond_dblc_upper_3
 PreBLC.c, [1194](#)
fasp_precond_dblc_upper_3_amg
 PreBLC.c, [1196](#)
fasp_precond_dbsr_amg
 PreBSR.c, [1209](#)
fasp_precond_dbsr_amg_nk
 PreBSR.c, [1209](#)
fasp_precond_dbsr_diag
 PreBSR.c, [1210](#)
fasp_precond_dbsr_diag_nc2
 PreBSR.c, [1210](#)
fasp_precond_dbsr_diag_nc3
 PreBSR.c, [1211](#)
fasp_precond_dbsr_diag_nc5
 PreBSR.c, [1211](#)
fasp_precond_dbsr_diag_nc7

PreBSR.c, [1212](#)
 fasp_precond_dbsr_ilu
 PreBSR.c, [1212](#)
 fasp_precond_dbsr_ilu_ls_omp
 PreBSR.c, [1213](#)
 fasp_precond_dbsr_ilu_mc_omp
 PreBSR.c, [1213](#)
 fasp_precond_dbsr_namli
 PreBSR.c, [1214](#)
 fasp_precond_diag
 PreCSR.c, [1229](#)
 fasp_precond_dstr_blockgs
 PreSTR.c, [1282](#)
 fasp_precond_dstr_diag
 PreSTR.c, [1282](#)
 fasp_precond_dstr_ilu0
 PreSTR.c, [1283](#)
 fasp_precond_dstr_ilu0_backward
 PreSTR.c, [1283](#)
 fasp_precond_dstr_ilu0_forward
 PreSTR.c, [1283](#)
 fasp_precond_dstr_ilu1
 PreSTR.c, [1284](#)
 fasp_precond_dstr_ilu1_backward
 PreSTR.c, [1284](#)
 fasp_precond_dstr_ilu1_forward
 PreSTR.c, [1285](#)
 fasp_precond_famg
 PreCSR.c, [1230](#)
 fasp_precond_ilu
 PreCSR.c, [1230](#)
 fasp_precond_ilu_backward
 PreCSR.c, [1231](#)
 fasp_precond_ilu_forward
 PreCSR.c, [1231](#)
 fasp_precond_namli
 PreCSR.c, [1231](#)
 fasp_precond_setup
 PreCSR.c, [1232](#)
 fasp_precond_swz
 PreCSR.c, [1232](#)
 fasp_set_gs_threads
 AuxThreads.c, [277](#)
 fasp_smat_identity
 BlaSmallMatInv.c, [486](#)
 fasp_smat_identity_nc2
 BlaSmallMatInv.c, [486](#)
 fasp_smat_identity_nc3
 BlaSmallMatInv.c, [487](#)
 fasp_smat_identity_nc5
 BlaSmallMatInv.c, [487](#)
 fasp_smat_identity_nc7
 BlaSmallMatInv.c, [488](#)
 fasp_smat_inv
 BlaSmallMatInv.c, [488](#)
 fasp_smat_inv_nc
 BlaSmallMatInv.c, [488](#)
 fasp_smat_inv_nc2
 BlaSmallMatInv.c, [489](#)
 fasp_smat_inv_nc3
 BlaSmallMatInv.c, [489](#)
 fasp_smat_inv_nc4
 BlaSmallMatInv.c, [490](#)
 fasp_smat_inv_nc5
 BlaSmallMatInv.c, [490](#)
 fasp_smat_inv_nc7
 BlaSmallMatInv.c, [490](#)
 fasp_smat_invp_nc
 BlaSmallMatInv.c, [491](#)
 fasp_smat_Linf
 BlaSmallMatInv.c, [491](#)
 fasp_smat_lu_decomp
 BlaSmallMatLU.c, [502](#)
 fasp_smat_lu_solve
 BlaSmallMatLU.c, [502](#)
 fasp_smoother_dbsr_gs
 ltrSmootherBSR.c, [762](#)
 fasp_smoother_dbsr_gs1
 ltrSmootherBSR.c, [763](#)
 fasp_smoother_dbsr_gs_ascend
 ltrSmootherBSR.c, [764](#)
 fasp_smoother_dbsr_gs_ascend1
 ltrSmootherBSR.c, [764](#)
 fasp_smoother_dbsr_gs_descend
 ltrSmootherBSR.c, [765](#)
 fasp_smoother_dbsr_gs_descend1
 ltrSmootherBSR.c, [765](#)
 fasp_smoother_dbsr_gs_order1
 ltrSmootherBSR.c, [766](#)
 fasp_smoother_dbsr_gs_order2
 ltrSmootherBSR.c, [766](#)
 fasp_smoother_dbsr_ilu
 ltrSmootherBSR.c, [767](#)
 fasp_smoother_dbsr_jacobi
 ltrSmootherBSR.c, [767](#)
 fasp_smoother_dbsr_jacobi1
 ltrSmootherBSR.c, [768](#)
 fasp_smoother_dbsr_jacobi_setup
 ltrSmootherBSR.c, [768](#)
 fasp_smoother_dbsr_sor
 ltrSmootherBSR.c, [769](#)
 fasp_smoother_dbsr_sor1
 ltrSmootherBSR.c, [770](#)
 fasp_smoother_dbsr_sor_ascend
 ltrSmootherBSR.c, [770](#)
 fasp_smoother_dbsr_sor_descend
 ltrSmootherBSR.c, [771](#)
 fasp_smoother_dbsr_sor_order

