Fast Auxiliary Space Preconditioning

Generated by Doxygen 1.9.2

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 systemsfrom 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.

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 "faspsolver/doc/".

To compile, you need a Fortran and a C compiler. First, you can type in the "faspsolver/" 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 "faspsolver/".

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:

Chapter 7

File Index

7.1 File List

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

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 <= 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 <= 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 diaginv_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 <= 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 presmoothers
- SHORT postsmooth_iter

  number of postsmoothers
- 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 amli_degree

  degree of the polynomial used by AMLI cycle
- REAL $*$ amli_coef

  coefficients of the polynomial used by AMLI cycle
- SHORT nl_amli_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_lfil

  level of fill-in for ILUs and ILUk
- REAL ILU_droptol

  drop tolerance for ILUt
- REAL ILU_relax

  relaxation for ILUs
- REAL ILU_permtol

  permuted if 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 ILUt

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 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],...,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 nband ∗ [(ngrid-|offsets|) ∗ 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 ∗ pdiri
- INT ∗ ediri
- 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 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 ILUt

Definition at line 408 of file fasp.h.

8.20.2.2 ILU_lfil

```
INT ILU_lfil
```

level of fill-in for ILUk

Definition at line 405 of file fasp.h.

8.20.2.3 ILU_permtol

```
REAL ILU_permtol
```

permuted if 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 precond_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 precond_type

```
SHORT precond_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 precond_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 precond_type

```
SHORT precond_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 precond 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 precond_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 precond_data_blc Struct Reference

Data for block preconditioners in dBLCmat format.

`#include <fasp_block.h>`

Data Fields

- dBLCmat ∗ 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 dBLCmat 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

```
dBLCmat∗ 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 precond_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 amli_degree

  degree of the polynomial used by AMLI cycle
- REAL ∗ amli_coef

  coefficients of the polynomial used by AMLI cycle
- REAL tentative_smooth

  smooth factor for smoothing the tentative prolongation
- SHORT nl_amli_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
```

Resriction for near kernal.

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 precond_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 precond_data_sweeping Struct Reference

Data for sweeping preconditioner.

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

Data Fields

- INT NumLayers
- dBLCmat * A
- dBLCmat * 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

```
dBLCmat* A
```

problem data, the sparse matrix

Definition at line 388 of file fasp_block.h.

8.32.2.2 Ai

```
dBLCmat* 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
```

lcoal 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 diagnal preconditioners in dBSRmat format.

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

Data Fields

- INT nb

  dimension of each sub-block
- dvector diag

  diagnal elements

8.33.1 Detailed Description

Data for diagnal preconditioners in dBSRmat format.

Note

This is needed for the diagnal preconditioner.

Definition at line 241 of file fasp_block.h.

8.33.2 Field Documentation

8.33.2.1 diag

```
dvector diag
```

diagnal 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 precond_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/smoother.

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.

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(I) icntl[(I)-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/`

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

ptrA	Pointer to a dCSRmat matrix
b	Pointer to the dvector of right-hand side term
u	Pointer to the dvector of solution
prtlvl	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

ptrA	Pointer to a dCSRmat matrix
b	Pointer to the dvector of right-hand side term
u	Pointer to the dvector of solution
mumps	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;
```

```
    const int nz = ptrA->nnz;
    int*      IA = ptrA->IA;
    int*      JA = ptrA->JA;
    double*   AA = ptrA->val;
    double*   f  = b->val;
    double*   x  = u->val;

    int*    irn;
    int*    jcn;
    double* a;
    double* rhs;
    int     i, j;
    int     begin_row, end_row;

#if DEBUG_MODE
    printf("### DEBUG: fasp_solver_mumps ...  [Start]\n");
    printf("### DEBUG: nr=%d,  nnz=%d\n", n, nz);
#endif

    // First check the matrix format
    if (IA[0] != 0 && IA[0] != 1) {
        printf("### ERROR: Matrix format is wrong -- IA[0] = %d\n", IA[0]);
        return ERROR_SOLVER_EXIT;
    }

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    /* Define A and rhs */
    irn = (int*)malloc(sizeof(int) * nz);
    jcn = (int*)malloc(sizeof(int) * nz);
    a   = (double*)malloc(sizeof(double) * nz);
    rhs = (double*)malloc(sizeof(double) * n);

    if (IA[0] == 0) { // C-convention
        for (i = 0; i < n; i++) {
            begin_row = IA[i];
            end_row   = IA[i + 1];
            for (j = begin_row; j < end_row; j++) {
                irn[j] = i + 1;
                jcn[j] = JA[j] + 1;
                a[j]   = AA[j];
            }
        }
    } else { // F-convention
        for (i = 0; i < n; i++) {
            begin_row = IA[i] - 1;
            end_row   = IA[i + 1] - 1;
            for (j = begin_row; j < end_row; j++) {
                irn[j] = i + 1;
                jcn[j] = JA[j];
                a[j]   = AA[j];
            }
        }
    }

    /* Initialize a MUMPS instance.  */
    id.job          = -1;
    id.par          = 1; // host involved in factorization/solve
    id.sym          = 0; // 0:  general, 1:  spd, 2:  sym
    id.comm_fortran = 0;
    dmumps_c(&id);

    /* Define the problem on the host */
    id.n   = n;
    id.nz  = nz;
    id.irn = irn;
    id.jcn = jcn;
    id.a   = a;
    id.rhs = rhs;

    if (prtlvl < PRINT_MOST) { // no debug
        id.ICNTL(1) = -1;
        id.ICNTL(2) = -1;
        id.ICNTL(3) = -1;
        id.ICNTL(4) = 0;
    } else {              // debug
        id.ICNTL(1) = 6; // err output stream
        id.ICNTL(2) = 6; // warn/info output stream
        id.ICNTL(3) = 6; // global output stream
        id.ICNTL(4) = 3; // 0:none, 1:  err, 2:  warn/stats, 3:diagnostics, 4:parameters
    }

    /* Call the MUMPS package */
    for (i = 0; i < n; i++) rhs[i] = f[i];

    id.job = 6; /* Combines phase 1, 2, and 3 */
    dmumps_c(&id); /* Sometimes segmentation faults in phase 1 */

    for (i = 0; i < n; i++) x[i] = id.rhs[i];

    id.job = -2;
    dmumps_c(&id); /* Terminate instance */

    free(irn);
    free(jcn);
    free(a);
    free(rhs);

    if (prtlvl > PRINT_MIN) {
        fasp_gettime(&end_time);
        fasp_cputime("MUMPS solver", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: fasp_solver_mumps ...  [Finish]\n");
#endif
    return FASP_SUCCESS;

#else

    printf("### ERROR: MUMPS is not available!\n");
    return ERROR_SOLVER_EXIT;

#endif
}

int fasp_solver_mumps_steps(dCSRmat* ptrA, dvector* b, dvector* u, Mumps_data* mumps)
{
#if WITH_MUMPS

    DMUMPS_STRUC_C id;

    int job = mumps->job;

    static int job_stat = 0;
    int        i, j;

    int*    irn;
    int*    jcn;
    double* a;
    double* rhs;

    switch (job) {

        case 1:
            {
#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Start]\n",
                       __FUNCTION__, job, job_stat);
#endif
                int       begin_row, end_row;
                const int n  = ptrA->row;
                const int nz = ptrA->nnz;
                int*      IA = ptrA->IA;
                int*      JA = ptrA->JA;
                double*   AA = ptrA->val;

                irn = id.irn = (int*)malloc(sizeof(int) * nz);
                jcn = id.jcn = (int*)malloc(sizeof(int) * nz);
                a = id.a = (double*)malloc(sizeof(double) * nz);
                rhs = id.rhs = (double*)malloc(sizeof(double) * n);

                // First check the matrix format
                if (IA[0] != 0 && IA[0] != 1) {
                    printf("### ERROR: Matrix format is wrong, IA[0] = %d!\n", IA[0]);
                    return ERROR_SOLVER_EXIT;
                }