ltrSmootherBSR.c, [771](#)
 fasp_smoother_dcsr_gs
 ltrSmootherCSR.c, [790](#)
 fasp_smoother_dcsr_gs_cf
 ltrSmootherCSR.c, [791](#)
 fasp_smoother_dcsr_gs_multicolor
 BlaSparseCSR.c, [571](#)
 fasp_smoother_dcsr_gscr
 ltrSmootherCSRcr.c, [813](#)
 fasp_smoother_dcsr_ilu
 ltrSmootherCSR.c, [792](#)
 fasp_smoother_dcsr_jacobi
 ltrSmootherCSR.c, [792](#)
 fasp_smoother_dcsr_kaczmarz
 ltrSmootherCSR.c, [793](#)
 fasp_smoother_dcsr_L1diag
 ltrSmootherCSR.c, [794](#)
 fasp_smoother_dcsr_poly
 ltrSmootherCSRpoly.c, [816](#)
 fasp_smoother_dcsr_poly_old
 ltrSmootherCSRpoly.c, [816](#)
 fasp_smoother_dcsr_sgs
 ltrSmootherCSR.c, [794](#)
 fasp_smoother_dcsr_sor
 ltrSmootherCSR.c, [795](#)
 fasp_smoother_dcsr_sor_cf
 ltrSmootherCSR.c, [796](#)
 fasp_smoother_dstr_gs
 ltrSmootherSTR.c, [825](#)
 fasp_smoother_dstr_gs1
 ltrSmootherSTR.c, [825](#)
 fasp_smoother_dstr_gs_ascend
 ltrSmootherSTR.c, [826](#)
 fasp_smoother_dstr_gs_cf
 ltrSmootherSTR.c, [826](#)
 fasp_smoother_dstr_gs_descend
 ltrSmootherSTR.c, [827](#)
 fasp_smoother_dstr_gs_order
 ltrSmootherSTR.c, [828](#)
 fasp_smoother_dstr_jacobi
 ltrSmootherSTR.c, [828](#)
 fasp_smoother_dstr_jacobi1
 ltrSmootherSTR.c, [829](#)
 fasp_smoother_dstr_sor
 ltrSmootherSTR.c, [829](#)
 fasp_smoother_dstr_sor1
 ltrSmootherSTR.c, [830](#)
 fasp_smoother_dstr_sor_ascend
 ltrSmootherSTR.c, [830](#)
 fasp_smoother_dstr_sor_cf
 ltrSmootherSTR.c, [831](#)
 fasp_smoother_dstr_sor_descend
 ltrSmootherSTR.c, [831](#)
 fasp_smoother_dstr_sor_order
 ltrSmootherSTR.c, [832](#)
 fasp_smoother_dstr_swz
 ltrSmootherSTR.c, [833](#)
 fasp_solver_amg
 SolAMG.c, [1307](#)
 fasp_solver_amli
 PreMGRecurAMLI.c, [1264](#)
 fasp_solver_dblc_itsolver
 SolBLC.c, [1310](#)
 fasp_solver_dblc_krylov
 SolBLC.c, [1311](#)
 fasp_solver_dblc_krylov_block3
 SolBLC.c, [1311](#)
 fasp_solver_dblc_krylov_block4
 SolBLC.c, [1312](#)
 fasp_solver_dblc_krylov_sweeping
 SolBLC.c, [1313](#)
 fasp_solver_dblc_pbcgs
 KryPbcgs.c, [853](#)
 fasp_solver_dblc_pcg
 KryPcg.c, [878](#)
 fasp_solver_dblc_pgcr
 KryPgcr.c, [905](#)
 fasp_solver_dblc_pgmres
 KryPgmres.c, [912](#)
 fasp_solver_dblc_pminres
 KryPminres.c, [936](#)
 fasp_solver_dblc_pvfgmres
 KryPvfgmres.c, [959](#)
 fasp_solver_dblc_pvgmres
 KryPvgmres.c, [978](#)
 fasp_solver_dblc_spbcgs
 KrySPbcgs.c, [1003](#)
 fasp_solver_dblc_spcg
 KrySPcg.c, [1025](#)
 fasp_solver_dblc_spgmres
 KrySPgmres.c, [1040](#)
 fasp_solver_dblc_spmminres
 KrySPminres.c, [1060](#)
 fasp_solver_dblc_spvgmres
 KrySPvgmres.c, [1080](#)
 fasp_solver_dbsr_itsolver
 SolBSR.c, [1320](#)
 fasp_solver_dbsr_krylov
 SolBSR.c, [1321](#)
 fasp_solver_dbsr_krylov_amg
 SolBSR.c, [1321](#)
 fasp_solver_dbsr_krylov_amg_nk
 SolBSR.c, [1322](#)
 fasp_solver_dbsr_krylov_diag
 SolBSR.c, [1323](#)
 fasp_solver_dbsr_krylov_ilu
 SolBSR.c, [1323](#)
 fasp_solver_dbsr_krylov_nk_amg

SolBSR.c, [1324](#)
fasp_solver_dbsr_pbcgs
 KryPbcgs.c, [854](#)
fasp_solver_dbsr_pcg
 KryPcg.c, [879](#)
fasp_solver_dbsr_pgmres
 KryPgmres.c, [913](#)
fasp_solver_dbsr_pvfgmres
 KryPvfgmres.c, [960](#)
fasp_solver_dbsr_pvgmres
 KryPvgmres.c, [979](#)
fasp_solver_dbsr_spbcs
 KrySPbcgs.c, [1003](#)
fasp_solver_dbsr_spgmres
 KrySPgmres.c, [1041](#)
fasp_solver_dbsr_spvgmres
 KrySPvgmres.c, [1080](#)
fasp_solver_dcsr_itsolver
 SolCSR.c, [1334](#)
fasp_solver_dcsr_itsolver_s
 SolCSR.c, [1334](#)
fasp_solver_dcsr_krylov
 SolCSR.c, [1335](#)
fasp_solver_dcsr_krylov_amg
 SolCSR.c, [1336](#)
fasp_solver_dcsr_krylov_amg_nk
 SolCSR.c, [1336](#)
fasp_solver_dcsr_krylov_diag
 SolCSR.c, [1337](#)
fasp_solver_dcsr_krylov_ilu
 SolCSR.c, [1337](#)
fasp_solver_dcsr_krylov_ilu_M
 SolCSR.c, [1338](#)
fasp_solver_dcsr_krylov_s
 SolCSR.c, [1339](#)
fasp_solver_dcsr_krylov_swz
 SolCSR.c, [1339](#)
fasp_solver_dcsr_pbcgs
 KryPbcgs.c, [855](#)
fasp_solver_dcsr_pcg
 KryPcg.c, [879](#)
fasp_solver_dcsr_pgcg
 KryPgcg.c, [899](#)
fasp_solver_dcsr_pgr
 KryPgr.c, [906](#)
fasp_solver_dcsr_pgmres
 KryPgmres.c, [914](#)
fasp_solver_dcsr_pminres
 KryPminres.c, [937](#)
fasp_solver_dcsr_pvfgmres
 KryPvfgmres.c, [961](#)
fasp_solver_dcsr_pvgmres
 KryPvgmres.c, [980](#)
fasp_solver_dcsr_spbcs
 KrySPbcgs.c, [1004](#)
fasp_solver_dcsr_spcg
 KrySPcg.c, [1026](#)
fasp_solver_dcsr_spgmres
 KrySPgmres.c, [1042](#)
fasp_solver_dcsr_spmminres
 KrySPminres.c, [1061](#)
fasp_solver_dcsr_spvgmres
 KrySPvgmres.c, [1081](#)
fasp_solver_dstr_itsolver
 SolSTR.c, [1374](#)
fasp_solver_dstr_krylov
 SolSTR.c, [1375](#)
fasp_solver_dstr_krylov_blockgs
 SolSTR.c, [1375](#)
fasp_solver_dstr_krylov_diag
 SolSTR.c, [1376](#)
fasp_solver_dstr_krylov_ilu
 SolSTR.c, [1376](#)
fasp_solver_dstr_pbcgs
 KryPbcgs.c, [856](#)
fasp_solver_dstr_pcg
 KryPcg.c, [880](#)
fasp_solver_dstr_pgmres
 KryPgmres.c, [915](#)
fasp_solver_dstr_pminres
 KryPminres.c, [937](#)
fasp_solver_dstr_pvgmres
 KryPvgmres.c, [981](#)
fasp_solver_dstr_spbcs
 KrySPbcgs.c, [1005](#)
fasp_solver_dstr_spcg
 KrySPcg.c, [1027](#)
fasp_solver_dstr_spgmres
 KrySPgmres.c, [1042](#)
fasp_solver_dstr_spmminres
 KrySPminres.c, [1062](#)
fasp_solver_dstr_spvgmres
 KrySPvgmres.c, [1082](#)
fasp_solver_famg
 SolFAMG.c, [1349](#)
fasp_solver_fmecycle
 PreMGCycleFull.c, [1255](#)
fasp_solver_itsolver
 SolMatFree.c, [1369](#)
fasp_solver_krylov
 SolMatFree.c, [1370](#)
fasp_solver_matfree_init
 SolMatFree.c, [1370](#)
fasp_solver_mgcycle
 PreMGCycle.c, [1247](#)
fasp_solver_mgcycle_bsr
 PreMGCycle.c, [1248](#)
fasp_solver_mgrecur