                // Define A and rhs
                if (IA[0] == 0) { // C-convention
                    for (i = 0; i < n; i++) {
                        begin_row = IA[i];
                        end_row   = IA[i + 1];
                        for (j = begin_row; j < end_row; j++) {
                            irn[j] = i + 1;
                            jcn[j] = JA[j] + 1;
                            a[j]   = AA[j];
                        }
                    }
                } else { // F-convention
                    for (i = 0; i < n; i++) {
                        begin_row = IA[i] - 1;
                        end_row   = IA[i + 1] - 1;
                        for (j = begin_row; j < end_row; j++) {
                            irn[j] = i + 1;
                            jcn[j] = JA[j];
                            a[j]   = AA[j];
                        }
                    }
                }

                /* Initialize a MUMPS instance.  */
                id.job          = -1;
                id.par          = 1;
                id.sym          = 0;
                id.comm_fortran = 0;
                dmumps_c(&id);

                /* Define the problem on the host */
                id.n   = n;
                id.nz  = nz;
                id.irn = irn;
                id.jcn = jcn;
                id.a   = a;
                id.rhs = rhs;

                /* No outputs */
                id.ICNTL(1) = -1;
                id.ICNTL(2) = -1;
                id.ICNTL(3) = -1;
                id.ICNTL(4) = 0;

                id.job = 4;
                dmumps_c(&id);
                job_stat = 1;

                mumps->id = id;

#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Finish]\n",
                       __FUNCTION__, job, job_stat);
#endif
                break;
            }

        case 2:
            {
#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Start]\n",
                       __FUNCTION__, job, job_stat);
#endif
                id = mumps->id;

                if (job_stat != 1)
                    printf("### ERROR: %s setup failed!\n", __FUNCTION__);

                /* Call the MUMPS package.  */
                for (i = 0; i < id.n; i++) id.rhs[i] = b->val[i];

                id.job = 3;
                dmumps_c(&id);

                for (i = 0; i < id.n; i++) u->val[i] = id.rhs[i];

#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Finish]\n",
                       __FUNCTION__, job, job_stat);
#endif
                break;
            }

        case 3:
            {
#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Start]\n",
                       __FUNCTION__, job, job_stat);
#endif
                id = mumps->id;

                if (job_stat != 1)
                    printf("### ERROR: %s setup failed!\n", __FUNCTION__);

                free(id.irn);
                free(id.jcn);
                free(id.a);
                free(id.rhs);
                id.job = -2;
                dmumps_c(&id); /* Terminate instance */

#if DEBUG_MODE
                printf("### DEBUG: %s, step %d, job_stat = %d...  [Finish]\n",
                       __FUNCTION__, job, job_stat);
#endif

                break;
            }

        default:
            printf("### ERROR: job = %d.  Should be 1, 2, or 3!\n", job);
            return ERROR_SOLVER_EXIT;
    }

    return FASP_SUCCESS;

#else

    printf("### ERROR: MUMPS is not available!\n");
    return ERROR_SOLVER_EXIT;

#endif
}

#if WITH_MUMPS
Mumps_data fasp_mumps_factorize(dCSRmat* ptrA, dvector* b, dvector* u,
                                const SHORT prtlvl)
{
    Mumps_data     mumps;
    DMUMPS_STRUC_C id;

    int       i, j;
    const int m  = ptrA->row;
    const int n  = ptrA->col;
    const int nz = ptrA->nnz;
    int*      IA = ptrA->IA;
    int*      JA = ptrA->JA;
    double*   AA = ptrA->val;

    int*    irn = id.irn = (int*)malloc(sizeof(int) * nz);
    int*    jcn = id.jcn = (int*)malloc(sizeof(int) * nz);
    double* a = id.a = (double*)malloc(sizeof(double) * nz);
    double* rhs = id.rhs = (double*)malloc(sizeof(double) * n);

    int begin_row, end_row;

#if DEBUG_MODE
    printf("### DEBUG: %s ...  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nz);
#endif

    clock_t start_time = clock();

    if (IA[0] == 0) { // C-convention
        for (i = 0; i < n; i++) {
            begin_row = IA[i];
            end_row   = IA[i + 1];
            for (j = begin_row; j < end_row; j++) {
                irn[j] = i + 1;
                jcn[j] = JA[j] + 1;
                a[j]   = AA[j];
            }
        }
    } else { // F-convention
        for (i = 0; i < n; i++) {
            begin_row = IA[i] - 1;
            end_row   = IA[i + 1] - 1;
            for (j = begin_row; j < end_row; j++) {
                irn[j] = i + 1;
                jcn[j] = JA[j];
                a[j]   = AA[j];
            }
        }
    }

    /* Initialize a MUMPS instance.  */
    id.job          = -1;
    id.par          = 1;
    id.sym          = 0;
    id.comm_fortran = 0;
    dmumps_c(&id);

    /* Define the problem on the host */
    id.n   = n;
    id.nz  = nz;
    id.irn = irn;
    id.jcn = jcn;
    id.a   = a;
    id.rhs = rhs;

    if (prtlvl < PRINT_MOST) { // no debug
        id.ICNTL(1) = -1;
        id.ICNTL(2) = -1;
        id.ICNTL(3) = -1;
        id.ICNTL(4) = 0;
    } else {              // debug
        id.ICNTL(1) = 6; // err output stream
        id.ICNTL(2) = 6; // warn/info output stream
        id.ICNTL(3) = 6; // global output stream
        id.ICNTL(4) = 3; // 0:none, 1:  err, 2:  warn/stats, 3:diagnostics, 4:parameters
    }

    id.job = 4;
    dmumps_c(&id);

    if (prtlvl > PRINT_MIN) {
        clock_t end_time = clock();
        double  fac_time = (double)(end_time - start_time) / (double)(CLOCKS_PER_SEC);
        printf("MUMPS factorize costs %f seconds.\n", fac_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ...  [Finish]\n", __FUNCTION__);
#endif

    mumps.id = id;

    return mumps;
}
#endif

#if WITH_MUMPS
void fasp_mumps_solve(dCSRmat* ptrA, dvector* b, dvector* u, Mumps_data mumps,
                      const SHORT prtlvl)
{
    int i, j;

    DMUMPS_STRUC_C id = mumps.id;

    const int m  = ptrA->row;
    const int n  = ptrA->row;
    const int nz = ptrA->nnz;
    int*      IA = ptrA->IA;
    int*      JA = ptrA->JA;
    double*   AA = ptrA->val;

    int*    irn = id.irn;
    int*    jcn = id.jcn;
    double* a   = id.a;
    double* rhs = id.rhs;

#if DEBUG_MODE
    printf("### DEBUG: %s ...  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nz);
#endif

```

```
    clock_t start_time = clock();

    double* f = b->val;
    double* x = u->val;

    /* Call the MUMPS package.  */
    for (i = 0; i < id.n; i++) rhs[i] = f[i];

    if (prtlvl < PRINT_MOST) { // no debug
        id.ICNTL(1) = -1;
        id.ICNTL(2) = -1;
        id.ICNTL(3) = -1;
        id.ICNTL(4) = 0;
    } else {                // debug
        id.ICNTL(1) = 6; // err output stream
        id.ICNTL(2) = 6; // warn/info output stream
        id.ICNTL(3) = 6; // global output stream
        id.ICNTL(4) = 3; // 0:none, 1:  err, 2:  warn/stats, 3:diagnostics, 4:parameters
    }

    id.job = 3;
    dmumps_c(&id);

    for (i = 0; i < id.n; i++) x[i] = id.rhs[i];

    if (prtlvl > PRINT_NONE) {
        clock_t end_time   = clock();
        double  solve_time = (double)(end_time - start_time) / (double)(CLOCKS_PER_SEC);
        printf("MUMPS costs %f seconds.\n", solve_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ...  [Finish]\n", __FUNCTION__);
#endif
}
#endif

#if WITH_MUMPS
void fasp_mumps_free(Mumps_data* mumps)
{
    DMUMPS_STRUC_C id = mumps->id;

    free(id.irn);
    free(id.jcn);
    free(id.a);
    free(id.rhs);
}
#endif

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.5 XtrPardiso.c File Reference

Interface to Intel MKL PARDISO direct solvers.

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

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

ptrA	Pointer to a dCSRmat matrix
b	Pointer to the dvector of right-hand side term
u	Pointer to the dvector of solution
prtlvl	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_functs.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,
                             const SHORT prtlvl)
{
#if WITH_PARDISO

    INT status = FASP_SUCCESS;

    MKL_INT n = ptrA->col;
    MKL_INT *ia = ptrA->IA;
    MKL_INT *ja = ptrA->JA;
    REAL *a = ptrA->val;

    MKL_INT mtype = 11;     /* Real unsymmetric matrix */
    MKL_INT nrhs = 1;       /* Number of right hand sides */
    MKL_INT idum;           /* Integer dummy */
    MKL_INT iparm[64];      /* Pardiso control parameters */
    MKL_INT maxfct, mnum, phase, error, msglvl;    /* Auxiliary variables */

    REAL * f = b->val;      /* RHS vector */
    REAL * x = u->val;      /* Solution vector */
    void *pt[64];           /* Internal solver memory pointer pt */
    double ddum;            /* Double dummy */

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
#endif

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    PARDISOINIT(pt, &mtype, iparm); /* Initialize */
    iparm[34] = 1;         /* Use 0-based indexing */
    maxfct = 1;            /* Maximum number of numerical factorizations */
    mnum = 1;              /* Which factorization to use */
    msglvl = 0;            /* Do not print statistical information in file */
    error = 0;             /* Initialize error flag */

    phase = 11; /* Reordering and symbolic factorization */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 ) {
        printf ("### ERROR: Symbolic factorization failed %d!\n", error);
        exit (1);
    }

    phase = 22; /* Numerical factorization */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 ) {
        printf ("\n### ERROR: Numerical factorization failed %d!\n", error);
        exit (2);
    }

    phase = 33; /* Back substitution and iterative refinement */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs, iparm, &msglvl, f, x, &error);

    if ( error != 0 ) {
        printf ("\n### ERROR: Solution failed %d!\n", error);
        exit (3);
    }

    if ( prtlvl > PRINT_MIN ) {
        fasp_gettime(&end_time);
        fasp_cputime("PARDISO solver", end_time - start_time);
    }

    phase = -1; /* Release internal memory */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, &ddum, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;

#else

    printf("### ERROR: PARDISO is not available!\n");
    return ERROR_SOLVER_EXIT;

#endif

}

#if WITH_PARDISO
INT fasp_pardiso_factorize (dCSRmat *ptrA,
                            Pardiso_data *pdata,
                            const SHORT prtlvl)
{
    INT status = FASP_SUCCESS;

    MKL_INT n = ptrA->col;
    MKL_INT *ia = ptrA->IA;
    MKL_INT *ja = ptrA->JA;
    REAL *a = ptrA->val;

    double  ddum;         /* Double dummy */
    MKL_INT nrhs = 1;     /* Number of right hand sides */
    MKL_INT idum;         /* Integer dummy */
    MKL_INT phase, error, msglvl;    /* Auxiliary variables */

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
#endif

    pdata->mtype = 11;    /* Real unsymmetric matrix */

    PARDISOINIT(pdata->pt, &(pdata->mtype), pdata->iparm); /* Initialize */
    pdata->iparm[34] = 1;  /* Use 0-based indexing */

    /* Numbers of processors, value of OMP_NUM_THREADS */
#ifdef _OPENMP
    pdata->iparm[2]  = fasp_get_num_threads();
#endif

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    pdata->maxfct = 1;     /* Maximum number of numerical factorizations */
    pdata->mnum = 1;       /* Which factorization to use */
    msglvl = 0;            /* Do not print statistical information in file */
    error = 0;             /* Initialize error flag */

    phase = 11; /* Reordering and symbolic factorization */
    PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase, &n,
             a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 ) {
        printf ("### ERROR: Symbolic factorization failed %d!\n", error);
        exit (1);
    }

    phase = 22; /* Numerical factorization */
    PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase, &n,
             a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum, &ddum, &error);

    if ( error != 0 ) {
        printf ("\n### ERROR: Numerical factorization failed %d!\n", error);
        exit (2);
    }

    if ( prtlvl > PRINT_MIN ) {
        fasp_gettime(&end_time);
        fasp_cputime("PARDISO setup", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_pardiso_solve (dCSRmat *ptrA,
                        dvector *b,
                        dvector *u,
                        Pardiso_data *pdata,
                        const SHORT prtlvl)
{
    INT status = FASP_SUCCESS;

    MKL_INT n = ptrA->col;
    MKL_INT *ia = ptrA->IA;
    MKL_INT *ja = ptrA->JA;

    REAL *a = ptrA->val;
    REAL * f = b->val;      /* RHS vector */
    REAL * x = u->val;      /* Solution vector */
    MKL_INT nrhs = 1;       /* Number of right hand sides */
    MKL_INT idum;           /* Integer dummy */
    MKL_INT phase, error, msglvl;    /* Auxiliary variables */

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    msglvl = 0; /* Do not print statistical information in file */

    phase = 33; /* Back substitution and iterative refinement */
    PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase,
             &n, a, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, f, x, &error);

    if ( error != 0 ) {
        printf ("### ERROR: Solution failed %d!\n", error);
        exit (3);
    }

    if ( prtlvl > PRINT_MIN ) {
        fasp_gettime(&end_time);
        fasp_cputime("PARDISO solve", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_pardiso_free_internal_mem (Pardiso_data *pdata)
{
    INT status = FASP_SUCCESS;

    MKL_INT *ia = NULL;
    MKL_INT *ja = NULL;

    double  ddum;          /* Double dummy */
    MKL_INT idum;          /* Integer dummy */
    MKL_INT nrhs = 1;      /* Number of right hand sides */
    MKL_INT phase, error, msglvl;    /* Auxiliary variables */

    msglvl = 0; /* Do not print statistical information in file */

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
#endif

    phase = -1;             /* Release internal memory */
    PARDISO (pdata->pt, &(pdata->maxfct), &(pdata->mnum), &(pdata->mtype), &phase,
             &idum, &ddum, ia, ja, &idum, &nrhs, pdata->iparm, &msglvl, &ddum,
             &ddum, &error);

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;
}

#endif

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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)!

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

A	Pointer to the dCSRmat matrix
filefrm	Name of the .frm file
fileamg	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

vec	Pointer to the dvector
filename	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;
```

```
    INT       rowsum_type   = 0;
    INT       symmetry_type = 0;

    INT       i,j;
    REAL      rowsum;

    fasp_dcsr_diagpref(A);

    /* check symmetry type of the matrix */
    symmetry_type = fasp_check_symm(A);

    /* check rowsum type of the matrix */
    for (i = 0; i < num_rowsA; ++i) {
        rowsum = 0.0;
        for (j = A_i[i]; j < A_i[i+1]; ++j) {
            rowsum += A_data[j];
        }
        if (rowsum*rowsum > 0.0) {
            rowsum_type = 1;
            break;
        }
    }

    /* Get the matrix type of A */
    if (symmetry_type == 0) {
        if (rowsum_type == 0)
            matrix_type = 11;
        else
            matrix_type = 12;
    }
    else {
        if (rowsum_type == 0)
            matrix_type = 21;
        else
            matrix_type = 22;
    }

    /* write the *.frm file */
    fp = fopen(filefrm, "w");
    fprintf(fp, "%s   %d\n", "f", 4);
    fprintf(fp, "%d %d %d %d %d\n", num_nonzeros, num_rowsA, matrix_type, 1, 0);
    fclose(fp);

    /* write the *.amg file */
    fp = fopen(fileamg, "w");
    for (j = 0; j <= num_rowsA; ++j) {
        fprintf(fp, "%d\n", A_i[j] + file_base);
    }
    for (j = 0; j < num_nonzeros; ++j) {
        fprintf(fp, "%d\n", A_j[j] + file_base);
    }
    if (A_data) {
        for (j = 0; j < num_nonzeros; ++j) {
            fprintf(fp, "%.15le\n", A_data[j]); // we always use "%.15le\n"
        }
    }
    else {
        fprintf(fp, "### WARNING: No matrix data!\n");
    }
    fclose(fp);

    return FASP_SUCCESS;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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_functs.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/