- PreMGRecur.c, [1260](#)
- fasp_solver_mumps
 - XtrMumps.c, [136](#)
- fasp_solver_mumps_steps
 - XtrMumps.c, [137](#)
- fasp_solver_namli
 - PreMGRecurAMLI.c, [1265](#)
- fasp_solver_namli_bsr
 - PreMGRecurAMLI.c, [1265](#)
- fasp_solver_pardiso
 - XtrPardiso.c, [144](#)
- fasp_solver_pbcgs
 - KryPbcgs.c, [856](#)
- fasp_solver_pcg
 - KryPcg.c, [881](#)
- fasp_solver_pgcg
 - KryPgcg.c, [900](#)
- fasp_solver_pgmres
 - KryPgmres.c, [916](#)
- fasp_solver_pminres
 - KryPminres.c, [938](#)
- fasp_solver_pvfgmres
 - KryPvfgmres.c, [962](#)
- fasp_solver_pvgmres
 - KryPvgmres.c, [981](#)
- fasp_solver_superlu
 - XtrSuperlu.c, [151](#)
- fasp_solver_umfpack
 - XtrUmfpack.c, [153](#)
- fasp_sparse_aat_
 - BlaSparseUtil.c, [601](#)
- fasp_sparse_abyb_
 - BlaSparseUtil.c, [601](#)
- fasp_sparse_abybms_
 - BlaSparseUtil.c, [602](#)
- fasp_sparse_aplbms_
 - BlaSparseUtil.c, [602](#)
- fasp_sparse_aplusb_
 - BlaSparseUtil.c, [603](#)
- fasp_sparse_iit_
 - BlaSparseUtil.c, [604](#)
- fasp_sparse_mis
 - BlaSparseUtil.c, [604](#)
- fasp_sparse_rapcmp_
 - BlaSparseUtil.c, [605](#)
- fasp_sparse_rapms_
 - BlaSparseUtil.c, [606](#)
- fasp_sparse_wta_
 - BlaSparseUtil.c, [606](#)
- fasp_sparse_wtams_
 - BlaSparseUtil.c, [607](#)
- fasp_sparse_ytx_
 - BlaSparseUtil.c, [608](#)
- fasp_sparse_ytxbig_
 - BlaSparseUtil.c, [608](#)
- FASP_SUCCESS
 - fasp_const.h, [186](#)
- fasp_swz_data_free
 - PreDataInit.c, [1243](#)
- fasp_swz_dcsr_setup
 - BlaSchwarzSetup.c, [450](#)
- fasp_symbfactor
 - BlalLU.c, [329](#)
- fasp_vector_read
 - BlalO.c, [421](#)
- fasp_vector_write
 - BlalO.c, [423](#)
- FASP_VERSION
 - fasp.h, [159](#)
- fct
 - mxv_matfree, [96](#)
 - precond, [98](#)
- FGPT
 - fasp_const.h, [186](#)
- FPFIRST
 - fasp_const.h, [187](#)
- FPNA_RATIO
 - fasp_const.h, [187](#)
- G0PT
 - fasp_const.h, [187](#)
- GE
 - fasp.h, [159](#)
- grid2d, [61](#)
 - e, [62](#)
 - edges, [62](#)
 - ediri, [62](#)
 - efather, [62](#)
 - fasp_grid.h, [203](#)
 - p, [63](#)
 - pdiri, [63](#)
 - pfather, [63](#)
 - s, [63](#)
 - t, [63](#)
 - tfather, [64](#)
 - triangles, [64](#)
 - vertices, [64](#)
- GT
 - fasp.h, [159](#)
- I
 - iCOOmat, [67](#)
- IA
 - dBSRmat, [46](#)
 - dCSRmat, [54](#)
 - iCSRmat, [69](#)
- iBLCmat, [64](#)
 - bcol, [65](#)
 - blocks, [65](#)

- brow, 65
- fasp_block.h, 174
- iblock
 - SWZ_data, 129
- ic
 - AMG_data, 19
 - ILU_data, 73
- icmap
 - AMG_data, 19
 - ILU_data, 73
- ICNTL
 - XtrMumps.c, 136
- iCOOmat, 66
 - col, 67
 - fasp.h, 163
 - I, 67
 - J, 67
 - nnz, 67
 - row, 67
 - val, 68
- iCSRmat, 68
 - col, 69
 - fasp.h, 163
 - IA, 69
 - JA, 69
 - nnz, 69
 - row, 70
 - val, 70
- idenmat, 70
 - col, 71
 - fasp.h, 163
 - row, 71
 - val, 71
- ijlu
 - ILU_data, 73
- ilength
 - BlalO.c, 424
- ilevL
 - ILU_data, 74
- ilevU
 - ILU_data, 74
- ILU_data, 72
 - A, 73
 - col, 73
 - ic, 73
 - icmap, 73
 - ijlu, 73
 - ilevL, 74
 - ilevU, 74
 - iperm, 74
 - jlevL, 74
 - jlevU, 74
 - luval, 75
 - nb, 75
 - ncolors, 75
 - nlevL, 75
 - nlevU, 75
 - nwork, 76
 - nzlu, 76
 - row, 76
 - type, 76
 - uptr, 76
 - work, 77
- ILU_droptol
 - AMG_param, 35
 - ILU_param, 78
 - input_param, 87
- ILU_levels
 - AMG_data, 20
 - AMG_data_bsr, 26
 - AMG_param, 35
- ILU_lfil
 - AMG_param, 35
 - ILU_param, 78
 - input_param, 88
- ILU_param, 77
 - ILU_droptol, 78
 - ILU_lfil, 78
 - ILU_permtol, 78
 - ILU_relax, 78
 - ILU_type, 78
 - print_level, 79
- ILU_permtol
 - AMG_param, 36
 - ILU_param, 78
 - input_param, 88
- ILU_relax
 - AMG_param, 36
 - ILU_param, 78
 - input_param, 88
- ilu_solve_time
 - ltrSmootherBSR.c, 772
- ILU_type
 - AMG_param, 36
 - ILU_param, 78
 - input_param, 88
- ILUk
 - fasp_const.h, 187
- ILUt
 - fasp_const.h, 187
- ILUtp
 - fasp_const.h, 187
- index
 - dCSRLmat, 52
- inifile
 - input_param, 88
- input_param, 79
 - AMG_aggregation_type, 81

AMG_aggressive_level, 81
 AMG_aggressive_path, 81
 AMG_amli_degree, 81
 AMG_coarse_dof, 81
 AMG_coarse_scaling, 82
 AMG_coarse_solver, 82
 AMG_coarsening_type, 82
 AMG_cycle_type, 82
 AMG_ILU_levels, 82
 AMG_interpolation_type, 83
 AMG_levels, 83
 AMG_max_aggregation, 83
 AMG_max_row_sum, 83
 AMG_maxit, 83
 AMG_nl_amli_krylov_type, 84
 AMG_pair_number, 84
 AMG_polynomial_degree, 84
 AMG_postsmooth_iter, 84
 AMG_presmooth_iter, 84
 AMG_quality_bound, 85
 AMG_relaxation, 85
 AMG_smooth_filter, 85
 AMG_smooth_order, 85
 AMG_smooth_restriction, 85
 AMG_smoother, 86
 AMG_strong_coupled, 86
 AMG_strong_threshold, 86
 AMG_SWZ_levels, 86
 AMG_tentative_smooth, 86
 AMG_tol, 87
 AMG_truncation_threshold, 87
 AMG_type, 87
 decoup_type, 87
 ILU_droptol, 87
 ILU_lfil, 88
 ILU_permtol, 88
 ILU_relax, 88
 ILU_type, 88
 inifile, 88
 itsolver_maxit, 89
 itsolver_tol, 89
 output_type, 89
 precondition_type, 89
 print_level, 89
 problem_num, 90
 restart, 90
 solver_type, 90
 stop_type, 90
 SWZ_blksolver, 90
 SWZ_maxlvl, 91
 SWZ_mmsize, 91
 SWZ_type, 91
 workdir, 91

INT

fasp.h, 159
 INTERP_DIR
 fasp_const.h, 187
 INTERP_ENG
 fasp_const.h, 187
 INTERP_EXT
 fasp_const.h, 188
 INTERP_STD
 fasp_const.h, 188
 interpolation_type
 AMG_param, 36
 iperm
 ILU_data, 74
 ISNAN
 fasp.h, 160
 ISPT
 fasp_const.h, 188
 ltrSmootherBSR.c, 761, 772
 fasp_smoother_dbsr_gs, 762
 fasp_smoother_dbsr_gs1, 763
 fasp_smoother_dbsr_gs_ascend, 764
 fasp_smoother_dbsr_gs_ascend1, 764
 fasp_smoother_dbsr_gs_descend, 765
 fasp_smoother_dbsr_gs_descend1, 765
 fasp_smoother_dbsr_gs_order1, 766
 fasp_smoother_dbsr_gs_order2, 766
 fasp_smoother_dbsr_ilu, 767
 fasp_smoother_dbsr_jacobi, 767
 fasp_smoother_dbsr_jacobi1, 768
 fasp_smoother_dbsr_jacobi_setup, 768
 fasp_smoother_dbsr_sor, 769
 fasp_smoother_dbsr_sor1, 770
 fasp_smoother_dbsr_sor_ascend, 770
 fasp_smoother_dbsr_sor_descend, 771
 fasp_smoother_dbsr_sor_order, 771
 ilu_solve_time, 772
 ltrSmootherCSR.c, 789, 796
 fasp_smoother_dcsr_gs, 790
 fasp_smoother_dcsr_gs_cf, 791
 fasp_smoother_dcsr_ilu, 792
 fasp_smoother_dcsr_jacobi, 792
 fasp_smoother_dcsr_kaczmarz, 793
 fasp_smoother_dcsr_L1diag, 794
 fasp_smoother_dcsr_sgs, 794
 fasp_smoother_dcsr_sor, 795
 fasp_smoother_dcsr_sor_cf, 796
 ltrSmootherCSRcr.c, 813, 814
 fasp_smoother_dcsr_gscr, 813
 ltrSmootherCSRpoly.c, 815, 817
 fasp_smoother_dcsr_poly, 816
 fasp_smoother_dcsr_poly_old, 816
 ltrSmootherSTR.c, 823, 833
 fasp_generate_diaginv_block, 824
 fasp_smoother_dstr_gs, 825

fasp_smoother_dstr_gs1, [825](#)
 fasp_smoother_dstr_gs_ascend, [826](#)
 fasp_smoother_dstr_gs_cf, [826](#)
 fasp_smoother_dstr_gs_descend, [827](#)
 fasp_smoother_dstr_gs_order, [828](#)
 fasp_smoother_dstr_jacobi, [828](#)
 fasp_smoother_dstr_jacobi1, [829](#)
 fasp_smoother_dstr_sor, [829](#)
 fasp_smoother_dstr_sor1, [830](#)
 fasp_smoother_dstr_sor_ascend, [830](#)
 fasp_smoother_dstr_sor_cf, [831](#)
 fasp_smoother_dstr_sor_descend, [831](#)
 fasp_smoother_dstr_sor_order, [832](#)
 fasp_smoother_dstr_swz, [833](#)
 ITS_param, [92](#)
 decoup_type, [92](#)
 itsolver_type, [92](#)
 maxit, [92](#)
 precond_type, [93](#)
 print_level, [93](#)
 restart, [93](#)
 stop_type, [93](#)
 tol, [93](#)
 itsolver_maxit
 input_param, [89](#)
 itsolver_tol
 input_param, [89](#)
 itsolver_type
 ITS_param, [92](#)
 ivector, [94](#)
 fasp.h, [163](#)
 row, [94](#)
 val, [94](#)
 J
 iCOOmat, [67](#)
 JA
 dBSRmat, [47](#)
 dCSRmat, [55](#)
 iCSRmat, [69](#)
 ja
 dCSRLmat, [52](#)
 jblock
 SWZ_data, [129](#)
 jlevL
 ILU_data, [74](#)
 jlevU
 ILU_data, [74](#)
 job
 Mumps_data, [95](#)
 KryPbcgs.c, [852](#), [857](#)
 fasp_solver_dblc_pbcgs, [853](#)
 fasp_solver_dbsr_pbcgs, [854](#)
 fasp_solver_dcsr_pbcgs, [855](#)
 fasp_solver_dstr_pbcgs, [856](#)
 KryPcg.c, [876](#), [882](#)
 fasp_solver_dblc_pcg, [878](#)
 fasp_solver_dbsr_pcg, [879](#)
 fasp_solver_dcsr_pcg, [879](#)
 fasp_solver_dstr_pcg, [880](#)
 fasp_solver_pcg, [881](#)
 KryPgcg.c, [899](#), [901](#)
 fasp_solver_dcsr_pgcg, [899](#)
 fasp_solver_pgcg, [900](#)
 KryPgcr.c, [904](#), [907](#)
 fasp_solver_dblc_pgcr, [905](#)
 fasp_solver_dcsr_pgcr, [906](#)
 KryPgmres.c, [911](#), [916](#)
 fasp_solver_dblc_pgmres, [912](#)
 fasp_solver_dbsr_pgmres, [913](#)
 fasp_solver_dcsr_pgmres, [914](#)
 fasp_solver_dstr_pgmres, [915](#)
 fasp_solver_pgmres, [916](#)
 KryPminres.c, [935](#), [939](#)
 fasp_solver_dblc_pminres, [936](#)
 fasp_solver_dcsr_pminres, [937](#)
 fasp_solver_dstr_pminres, [937](#)
 fasp_solver_pminres, [938](#)
 KryPvfgmres.c, [958](#), [963](#)
 fasp_solver_dblc_pvfgmres, [959](#)
 fasp_solver_dbsr_pvfgmres, [960](#)
 fasp_solver_dcsr_pvfgmres, [961](#)
 fasp_solver_pvfgmres, [962](#)
 KryPvgmres.c, [977](#), [982](#)
 fasp_solver_dblc_pvgmres, [978](#)
 fasp_solver_dbsr_pvgmres, [979](#)
 fasp_solver_dcsr_pvgmres, [980](#)
 fasp_solver_dstr_pvgmres, [981](#)
 fasp_solver_pvgmres, [981](#)
 KrySPbcgs.c, [1002](#), [1006](#)
 fasp_solver_dblc_spbcgs, [1003](#)
 fasp_solver_dbsr_spbcgs, [1003](#)
 fasp_solver_dcsr_spbcgs, [1004](#)
 fasp_solver_dstr_spbcgs, [1005](#)
 KrySPcg.c, [1024](#), [1027](#)
 fasp_solver_dblc_spcg, [1025](#)
 fasp_solver_dcsr_spcg, [1026](#)
 fasp_solver_dstr_spcg, [1027](#)
 KrySPgmres.c, [1039](#), [1043](#)
 fasp_solver_dblc_spgmres, [1040](#)
 fasp_solver_dbsr_spgmres, [1041](#)
 fasp_solver_dcsr_spgmres, [1042](#)
 fasp_solver_dstr_spgmres, [1042](#)
 KrySPminres.c, [1059](#), [1062](#)
 fasp_solver_dblc_spminres, [1060](#)
 fasp_solver_dcsr_spminres, [1061](#)
 fasp_solver_dstr_spminres, [1062](#)