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

ptrA	Pointer to a dCSRmat matrix
b	Pointer to the dvector of right-hand side term
u	Pointer to the dvector of solution
prtlvl	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.

```c

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

#include "fasp.h"
#include "fasp_functs.h"

#if WITH_SuperLU
#include "slu_ddefs.h"
#endif

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

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

#if WITH_SuperLU

    SuperMatrix A, L, U, B;

    int* perm_r; /* row permutations from partial pivoting */
    int* perm_c; /* column permutation vector */
    int  nrhs = 1, info, m = ptrA->row, n = ptrA->col, nnz = ptrA->nnz;

    if (prtlvl > PRINT_NONE) printf("superlu:  nr=%d, nc=%d, nnz=%d\n", m, n, nnz);

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    dCSRmat tempA = fasp_dcsr_create(m, n, nnz);
    fasp_dcsr_cp(ptrA, &tempA);

    dvector tempb = fasp_dvec_create(n);
    fasp_dvec_cp(b, &tempb);

    /* Create matrix A in the format expected by SuperLU. */
    dCreate_CompCol_Matrix(&A, m, n, nnz, tempA.val, tempA.JA, tempA.IA, SLU_NR, SLU_D,
                           SLU_GE);

    /* Create right-hand side B. */
    dCreate_Dense_Matrix(&B, m, nrhs, tempb.val, m, SLU_DN, SLU_D, SLU_GE);

    if (!(perm_r = intMalloc(m))) ABORT("Malloc fails for perm_r[].");
    if (!(perm_c = intMalloc(n))) ABORT("Malloc fails for perm_c[].");

    /* Set the default input options.  */
    superlu_options_t options;
    set_default_options(&options);
    options.ColPerm = COLAMD; // MMD_AT_PLUS_A; MMD_ATA; NATURAL;

    /* Initialize the statistics variables.  */
    SuperLUStat_t stat;
    StatInit(&stat);

    /* SuperLU */
    dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);

    DNformat* BB = (DNformat*)B.Store;
    u->val       = (double*)BB->nzval;
    u->row       = n;

    if (prtlvl > PRINT_MIN) {
        fasp_gettime(&end_time);
        fasp_cputime("SUPERLU solver", end_time - start_time);
    }

    /* De-allocate storage */
    SUPERLU_FREE(perm_r);
    SUPERLU_FREE(perm_c);
    Destroy_CompCol_Matrix(&A);
    Destroy_SuperMatrix_Store(&B);
    Destroy_SuperNode_Matrix(&L);
    Destroy_CompCol_Matrix(&U);
    StatFree(&stat);

    return FASP_SUCCESS;

#else

    printf("### ERROR: SuperLU is not available!\n");
    return ERROR_SOLVER_EXIT;

#endif
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

ptrA	Pointer to a dCSRmat matrix
b	Pointer to the dvector of right-hand side term
u	Pointer to the dvector of solution
prtlvl	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_functs.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);
```

```
    }
    umfpack_di_free_numeric(&Numeric);

    if ( prtlvl > PRINT_MIN ) {
        fasp_gettime(&end_time);
        fasp_cputime("UMFPACK costs", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;

#else

    printf("### ERROR: UMFPACK is not available!\n");
    return ERROR_SOLVER_EXIT;

#endif

}

#if WITH_UMFPACK
void* fasp_umfpack_factorize (dCSRmat *ptrA,
                              const SHORT prtlvl)
{
    const INT n = ptrA->col;

    INT *Ap = ptrA->IA;
    INT *Ai = ptrA->JA;
    double *Ax = ptrA->val;
    void *Symbolic;
    void *Numeric;

#if DEBUG_MODE
    const INT m = ptrA->row;
    const INT nnz = ptrA->nnz;
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
#endif

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    umfpack_di_symbolic (n, n, Ap, Ai, Ax, &Symbolic, NULL, NULL);
    umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric, NULL, NULL);
    umfpack_di_free_symbolic (&Symbolic);

    if ( prtlvl > PRINT_MIN ) {
        fasp_gettime(&end_time);
        fasp_cputime("UMFPACK setup", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return Numeric;
}

INT fasp_umfpack_solve (dCSRmat *ptrA,
                        dvector *b,
                        dvector *u,
                        void *Numeric,
                        const SHORT prtlvl)
{
    INT *Ap = ptrA->IA;
    INT *Ai = ptrA->JA;
    double *Ax = ptrA->val;
    INT status = FASP_SUCCESS;

#if DEBUG_MODE
    const INT m = ptrA->row;
    const INT n = ptrA->col;
    const INT nnz = ptrA->nnz;
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
#endif

    REAL start_time, end_time;
    fasp_gettime(&start_time);

    status = umfpack_di_solve (UMFPACK_A, Ap, Ai, Ax, u->val, b->val, Numeric, NULL, NULL);

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&end_time);
        fasp_cputime("UMFPACK solve", end_time - start_time);
    }

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_umfpack_free_numeric (void *Numeric)
{
    INT status = FASP_SUCCESS;

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Start]\n", __FUNCTION__);
#endif

    umfpack_di_free_numeric (&Numeric);

#if DEBUG_MODE
    printf("### DEBUG: %s ......  [Finish]\n", __FUNCTION__);
#endif

    return status;
}

#endif

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

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.

```

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

#include "fasp_const.h"

#if WITH_MUMPS
#include "dmumps_c.h"
#endif

#if WITH_PARDISO
#include "mkl_pardiso.h"
#include "mkl_types.h"
#endif

#ifndef __FASP_HEADER__      /*-- allow multiple inclusions --*/
#define __FASP_HEADER__
/*---------------------------*/
/*---  Macros definition  ---*/
/*---------------------------*/

#define FASP_VERSION     2.0
#define MULTI_COLOR_ORDER OFF
#define DLMALLOC         OFF
#define NEDMALLOC        OFF
// When this flag is OFF, do not force C1 criterion for the classical AMG method
#define RS_C1            ON
// When this flag is ON, the matrix rows will be reordered as diagonal entries first
#define DIAGONAL_PREF    OFF
#define SHORT            short
#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))
#define MIN(a,b)  (((a)<(b))?(a):(b))
#define ABS(a)    (((a)>=0.0)?(a):-(a))
```

```
#define GT(a,b)  (((a)>(b))?(TRUE):(FALSE))
#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))
#define PUT_REAL(A) printf("### DEBUG: %s = %e\n", #A, (A))
/*----------------------------*/
/*---  Matrix and vector  ---*/
/*----------------------------*/

typedef struct ddenmat{

    INT row;

    INT col;

    REAL **val;

} ddenmat;
typedef struct idenmat{

    INT row;

    INT col;

    INT **val;

} idenmat;
typedef struct dCSRmat{

    INT row;

    INT col;

    INT nnz;

    INT *IA;

    INT *JA;

    REAL *val;

#if MULTI_COLOR_ORDER
    INT color;
    INT *IC;
    INT *ICMAP;
#endif

} dCSRmat;
typedef struct iCSRmat{

    INT row;

    INT col;

    INT nnz;

    INT *IA;

    INT *JA;

    INT *val;

} iCSRmat;
typedef struct dCOOmat{

    INT row;

    INT col;

    INT nnz;

    INT *rowind;

    INT *colind;

    REAL *val;

} dCOOmat;
typedef struct iCOOmat{

    INT row;

    INT col;

    INT nnz;

    INT *I;

    INT *J;

    INT *val;

} iCOOmat;
typedef struct dCSRLmat{

    INT row;

    INT col;

    INT nnz;

    INT dif;

    INT *nz_diff;

    INT *index;

    INT *start;

    INT *ja;

    REAL *val;

} dCSRLmat;
typedef struct dSTRmat{

    INT nx;

    INT ny;

    INT nz;

    INT nxy;

    INT nc;

    INT ngrid;

    REAL *diag;

    INT nband;

    INT *offsets;

    REAL **offdiag;

} dSTRmat;
typedef struct dvector{

    INT row;

    REAL *val;

} dvector;
typedef struct ivector{

    INT row;

    INT *val;

} ivector;
/*---------------------------*/
/*--- Parameter structures --*/
/*---------------------------*/

typedef struct {

    SHORT print_level;
    SHORT itsolver_type;
    SHORT decoup_type;
    SHORT precond_type;
    SHORT stop_type;
    INT   restart;
    INT   maxit;
    REAL  tol;
} ITS_param;
typedef struct {