KrySPvgmres.c, [1079](#), [1083](#)
 fasp_solver_dblc_spvgmres, [1080](#)
 fasp_solver_dbsr_spvgmres, [1080](#)
 fasp_solver_dcsr_spvgmres, [1081](#)
 fasp_solver_dstr_spvgmres, [1082](#)

LE

 fasp.h, [160](#)

local_A

 precond_data_sweeping, [123](#)

local_index

 precond_data_sweeping, [123](#)

local_LU

 precond_data_sweeping, [123](#)

LONG

 fasp.h, [160](#)

LONGLONG

 fasp.h, [160](#)

LONGREAL

 fasp.h, [160](#)

LS

 fasp.h, [160](#)

LU

 AMG_data, [20](#)

 AMG_data_bsr, [26](#)

 precond_data, [102](#)

 precond_data_bsr, [111](#)

 precond_data_str, [118](#)

LU_diag

 precond_data_blc, [107](#)

luval

 ILU_data, [75](#)

mask

 SWZ_data, [129](#)

MAT_bBSR

 fasp_const.h, [188](#)

MAT_bCSR

 fasp_const.h, [188](#)

MAT_BLC

 fasp_const.h, [188](#)

MAT_BSR

 fasp_const.h, [188](#)

MAT_bSTR

 fasp_const.h, [188](#)

MAT_CSR

 fasp_const.h, [189](#)

MAT_CSRL

 fasp_const.h, [189](#)

MAT_FREE

 fasp_const.h, [189](#)

MAT_STR

 fasp_const.h, [189](#)

MAT_SymCSR

 fasp_const.h, [189](#)

MAX

 fasp.h, [160](#)

max_aggregation

 AMG_param, [36](#)

MAX_AMG_LVL

 fasp_const.h, [189](#)

MAX_CRATE

 fasp_const.h, [189](#)

max_levels

 AMG_data, [20](#)

 AMG_data_bsr, [26](#)

 AMG_param, [37](#)

 precond_data, [102](#)

 precond_data_bsr, [111](#)

 precond_data_str, [118](#)

MAX_REFINE_LVL

 fasp_const.h, [189](#)

MAX_RESTART

 fasp_const.h, [190](#)

max_row_sum

 AMG_param, [37](#)

MAX_STAG

 fasp_const.h, [190](#)

maxa

 SWZ_data, [129](#)

maxbs

 SWZ_data, [129](#)

maxit

 AMG_param, [37](#)

 ITS_param, [92](#)

 precond_data, [102](#)

 precond_data_bsr, [111](#)

 precond_data_str, [118](#)

memt

 SWZ_data, [130](#)

mgl

 precond_data_blc, [107](#)

mgl_data

 precond_data, [102](#)

 precond_data_bsr, [111](#)

 precond_data_str, [118](#)

Million

 AuxMemory.c, [239](#)

MIN

 fasp.h, [161](#)

MIN_CDOF

 fasp_const.h, [190](#)

MIN_CRATE

 fasp_const.h, [190](#)

mulcol_independ_set

 BlalLUSetupBSR.c, [357](#)

MULTI_COLOR_ORDER

 fasp.h, [161](#)

mumps

- AMG_data, 20
 - AMG_data_bsr, 27
 - SWZ_data, 130
- Mumps_data, 95
 - job, 95
- mxv_matfree, 96
 - data, 96
 - fct, 96
- nb
 - dBSRmat, 47
 - ILU_data, 75
 - precond_diag_bsr, 125
- nband
 - dSTRmat, 58
- nblk
 - SWZ_data, 130
- nc
 - dSTRmat, 58
 - precond_diag_str, 126
- ncolors
 - ILU_data, 75
- near_kernel_basis
 - AMG_data, 20
 - AMG_data_bsr, 27
- near_kernel_dim
 - AMG_data, 21
 - AMG_data_bsr, 27
- NEDMALLOC
 - fasp.h, 161
- neigh
 - precond_data_str, 118
- ngrid
 - dSTRmat, 58
- NL_AMLI_CYCLE
 - fasp_const.h, 190
- nl_amli_krylov_type
 - AMG_param, 37
 - precond_data, 103
 - precond_data_bsr, 112
- nlevL
 - ILU_data, 75
- nlevU
 - ILU_data, 75
- NNZ
 - dBSRmat, 47
- nnz
 - dCOOmat, 49
 - dCSRLmat, 52
 - dCSRmat, 55
 - iCOOmat, 67
 - iCSRmat, 69
- NO_ORDER
 - fasp_const.h, 190
- NPAIR
 - fasp_const.h, 190
- num_levels
 - AMG_data, 21
 - AMG_data_bsr, 27
- Numeric
 - AMG_data, 21
 - AMG_data_bsr, 27
- NumLayers
 - precond_data_sweeping, 123
- nwork
 - ILU_data, 76
- nx
 - dSTRmat, 59
- nxy
 - dSTRmat, 59
- ny
 - dSTRmat, 59
- nz
 - dSTRmat, 59
- nz_diff
 - dCSRLmat, 52
- nzlu
 - ILU_data, 76
- OFF
 - fasp_const.h, 190
- offdiag
 - dSTRmat, 59
- offsets
 - dSTRmat, 60
- ON
 - fasp_const.h, 191
- OPENMP_HOLDS
 - fasp_const.h, 191
- order
 - precond_data_str, 119
- output_type
 - input_param, 89
- P
 - AMG_data, 21
 - AMG_data_bsr, 28
- p
 - grid2d, 63
- P_nk
 - AMG_data_bsr, 28
 - precond_data, 103
 - precond_data_bsr, 112
- pair_number
 - AMG_param, 37
- PAIRWISE
 - fasp_const.h, 191
- Pardiso_data, 97
 - pt, 97