    SHORT print_level;

    SHORT ILU_type;

    INT ILU_lfil;

    REAL ILU_droptol;

    REAL ILU_relax;

    REAL ILU_permtol;

} ILU_param;
typedef struct {

    SHORT print_level;

    SHORT SWZ_type;

    INT SWZ_maxlvl;

    INT SWZ_mmsize;

    INT SWZ_blksolver;

} SWZ_param;
typedef struct {

    SHORT AMG_type;

    SHORT print_level;

    INT maxit;

    REAL tol;

    SHORT max_levels;

    INT coarse_dof;

    SHORT cycle_type;

    REAL quality_bound;

    SHORT smoother;

    SHORT smooth_order; // 1:  nature order 2:  C/F order (both are symmetric)

    SHORT presmooth_iter;

    SHORT postsmooth_iter;

    REAL relaxation;

    SHORT polynomial_degree;

    SHORT coarse_solver;

    SHORT coarse_scaling;

    SHORT amli_degree;

    REAL *amli_coef;

    SHORT nl_amli_krylov_type;

    SHORT coarsening_type;

    SHORT aggregation_type;

    SHORT interpolation_type;

    REAL strong_threshold;

    REAL max_row_sum;

    REAL truncation_threshold;

    INT aggressive_level;

    INT aggressive_path;

    INT pair_number;

    REAL strong_coupled;

    INT max_aggregation;

    REAL tentative_smooth;

    SHORT smooth_filter;

    SHORT smooth_restriction;

    SHORT ILU_levels;

    SHORT ILU_type;

    INT ILU_lfil;

    REAL ILU_droptol;

    REAL ILU_relax;

    REAL ILU_permtol;

    INT SWZ_levels;

    INT SWZ_mmsize;

    INT SWZ_maxlvl;

    INT SWZ_type;

    INT SWZ_blksolver;

} AMG_param;
/*---------------------------*/
/*--- Work data structures --*/
/*---------------------------*/

typedef struct {

#if WITH_MUMPS
    DMUMPS_STRUC_C id;
#endif

    INT job;

} Mumps_data;
typedef struct {

    void *pt[64];

#if WITH_PARDISO
    MKL_INT iparm[64];

    MKL_INT mtype;

    MKL_INT maxfct;

    MKL_INT mnum;

#endif

} Pardiso_data;
typedef struct {

    dCSRmat *A;

    INT type;

    INT row;

    INT col;

    INT nzlu;

    INT *ijlu;

    REAL *luval;

    INT nb;

    INT nwork;

    REAL *work;

    INT *iperm;
    // iperm[0:n-1]   = old indices of unknowns
    // iperm[n:2*n-1] = reverse permutation = new indices.

    INT ncolors;

    INT *ic;

    INT *icmap;

    INT *uptr;

    INT nlevL;

    INT nlevU;

    INT *ilevL;

    INT *ilevU;

    INT *jlevL;

    INT *jlevU;

} ILU_data;
typedef struct {

    /* matrix information */

    dCSRmat A;  // note:  must start from 1!!  Change later

    /* blocks information */

    INT nblk;

    INT *iblock;

    INT *jblock;

    REAL *rhsloc;

    dvector rhsloc1;

    dvector xloc1;

    REAL *au;

    REAL *al;

    INT SWZ_type;

    INT blk_solver;

    INT memt;

    INT *mask;

    INT maxbs;

    INT *maxa;

    dCSRmat *blk_data;

#if WITH_UMFPACK
    void **numeric;
#endif

```

```
#if WITH_MUMPS
    DMUMPS_STRUC_C *id;
#endif

    Mumps_data *mumps;

    SWZ_param *swzparam;

} SWZ_data;
typedef struct {

    /* Level information */

    SHORT max_levels;

    SHORT num_levels;

    /* Problem information */

    dCSRmat A;

    dCSRmat R;

    dCSRmat P;

    dvector b;

    dvector x;

    /* Extra information */

    void *Numeric;

    Pardiso_data pdata;

    ivector cfmark;

    INT ILU_levels;

    ILU_data LU;

    INT near_kernel_dim;

    REAL **near_kernel_basis;

    // Smoother order information

    INT SWZ_levels;

    SWZ_data Schwarz;

    dvector w;

    Mumps_data mumps;

    INT cycle_type;

    INT *ic;

    INT *icmap;

    INT colors;

    REAL weight;

#if MULTI_COLOR_ORDER
    REAL GS_Theta;
#endif

} AMG_data;
typedef struct {

    SHORT AMG_type;

    SHORT print_level;

    INT maxit;

    SHORT max_levels;

    REAL tol;

    SHORT cycle_type;

    SHORT smoother;

    SHORT smooth_order;

    SHORT presmooth_iter;

    SHORT postsmooth_iter;

    REAL relaxation;

    SHORT polynomial_degree;

    SHORT coarsening_type;

    SHORT coarse_solver;

    SHORT coarse_scaling;

    SHORT amli_degree;

    SHORT nl_amli_krylov_type;

    REAL tentative_smooth;

    REAL *amli_coef;

    AMG_data *mgl_data;

    ILU_data *LU;

    dCSRmat *A;

    // extra near kernel space

    dCSRmat *A_nk;

    dCSRmat *P_nk;

    dCSRmat *R_nk;

    // temporary work space

    dvector r;

    REAL *w;

} precond_data;
typedef struct {

    SHORT AMG_type;

    SHORT print_level;

    INT maxit;

    SHORT max_levels;

    REAL tol;

    SHORT cycle_type;

    SHORT smoother;

    SHORT presmooth_iter;

    SHORT postsmooth_iter;

    SHORT coarsening_type;

    REAL relaxation;

    SHORT coarse_scaling;

    AMG_data *mgl_data;

    ILU_data *LU;

    SHORT scaled;

    dCSRmat *A;

    dSTRmat *A_str;

    dSTRmat *SS_str;

    // data for GS/block GS smoothers (STR format)

    dvector *diaginv;

    ivector *pivot;

    dvector *diaginvS;

    ivector *pivotS;

    ivector *order;

    ivector *neigh;

    // temporary work space

    dvector r;

    REAL *w;

} precond_data_str;
typedef struct {

    INT nc;

    dvector diag;

} precond_diag_str;
typedef struct {

    void *data;

    void (*fct)(REAL *, REAL *, void *);

} precond;
typedef struct {

    void *data;

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

} mxv_matfree;
typedef struct {

    // output flags
    SHORT print_level;
    SHORT output_type;
    // problem parameters
    char inifile[STRLEN];
    char workdir[STRLEN];
    INT  problem_num;
    // parameters for iterative solvers
    SHORT solver_type;
    SHORT decoup_type;
    SHORT precond_type;
    SHORT stop_type;
    REAL itsolver_tol;
    INT itsolver_maxit;
    INT restart;
    // parameters for ILU
    SHORT ILU_type;
    INT ILU_lfil;
    REAL ILU_droptol;
    REAL ILU_relax;
    REAL ILU_permtol;
    // parameter for Schwarz
    INT SWZ_mmsize;
    INT SWZ_maxlvl;
    INT SWZ_type;
    INT SWZ_blksolver;
    // parameters for AMG
    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;
    // parameters for classical AMG
    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;
    //  parameters for smoothed aggregation AMG
    REAL AMG_strong_coupled;
    INT AMG_max_aggregation;
    REAL AMG_tentative_smooth;
    SHORT AMG_smooth_filter;
    SHORT AMG_smooth_restriction;
} input_param;
/*
* OpenMP definitions and declarations
*/
#ifdef _OPENMP

#include "omp.h"

#define ILU_MC_OMP       OFF
//extern INT omp_count;   /**< Counter for multiple calls:  Remove later!!!  --Chensong  */

extern INT THDs_AMG_GS;
extern INT THDs_CPR_lGS;
extern INT THDs_CPR_gGS;
#ifdef DETAILTIME
extern REAL total_linear_time;
extern REAL total_start_time;
extern REAL total_setup_time;
extern INT  total_iter;
extern INT  fasp_called_times;
#endif

#endif /* end if for _OPENMP */

#endif /* end if for __FASP_HEADER__ */

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 diagnal 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.