pcgrid2d
 fasp_grid.h, 203
 pdata
 AMG_data, 21
 AMG_data_bsr, 28
 pdiri
 grid2d, 63
 pfather
 grid2d, 63
 pgrid2d
 fasp_grid.h, 203
 pivot
 precond_data_str, 119
 pivotS
 precond_data_str, 119
 polynomial_degree
 AMG_param, 38
 precond_data, 103
 postsmooth_iter
 AMG_param, 38
 precond_data, 103
 precond_data_bsr, 112
 precond_data_str, 119
 PP
 AMG_data_bsr, 28
 PP_LU
 AMG_data_bsr, 28
 PreAMGCoarsenCR.c, 1101, 1102
 AMG_COARSEN_CR, 1101
 fasp_amg_coarsening_cr, 1101
 PreAMGCoarsenRS.c, 1106, 1108
 fasp_amg_coarsening_rs, 1107
 PreAMGInterp.c, 1130, 1131
 fasp_amg_interp, 1131
 PreAMGInterpEM.c, 1141, 1142
 fasp_amg_interp_em, 1142
 PreAMGSetupCR.c, 1151, 1152
 fasp_amg_setup_cr, 1151
 PreAMGSetupRS.c, 1153, 1154
 fasp_amg_setup_rs, 1154
 PreAMGSetupSA.c, 1158, 1159
 fasp_amg_setup_sa, 1159
 PreAMGSetupSABSR.c, 1168, 1170
 fasp_amg_setup_sa_bsr, 1169
 PreAMGSetupUA.c, 1175, 1176
 fasp_amg_setup_ua, 1175
 PreAMGSetupUABSR.c, 1181, 1182
 fasp_amg_setup_ua_bsr, 1181
 PreBLC.c, 1186, 1196
 fasp_precond_dblc_diag_3, 1187
 fasp_precond_dblc_diag_3_amg, 1187
 fasp_precond_dblc_diag_4, 1188
 fasp_precond_dblc_lower_3, 1188
 fasp_precond_dblc_lower_3_amg, 1190
 fasp_precond_dblc_lower_4, 1190
 fasp_precond_dblc_SGS_3, 1192
 fasp_precond_dblc_SGS_3_amg, 1192
 fasp_precond_dblc_sweeping, 1194
 fasp_precond_dblc_upper_3, 1194
 fasp_precond_dblc_upper_3_amg, 1196
 PreBSR.c, 1208, 1215
 fasp_precond_dbsr_amg, 1209
 fasp_precond_dbsr_amg_nk, 1209
 fasp_precond_dbsr_diag, 1210
 fasp_precond_dbsr_diag_nc2, 1210
 fasp_precond_dbsr_diag_nc3, 1211
 fasp_precond_dbsr_diag_nc5, 1211
 fasp_precond_dbsr_diag_nc7, 1212
 fasp_precond_dbsr_ilu, 1212
 fasp_precond_dbsr_ilu_ls_omp, 1213
 fasp_precond_dbsr_ilu_mc_omp, 1213
 fasp_precond_dbsr_namli, 1214
 PREC_AMG
 fasp_const.h, 191
 PREC_DIAG
 fasp_const.h, 191
 PREC_FMG
 fasp_const.h, 191
 PREC_ILU
 fasp_const.h, 191
 PREC_NULL
 fasp_const.h, 192
 PREC_SCHWARZ
 fasp_const.h, 192
 precond, 97
 data, 98
 fct, 98
 precond_data, 99
 A, 100
 A_nk, 100
 AMG_type, 100
 amli_coef, 101
 amli_degree, 101
 coarse_scaling, 101
 coarse_solver, 101
 coarsening_type, 101
 cycle_type, 102
 LU, 102
 max_levels, 102
 maxit, 102
 mgl_data, 102
 nl_amli_krylov_type, 103
 P_nk, 103
 polynomial_degree, 103
 postsmooth_iter, 103
 presmooth_iter, 103
 print_level, 104
 r, 104

- R_nk, 104
- relaxation, 104
- smooth_order, 104
- smoother, 105
- tentative_smooth, 105
- tol, 105
- w, 105
- precond_data_blc, 106
 - A_diag, 106
 - Abc, 106
 - amgparam, 106
 - LU_diag, 107
 - mgl, 107
 - r, 107
- precond_data_bsr, 107
 - A, 109
 - A_nk, 109
 - AMG_type, 109
 - amli_coef, 110
 - amli_degree, 110
 - coarse_scaling, 110
 - coarse_solver, 110
 - coarsening_type, 110
 - cycle_type, 111
 - LU, 111
 - max_levels, 111
 - maxit, 111
 - mgl_data, 111
 - nl_amli_krylov_type, 112
 - P_nk, 112
 - postsmooth_iter, 112
 - pres_mgl_data, 112
 - presmooth_iter, 112
 - print_level, 113
 - r, 113
 - R_nk, 113
 - relaxation, 113
 - smooth_order, 113
 - smoother, 114
 - tentative_smooth, 114
 - tol, 114
 - w, 114
- precond_data_str, 115
 - A, 116
 - A_str, 116
 - AMG_type, 116
 - coarse_scaling, 117
 - coarsening_type, 117
 - cycle_type, 117
 - diaginv, 117
 - diaginvS, 117
 - LU, 118
 - max_levels, 118
 - maxit, 118
 - mgl_data, 118
 - neigh, 118
 - order, 119
 - pivot, 119
 - pivotS, 119
 - postsmooth_iter, 119
 - presmooth_iter, 119
 - print_level, 120
 - r, 120
 - relaxation, 120
 - scaled, 120
 - smoother, 120
 - SS_str, 121
 - tol, 121
 - w, 121
- precond_data_sweeping, 121
 - A, 122
 - Ai, 122
 - local_A, 123
 - local_index, 123
 - local_LU, 123
 - NumLayers, 123
 - r, 123
 - w, 124
- precond_diag_bsr, 124
 - diag, 125
 - nb, 125
- precond_diag_str, 125
 - diag, 126
 - nc, 126
- precond_type
 - input_param, 89
 - ITS_param, 93
- PreCSR.c, 1227, 1233
 - fasp_precond_amg, 1228
 - fasp_precond_amg_nk, 1228
 - fasp_precond_amli, 1229
 - fasp_precond_diag, 1229
 - fasp_precond_famg, 1230
 - fasp_precond_ilu, 1230
 - fasp_precond_ilu_backward, 1231
 - fasp_precond_ilu_forward, 1231
 - fasp_precond_namli, 1231
 - fasp_precond_setup, 1232
 - fasp_precond_swz, 1232
- PreDataInit.c, 1239, 1243
 - fasp_amg_data_bsr_create, 1240
 - fasp_amg_data_bsr_free, 1240
 - fasp_amg_data_create, 1241
 - fasp_amg_data_free, 1241
 - fasp_ilu_data_create, 1242
 - fasp_ilu_data_free, 1242
 - fasp_precond_data_init, 1242
 - fasp_swz_data_free, 1243

PreMGCycle.c, [1247](#), [1248](#)
 fasp_solver_mgcycle, [1247](#)
 fasp_solver_mgcycle_bsr, [1248](#)
 PreMGCycleFull.c, [1254](#), [1256](#)
 fasp_solver_fmgcycle, [1255](#)
 PreMGRecur.c, [1260](#), [1261](#)
 fasp_solver_mgrecur, [1260](#)
 PreMGRecurAMLI.c, [1263](#), [1266](#)
 fasp_amg_amli_coef, [1263](#)
 fasp_solver_amli, [1264](#)
 fasp_solver_namli, [1265](#)
 fasp_solver_namli_bsr, [1265](#)
 PreMGSolve.c, [1274](#), [1277](#)
 fasp_amg_solve, [1275](#)
 fasp_amg_solve_amli, [1276](#)
 fasp_amg_solve_namli, [1276](#)
 fasp_famg_solve, [1277](#)
 pres_mgl_data
 precond_data_bsr, [112](#)
 presmooth_iter
 AMG_param, [38](#)
 precond_data, [103](#)
 precond_data_bsr, [112](#)
 precond_data_str, [119](#)
 PreSTR.c, [1281](#), [1285](#)
 fasp_precond_dstr_blockgs, [1282](#)
 fasp_precond_dstr_diag, [1282](#)
 fasp_precond_dstr_ilu0, [1283](#)
 fasp_precond_dstr_ilu0_backward, [1283](#)
 fasp_precond_dstr_ilu0_forward, [1283](#)
 fasp_precond_dstr_ilu1, [1284](#)
 fasp_precond_dstr_ilu1_backward, [1284](#)
 fasp_precond_dstr_ilu1_forward, [1285](#)
 PRINT_ALL
 fasp_const.h, [192](#)
 print_level
 AMG_param, [38](#)
 ILU_param, [79](#)
 input_param, [89](#)
 ITS_param, [93](#)
 precond_data, [104](#)
 precond_data_bsr, [113](#)
 precond_data_str, [120](#)
 SWZ_param, [132](#)
 PRINT_MIN
 fasp_const.h, [192](#)
 PRINT_MORE
 fasp_const.h, [192](#)
 PRINT_MOST
 fasp_const.h, [192](#)
 PRINT_NONE
 fasp_const.h, [192](#)
 PRINT_SOME
 fasp_const.h, [192](#)
 problem_num
 input_param, [90](#)
 pt
 Pardiso_data, [97](#)
 PUT_INT
 fasp.h, [161](#)
 PUT_REAL
 fasp.h, [161](#)
 pw
 AMG_data_bsr, [29](#)
 quality_bound
 AMG_param, [38](#)
 R
 AMG_data, [22](#)
 AMG_data_bsr, [29](#)
 r
 precond_data, [104](#)
 precond_data_blc, [107](#)
 precond_data_bsr, [113](#)
 precond_data_str, [120](#)
 precond_data_sweeping, [123](#)
 R_nk
 AMG_data_bsr, [29](#)
 precond_data, [104](#)
 precond_data_bsr, [113](#)
 REAL
 fasp.h, [161](#)
 relaxation
 AMG_param, [39](#)
 precond_data, [104](#)
 precond_data_bsr, [113](#)
 precond_data_str, [120](#)
 restart
 input_param, [90](#)
 ITS_param, [93](#)
 rhsloc
 SWZ_data, [130](#)
 rhsloc1
 SWZ_data, [130](#)
 ROW
 dBSRmat, [47](#)
 row
 dCOOmat, [49](#)
 dCSRLmat, [52](#)
 dCSRmat, [55](#)
 ddenmat, [56](#)
 dvector, [60](#)
 iCOOmat, [67](#)
 iCSRmat, [70](#)
 idenmat, [71](#)
 ILU_data, [76](#)
 ivector, [94](#)
 rowind