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.

```
#include "fasp.h"

#ifndef __FASPBLOCK_HEADER__       /*-- allow multiple inclusions --*/
#define __FASPBLOCK_HEADER__
/*---------------------------*/
/*---   Data structures   ---*/
/*---------------------------*/

typedef struct dBSRmat {

    INT ROW;

    INT COL;

    INT NNZ;

    INT nb; // NOTE: for the moment, allow nb*nb full block

    INT storage_manner; // 0:  row-major order, 1:  column-major order

    REAL *val;

    INT *IA;

    INT *JA;

} dBSRmat;
typedef struct dBLCmat {

    INT brow;

    INT bcol;

    dCSRmat **blocks;

} dBLCmat;
typedef struct iBLCmat {

    INT brow;

    INT bcol;

    iCSRmat **blocks;

} iBLCmat;
typedef struct block_dvector {

    INT brow;

    dvector **blocks;

} block_dvector;
typedef struct block_ivector {

    INT brow;

    ivector **blocks;

} block_ivector;
/*---------------------------*/
/*--- Parameter structures --*/
/*---------------------------*/

typedef struct {

    INT max_levels;

    INT num_levels;

    dBSRmat A;

    dBSRmat R;

    dBSRmat P;

    dvector b;

    dvector x;

    dvector diaginv;

    dCSRmat Ac;

    void *Numeric;

    Pardiso_data pdata;

    dCSRmat PP;

    REAL *pw;

    dBSRmat SS;

    REAL *sw;

    dvector diaginv_SS;

    ILU_data PP_LU;

    ivector cfmark;

    INT ILU_levels;

    ILU_data LU;

    INT near_kernel_dim;

    REAL **near_kernel_basis;

    //----------------------------------------
    // extra near kernal space for extra solve

    dCSRmat *A_nk;

    dCSRmat *P_nk;

    dCSRmat *R_nk;
    //----------------------------------------

    dvector w;

    Mumps_data mumps;

} AMG_data_bsr;
typedef struct {

    INT nb;

    dvector diag;

} precond_diag_bsr;
typedef struct {

    SHORT AMG_type;

    SHORT print_level;

    INT maxit;

    INT max_levels;

    REAL tol;

    SHORT cycle_type;

    SHORT smoother;

    SHORT smooth_order;

    SHORT presmooth_iter;

    SHORT postsmooth_iter;

    SHORT coarsening_type;

    REAL relaxation;

    SHORT coarse_solver;

    SHORT coarse_scaling;

    SHORT amli_degree;

    REAL *amli_coef;

    REAL tentative_smooth;

    SHORT nl_amli_krylov_type;

    AMG_data_bsr *mgl_data;

    AMG_data *pres_mgl_data;

    ILU_data *LU;

    dBSRmat *A;

    // extra near kernal space

    dCSRmat *A_nk;

    dCSRmat *P_nk;

    dCSRmat *R_nk;

    dvector r;

    REAL *w;

} precond_data_bsr;
typedef struct {

    /*-------------------------------------*/
    /* Basic data for block preconditioner */
    /*-------------------------------------*/
    dBLCmat *Ablc;
    dCSRmat *A_diag;
    dvector r;
    /*-------------------------------*/
    /* Data for the diagonal blocks */
    /*-------------------------------*/

    /*--- solve by direct solver ---*/
    void **LU_diag;
    /*--- solve by AMG ---*/
    AMG_data **mgl;
    AMG_param *amgparam;
} precond_data_blc;
typedef struct {

    INT NumLayers;
    dBLCmat *A;
    dBLCmat *Ai;
    dCSRmat *local_A;
    void **local_LU;
    ivector *local_index;
    // temprary work spaces
    dvector r;
    REAL *w;
} precond_data_sweeping;
#endif /* end if for __FASPBLOCK_HEADER__ */

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 ILUt 2
- #define ILUtp 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.

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 precond 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 precond
Definition at line 140 of file fasp_const.h.

9.17.2.83 PREC_DIAG

```
#define PREC_DIAG 1
```

with diagonal precond
Definition at line 139 of file fasp_const.h.

9.17.2.84 PREC_FMG

```
#define PREC_FMG 3
```

with full AMG precond
Definition at line 141 of file fasp_const.h.

9.17.2.85 PREC_ILU

```
#define PREC_ILU 4
```

with ILU precond
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 precond
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.

```
#ifndef __FASP_CONST__             /*-- allow multiple inclusions --*/
#define __FASP_CONST__

#define FASP_SUCCESS            0
//---------------------------------------------------------------------------------
#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
#define FALSE                   0
#define ON                      1
#define OFF                     0
#define PRINT_NONE              0
#define PRINT_MIN               1
#define PRINT_SOME              2
#define PRINT_MORE              4
#define PRINT_MOST              8
#define PRINT_ALL              10
#define MAT_FREE                0
//---------------------------------------------------------------------------------
#define MAT_CSR                 1
#define MAT_BSR                 2
#define MAT_STR                 3
#define MAT_CSRL                6
#define MAT_SymCSR              7
#define MAT_BLC                 8
//---------------------------------------------------------------------------------
//    For bordered systems in reservoir simulation
//---------------------------------------------------------------------------------
#define MAT_bCSR               11
#define MAT_bBSR               12
#define MAT_bSTR               13
#define SOLVER_DEFAULT          0
```

```
//--------------------------------------------------------------------------------
#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
#define STOP_REL_PRECRES        2
#define STOP_MOD_REL_RES        3
#define PREC_NULL               0
#define PREC_DIAG               1
#define PREC_AMG                2
#define PREC_FMG                3
#define PREC_ILU                4
#define PREC_SCHWARZ            5
#define ILUk                    1
#define ILUt                    2
#define ILUtp                   3
#define SCHWARZ_FORWARD         1
#define SCHWARZ_BACKWARD        2
#define SCHWARZ_SYMMETRIC       3
#define CLASSIC_AMG             1
#define SA_AMG                  2
#define UA_AMG                  3
#define PAIRWISE                1
#define VMB                     2
#define NPAIR                   3
#define SPAIR                   4
#define V_CYCLE                 1
#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
#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
#define SMOOTHER_SPETEN        19
#define COARSE_RS               1
#define COARSE_RSP              2
#define COARSE_CR               3
#define COARSE_AC               4
#define COARSE_MIS              5
#define INTERP_DIR              1
#define INTERP_STD              2
#define INTERP_ENG              3
#define INTERP_EXT              6
#define G0PT                   -5
#define UNPT                   -1
#define FGPT                    0
#define CGPT                    1
#define ISPT                    2
#define NO_ORDER                0
#define CF_ORDER                1
```

```
#define USERDEFINED            0
#define CPFIRST                1
#define FPFIRST               -1
#define ASCEND                12
#define DESCEND               21
#define BIGREAL            1e+20
#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
#endif                             /* end if for __FASP_CONST__ */

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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.

```
#ifndef __FASPGRID_HEADER__  /*-- allow multiple inclusions --*/
#define __FASPGRID_HEADER__
typedef struct grid2d {

    REAL (*p)[2];
    INT (*e)[2];
    INT (*t)[3];
    INT (*s)[3];
    INT *pdiri;
    INT *ediri;
    INT *pfather;
    INT *efather;
    INT *tfather;
    INT vertices;
    INT edges;
    INT triangles;
} grid2d;
typedef grid2d *pgrid2d;
typedef const grid2d *pcgrid2d;
#endif /* end if for __FASPGRID_HEADER__ */

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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

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

n	Number of variables
x	Pointer to the vector
val	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

n	Number of variables
x	Pointer to the original vector
y	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

n	Number of variables
x	Pointer to the vector
val	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.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_darray_set (const INT   n,
                      REAL       *x,
                      const REAL  val)
{
    SHORT use_openmp = FALSE;

#ifdef _OPENMP
    INT nthreads = 1;

    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (val == 0.0) {
        if (use_openmp) {
#ifdef _OPENMP
            INT mybegin,myend,myid;
#pragma omp parallel for private(myid,mybegin,myend)
            for (myid = 0; myid < nthreads; myid ++) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                memset(&x[mybegin], 0x0, sizeof(REAL)*(myend-mybegin));
            }
#endif
        }
        else
            memset(x, 0x0, sizeof(REAL)*n);
    }
    else {
        INT i;

        if (use_openmp) {
#ifdef _OPENMP
            INT mybegin,myend,myid;
#pragma omp parallel for private(myid,mybegin,myend,i)
            for (myid = 0; myid < nthreads; myid ++) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                for (i=mybegin; i<myend; ++i) x[i]=val;
            }
#endif
        }
        else {
            for (i=0; i<n; ++i) x[i] = val;
        }
    }
}

void fasp_iarray_set (const INT   n,
                      INT        *x,
                      const INT   val)
{
    SHORT use_openmp = FALSE;

#ifdef _OPENMP
    INT nthreads = 1;

    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (val == 0) {
        if (use_openmp) {
#ifdef _OPENMP
            INT mybegin,myend,myid;
#pragma omp parallel for private(myid, mybegin, myend)
            for (myid = 0; myid < nthreads; myid ++) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                memset(&x[mybegin], 0, sizeof(INT)*(myend-mybegin));
            }
#endif
        }
        else {
            memset(x, 0, sizeof(INT)*n);
        }
    }
    else {
        INT i;

        if (use_openmp) {
#ifdef _OPENMP
            INT mybegin,myend,myid;
#pragma omp parallel for private(myid, mybegin, myend,i)
            for (myid = 0; myid < nthreads; myid ++) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                for (i=mybegin; i<myend; ++i) x[i]=val;
            }
#endif
        }
        else {
            for (i=0; i<n; ++i) x[i]=val;
        }
    }
}

void fasp_darray_cp (const INT    n,
                     const REAL  *x,
                     REAL        *y)
{
    memcpy(y, x, n*sizeof(REAL));
}


void fasp_iarray_cp (const INT    n,
                     const INT   *x,
                     INT         *y)
{
    memcpy(y, x, n*sizeof(INT));
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

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

bytes	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

Unsigend long ineger 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

Unsigend long ineger 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.

```
#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

unsigned long fasp_aux_change_endian4 (const unsigned long x)
{
    unsigned char *ptr = (unsigned char *)&x;
    return (ptr[0] « 24) | (ptr[1] « 16) | (ptr[2] « 8) | ptr[3];
}

double fasp_aux_change_endian8 (const double x)
{
    double dbl;
    unsigned char *bytes, *buffer;

    buffer=(unsigned char *)&dbl;
    bytes=(unsigned char *)&x;

    buffer[0]=bytes[7];
    buffer[1]=bytes[6];
    buffer[2]=bytes[5];
    buffer[3]=bytes[4];
    buffer[4]=bytes[3];
    buffer[5]=bytes[2];
    buffer[6]=bytes[1];
    buffer[7]=bytes[0];
    return dbl;
}

double fasp_aux_bbyteToldouble (const unsigned char bytes[])
{
    double dbl;
    unsigned char *buffer;
    buffer=(unsigned char *)&dbl;
    buffer[0]=bytes[7];
    buffer[1]=bytes[6];
    buffer[2]=bytes[5];
    buffer[3]=bytes[4];
    buffer[4]=bytes[3];
    buffer[5]=bytes[2];
    buffer[6]=bytes[1];
    buffer[7]=bytes[0];
    return dbl;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.25 AuxGivens.c File Reference

Givens transformation.

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

Functions

- void fasp_aux_givens (const REAL beta, const dCSRmat *H, dvector *y, REAL *work)

  *Perform Givens rotations to compute y |beta*e_1- H*y|.*

9.25.1 Detailed Description

Givens transformation.

Note

This file contains Level-0 (Aux) functions.

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 |beta∗e_1- H∗y|.

Parameters

beta	Norm of residual r_0
H	Upper Hessenberg dCSRmat matrix: (m+1)∗m
y	Minimizer of |beta∗e_1- H∗y|
work	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_aux_givens (const REAL      beta,
                      const dCSRmat   *H,
                         dvector        *y,
                         REAL           *work)
{
    const INT  Hsize = H->row;
    INT        i, j, istart, idiag, ip1start;
    REAL       h0, h1, r, c, s, tempi, tempip1, sum;

    memset(&work, 0x0, sizeof(REAL)*Hsize);
    work[0] = beta;

    for ( i=0; i<Hsize-1; ++i ) {
        istart   = H->IA[i];
        ip1start = H->IA[i+1];
        if (i==0) idiag = istart;
        else idiag = istart+1;

        h0 = H->val[idiag];      // h0=H[i][i]
        h1 = H->val[H->IA[i+1]]; // h1=H[i+1][i]
        r  = sqrt(h0*h0+h1*h1);
        c  = h0/r; s = h1/r;

        for ( j=idiag; j<ip1start; ++j ) {
            tempi   = H->val[j];
            tempip1 = H->val[ip1start+(j-idiag)];
            H->val[j] = c*tempi+s*tempip1;
            H->val[ip1start+(j-idiag)] = c*tempip1-s*tempi;
        }

        tempi   = c*work[i]+s*work[i+1];
        tempip1 = c*work[i+1]-s*work[i];

        work[i] = tempi; work[i+1]=tempip1;
    }

    for ( i = Hsize-2; i >= 0; --i ) {
        sum = work[i];
        istart = H->IA[i];
        if (i==0) idiag = istart;
        else idiag = istart+1;

        for ( j=Hsize-2; j>i; --j ) sum-=H->val[idiag+j-i]*y->val[j];

        y->val[i] = sum/H->val[idiag];
    }

}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.27 AuxGraphics.c File Reference

Graphical output for CSR matrix.

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

---

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

A	Pointer to the dBSRmat matrix
fname	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

A	Pointer to the dBSRmat matrix
filename	File name
size	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

A	Pointer to the dBSRmat matrix
fname	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

A	Pointer to the dCSRmat matrix
filename	File name
size	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