- dCOOmat, [50](#)
- RS_C1
 - fasp.h, [162](#)
- s
 - grid2d, [63](#)
- SA_AMG
 - fasp_const.h, [193](#)
- scaled
 - precond_data_str, [120](#)
- Schwarz
 - AMG_data, [22](#)
- SCHWARZ_BACKWARD
 - fasp_const.h, [193](#)
- SCHWARZ_FORWARD
 - fasp_const.h, [193](#)
- SCHWARZ_SYMMETRIC
 - fasp_const.h, [193](#)
- SHORT
 - fasp.h, [162](#)
- SMALLREAL
 - fasp_const.h, [193](#)
- SMALLREAL2
 - fasp_const.h, [193](#)
- smooth_filter
 - AMG_param, [39](#)
- smooth_order
 - AMG_param, [39](#)
 - precond_data, [104](#)
 - precond_data_bsr, [113](#)
- smooth_restriction
 - AMG_param, [39](#)
- smoother
 - AMG_param, [39](#)
 - precond_data, [105](#)
 - precond_data_bsr, [114](#)
 - precond_data_str, [120](#)
- SMOOTHER_BLKOil
 - fasp_const.h, [193](#)
- SMOOTHER_CG
 - fasp_const.h, [194](#)
- SMOOTHER_GS
 - fasp_const.h, [194](#)
- SMOOTHER_GSOR
 - fasp_const.h, [194](#)
- SMOOTHER_JACOBI
 - fasp_const.h, [194](#)
- SMOOTHER_L1DIAG
 - fasp_const.h, [194](#)
- SMOOTHER_POLY
 - fasp_const.h, [194](#)
- SMOOTHER_SGS
 - fasp_const.h, [194](#)
- SMOOTHER_SGSOR
 - fasp_const.h, [194](#)
- SMOOTHER_SOR
 - fasp_const.h, [195](#)
- SMOOTHER_SPETEN
 - fasp_const.h, [195](#)
- SMOOTHER_SSOR
 - fasp_const.h, [195](#)
- SolAMG.c, [1307](#), [1308](#)
 - fasp_solver_amg, [1307](#)
- SolBLC.c, [1309](#), [1314](#)
 - fasp_solver_dblc_itsolver, [1310](#)
 - fasp_solver_dblc_krylov, [1311](#)
 - fasp_solver_dblc_krylov_block3, [1311](#)
 - fasp_solver_dblc_krylov_block4, [1312](#)
 - fasp_solver_dblc_krylov_sweeping, [1313](#)
- SolBSR.c, [1319](#), [1325](#)
 - fasp_solver_dbsr_itsolver, [1320](#)
 - fasp_solver_dbsr_krylov, [1321](#)
 - fasp_solver_dbsr_krylov_amg, [1321](#)
 - fasp_solver_dbsr_krylov_amg_nk, [1322](#)
 - fasp_solver_dbsr_krylov_diag, [1323](#)
 - fasp_solver_dbsr_krylov_ilu, [1323](#)
 - fasp_solver_dbsr_krylov_nk_amg, [1324](#)
- SolCSR.c, [1333](#), [1340](#)
 - fasp_solver_dcsr_itsolver, [1334](#)
 - fasp_solver_dcsr_itsolver_s, [1334](#)
 - fasp_solver_dcsr_krylov, [1335](#)
 - fasp_solver_dcsr_krylov_amg, [1336](#)
 - fasp_solver_dcsr_krylov_amg_nk, [1336](#)
 - fasp_solver_dcsr_krylov_diag, [1337](#)
 - fasp_solver_dcsr_krylov_ilu, [1337](#)
 - fasp_solver_dcsr_krylov_ilu_M, [1338](#)
 - fasp_solver_dcsr_krylov_s, [1339](#)
 - fasp_solver_dcsr_krylov_swz, [1339](#)
- SolFAMG.c, [1348](#), [1349](#)
 - fasp_solver_famg, [1349](#)
- SolGMGPoisson.c, [1351](#), [1358](#)
 - fasp_poisson_fgmg1d, [1352](#)
 - fasp_poisson_fgmg2d, [1352](#)
 - fasp_poisson_fgmg3d, [1353](#)
 - fasp_poisson_gmg1d, [1354](#)
 - fasp_poisson_gmg2d, [1354](#)
 - fasp_poisson_gmg3d, [1355](#)
 - fasp_poisson_gmgcg1d, [1356](#)
 - fasp_poisson_gmgcg2d, [1356](#)
 - fasp_poisson_gmgcg3d, [1357](#)
- SolMatFree.c, [1369](#), [1371](#)
 - fasp_solver_itsolver, [1369](#)
 - fasp_solver_krylov, [1370](#)
 - fasp_solver_matfree_init, [1370](#)
- SolSTR.c, [1373](#), [1377](#)
 - fasp_solver_dstr_itsolver, [1374](#)
 - fasp_solver_dstr_krylov, [1375](#)
 - fasp_solver_dstr_krylov_blockgs, [1375](#)

- xloc1, [131](#)
- SWZ_levels
 - AMG_data, [22](#)
 - AMG_param, [40](#)
- SWZ_maxlvl
 - AMG_param, [40](#)
 - input_param, [91](#)
 - SWZ_param, [132](#)
- SWZ_mmsize
 - AMG_param, [41](#)
 - input_param, [91](#)
 - SWZ_param, [133](#)
- SWZ_param, [131](#)
 - print_level, [132](#)
 - SWZ_blksolver, [132](#)
 - SWZ_maxlvl, [132](#)
 - SWZ_mmsize, [133](#)
 - SWZ_type, [133](#)
- SWZ_type
 - AMG_param, [41](#)
 - input_param, [91](#)
 - SWZ_data, [131](#)
 - SWZ_param, [133](#)
- swzparam
 - SWZ_data, [131](#)
- t
 - grid2d, [63](#)
- tentative_smooth
 - AMG_param, [41](#)
 - precond_data, [105](#)
 - precond_data_bsr, [114](#)
- tfather
 - grid2d, [64](#)
- THDs_AMG_GS
 - AuxThreads.c, [277](#)
- THDs_CPR_gGS
 - AuxThreads.c, [277](#)
- THDs_CPR_IGS
 - AuxThreads.c, [278](#)
- tol
 - AMG_param, [41](#)
 - ITS_param, [93](#)
 - precond_data, [105](#)
 - precond_data_bsr, [114](#)
 - precond_data_str, [121](#)
- topologic_sort_ILU
 - BlaILUSetupBSR.c, [358](#)
- total_alloc_count
 - BlaSpmvCSR.c, [706](#)
- total_alloc_mem
 - BlaSpmvCSR.c, [706](#)
- triangles
 - grid2d, [64](#)
- TRUE
 - fasp_const.h, [198](#)
- truncation_threshold
 - AMG_param, [41](#)
- type
 - ILU_data, [76](#)
- UA_AMG
 - fasp_const.h, [199](#)
- UNPT
 - fasp_const.h, [199](#)
- uptr
 - ILU_data, [76](#)
- USERDEFINED
 - fasp_const.h, [199](#)
- V_CYCLE
 - fasp_const.h, [199](#)
- val
 - dBSRmat, [48](#)
 - dCOOmat, [50](#)
 - dCSRLmat, [53](#)
 - dCSRmat, [55](#)
 - ddenmat, [56](#)
 - dvector, [61](#)
 - iCOOmat, [68](#)
 - iCSRmat, [70](#)
 - idenmat, [71](#)
 - ivector, [94](#)
- vertices
 - grid2d, [64](#)
- VMB
 - fasp_const.h, [199](#)
- VW_CYCLE
 - fasp_const.h, [199](#)
- w
 - AMG_data, [22](#)
 - AMG_data_bsr, [30](#)
 - precond_data, [105](#)
 - precond_data_bsr, [114](#)
 - precond_data_str, [121](#)
 - precond_data_sweeping, [124](#)
- W_CYCLE
 - fasp_const.h, [199](#)
- weight
 - AMG_data, [22](#)
- work
 - ILU_data, [77](#)
- workdir
 - input_param, [91](#)
- WV_CYCLE
 - fasp_const.h, [199](#)
- x

- AMG_data, [23](#)
- AMG_data_bsr, [30](#)
- xloc1
 - SWZ_data, [131](#)
- XtrMumps.c, [135](#), [137](#)
 - fasp_solver_mumps, [136](#)
 - fasp_solver_mumps_steps, [137](#)
 - ICNTL, [136](#)
- XtrPardiso.c, [143](#), [144](#)
 - fasp_solver_pardiso, [144](#)
- XtrSamg.c, [147](#), [149](#)
 - dCSRmat2SAMGInput, [148](#)
 - dvector2SAMGInput, [148](#)
- XtrSuperlu.c, [150](#), [152](#)
 - fasp_solver_superlu, [151](#)
- XtrUmfpack.c, [153](#), [154](#)
 - fasp_solver_umfpack, [153](#)