pg	Pointer to grid in 2d
level	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.

```
#include <math.h>

#include "fasp.h"
#include "fasp_grid.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static void put_byte (FILE *fp, const int c);
static void put_word (FILE *fp, const int w);
static void put_dword (FILE *fp, const int d);
static int write_bmp16 (const char *fname, int m, int n, const char map[]);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_dcsr_subplot (const dCSRmat  *A,
                        const char     *filename,
                        int             size)
{
    INT m = A->row, n = A->col, minmn = MIN(m,n);
    int i, j, k;
    char *map;

    if ( size>minmn ) size = minmn;
    map = (char *)fasp_mem_calloc(size * size, sizeof(char));

    printf("Writing matrix pattern to '%s'...\n",filename);

    memset((void *)map, 0x0F, size * size);

    for (i = 0; i < size; ++i) {
        for (j = A->IA[i]; j < A->IA[i+1]; ++j) {
            if (A->JA[j]<size) {
                k = size*i + A->JA[j];
                if (map[k] != 0x0F)
                    map[k] = 0x0F;
                else if (A->val[j] > 1e-20)
                    map[k] = 0x09; /* bright blue */
                else if (A->val[j] < -1e-20)
                    map[k] = 0x0C; /* bright red */
                else
                    map[k] = 0x06; /* brown */
            } // end if
        } // end for j
    } // end for i

    write_bmp16(filename, size, size, map);

    fasp_mem_free(map); map = NULL;
}

void fasp_dcsr_plot (const dCSRmat  *A,
                     const char     *fname)
{
    FILE *fp;
    INT offset, bmsize, i, j, b;
    INT n = A->col, m = A->row;
    INT size;

    INT col;
    REAL val;
    char *map;

    size = ( (n+7)/8 )*8;

    map = (char *)fasp_mem_calloc(size, sizeof(char));

    memset(map, 0x0F, size);

    if (!(1 <= m && m <= 32767))
        printf("### ERROR: Invalid num of rows %d!  [%s]\n", m, __FUNCTION__);

    if (!(1 <= n && n <= 32767))
        printf("### ERROR: Invalid num of cols %d!  [%s]\n", n, __FUNCTION__);

    fp = fopen(fname, "wb");
    if (fp == NULL) {
        printf("### ERROR: Unable to create `%s'!  [%s]\n", fname, __FUNCTION__);
        goto FINISH;
    }

    offset = 14 + 40 + 16 * 4;
    bmsize = (4 * n + 31) / 32;
    /* struct BMPFILEHEADER (14 bytes) */
    /* UINT bfType */          put_byte(fp, 'B'); put_byte(fp, 'M');
    /* DWORD bfSize */         put_dword(fp, offset + bmsize * 4);
    /* UINT bfReserved1 */     put_word(fp, 0);
    /* UNIT bfReserved2 */     put_word(fp, 0);
    /* DWORD bfOffBits */      put_dword(fp, offset);
    /* struct BMPINFOHEADER (40 bytes) */
    /* DWORD biSize */         put_dword(fp, 40);
    /* LONG biWidth */         put_dword(fp, n);
    /* LONG biHeight */        put_dword(fp, m);
    /* WORD biPlanes */        put_word(fp, 1);
    /* WORD biBitCount */      put_word(fp, 4);
    /* DWORD biCompression */  put_dword(fp, 0 /* BI_RGB */);
    /* DWORD biSizeImage */    put_dword(fp, 0);
    /* LONG biXPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* LONG biYPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* DWORD biClrUsed */      put_dword(fp, 0);
    /* DWORD biClrImportant */ put_dword(fp, 0);
    /* struct RGBQUAD (16 * 4 = 64 bytes) */
    /* CGA-compatible colors:  */
    /* 0x00 = black */         put_dword(fp, 0x000000);
    /* 0x01 = blue */          put_dword(fp, 0x000080);
    /* 0x02 = green */         put_dword(fp, 0x008000);
    /* 0x03 = cyan */          put_dword(fp, 0x008080);
    /* 0x04 = red */           put_dword(fp, 0x800000);
    /* 0x05 = magenta */       put_dword(fp, 0x800080);
    /* 0x06 = brown */         put_dword(fp, 0x808000);
    /* 0x07 = light gray */    put_dword(fp, 0xC0C0C0);
    /* 0x08 = dark gray */     put_dword(fp, 0x808080);
    /* 0x09 = bright blue */   put_dword(fp, 0x0000FF);
    /* 0x0A = bright green */  put_dword(fp, 0x00FF00);
    /* 0x0B = bright cyan */   put_dword(fp, 0x00FFFF);
    /* 0x0C = bright red */    put_dword(fp, 0xFF0000);
    /* 0x0D = bright magenta */put_dword(fp, 0xFF00FF);
    /* 0x0E = yellow */        put_dword(fp, 0xFFFF00);
    /* 0x0F = white */         put_dword(fp, 0xFFFFFF);
    /* pixel data bits */
    b = 0;

    //  for(i=((m+7)/8)*8 - 1; i>=m; i--){
    //      memset(map, 0x0F, size);
    //        for (j = 0; j < size; ++j) {
    //            b «= 4;
    //            b |= (j < n ?  map[j] & 15 :  0);
    //            if (j & 1) put_byte(fp, b);
    //        }
    //  }

    for ( i = A->row-1; i >=0; i-- ) {
        memset(map, 0x0F, size);

        for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
            col =  A->JA[j];
            val =  A->val[j];
            if (map[col] != 0x0F)
                map[col] = 0x0F;
            else if ( val > 1e-20)
                map[col] = 0x09; /* bright blue */
            else if ( val < -1e-20)
                map[col] = 0x0C; /* bright red */
            else if (val == 0)
                map[col] = 0x00; /* bright red */
            else
                map[col] = 0x06; /* brown */
        } // for j

        for (j = 0; j < size; ++j) {
            b «= 4;
            b |= (j < n ?  map[j] & 15 :  0);
            if (j & 1) put_byte(fp, b);
        }
    }

    fflush(fp);
    if (ferror(fp)) {
        printf("### ERROR: Write error on ‘%s’!  [%s]\n", fname, __FUNCTION__);
    }

FINISH: if (fp != NULL) fclose(fp);

    fasp_mem_free(map); map = NULL;
}

void fasp_dbsr_subplot (const dBSRmat  *A,
                        const char     *filename,
                        int             size)
{
    INT m = A->ROW;
    INT n = A->COL;
    INT nb = A->nb;
    INT nb2 = nb*nb;
    INT offset;
    INT row, col, i, j, k, l, minmn=nb*MIN(m,n);
    REAL val;
    char *map;

    if (size>minmn) size=minmn;

    printf("Writing matrix pattern to ‘%s’...\n",filename);

    map = (char *)fasp_mem_calloc(size * size, sizeof(char));

    memset((void *)map, 0x0F, size * size);

    for ( i = 0; i < size/nb; i++ ) {

        for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
            for ( k = 0; k < A->nb; k++ ) {
                for ( l = 0; l < A->nb; l++ ) {

                    row = i*nb + k;
                    col =  A->JA[j]*nb + l;
                    val = A->val[ A->JA[j]*nb2 + k*nb + l];

                    if (col<size) {

                        offset = size*row + col;

                        if (map[offset] != 0x0F)
                            map[offset] = 0x0F;
                        else if ( val > 1e-20)
                            map[offset] = 0x09; /* bright blue */
                        else if ( val < -1e-20)
                            map[offset] = 0x0C; /* bright red */
                        else if (val == 0)
                            map[offset] = 0x00; /* bright red */
                        else
                            map[offset] = 0x06; /* brown */
                    } // end if
                }
            }
        }
    }

    write_bmp16(filename, size, size, map);

    fasp_mem_free(map); map = NULL;
```

```
}

void fasp_dbsr_plot (const dBSRmat  *A,
                     const char     *fname)
{
    FILE *fp;
    INT offset, bmsize, i, j, b;
    INT size;
    INT nb = A->nb;
    INT nb2 = nb*nb;
    INT n = A->COL*A->nb, m = A->ROW*A->nb;
    INT col,k,l;
    REAL val;
    char *map;

    size = ( (n+7)/8 )*8;

    map = (char *)fasp_mem_calloc(size, sizeof(char));

    memset((void *)map, 0x0F, size);

    if (!(1 <= m && m <= 32767))
        printf("### ERROR: Invalid num of rows %d!  [%s]\n", m, __FUNCTION__);

    if (!(1 <= n && n <= 32767))
        printf("### ERROR: Invalid num of cols %d!  [%s]\n", n, __FUNCTION__);

    fp = fopen(fname, "wb");
    if (fp == NULL) {
        printf("### ERROR: Unable to create '%s'!  [%s]\n", fname, __FUNCTION__);
        goto FINISH;
    }

    offset = 14 + 40 + 16 * 4;
    bmsize = (4 * n + 31) / 32;
    /* struct BMPFILEHEADER (14 bytes) */
    /* UINT bfType */          put_byte(fp, 'B'); put_byte(fp, 'M');
    /* DWORD bfSize */         put_dword(fp, offset + bmsize * 4);
    /* UINT bfReserved1 */     put_word(fp, 0);
    /* UNIT bfReserved2 */     put_word(fp, 0);
    /* DWORD bfOffBits */      put_dword(fp, offset);
    /* struct BMPINFOHEADER (40 bytes) */
    /* DWORD biSize */         put_dword(fp, 40);
    /* LONG biWidth */         put_dword(fp, n);
    /* LONG biHeight */        put_dword(fp, m);
    /* WORD biPlanes */        put_word(fp, 1);
    /* WORD biBitCount */      put_word(fp, 4);
    /* DWORD biCompression */  put_dword(fp, 0 /* BI_RGB */);
    /* DWORD biSizeImage */    put_dword(fp, 0);
    /* LONG biXPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* LONG biYPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* DWORD biClrUsed */      put_dword(fp, 0);
    /* DWORD biClrImportant */ put_dword(fp, 0);
    /* struct RGBQUAD (16 * 4 = 64 bytes) */
    /* CGA-compatible colors:  */
    /* 0x00 = black */         put_dword(fp, 0x000000);
    /* 0x01 = blue */          put_dword(fp, 0x000080);
    /* 0x02 = green */         put_dword(fp, 0x008000);
    /* 0x03 = cyan */          put_dword(fp, 0x008080);
    /* 0x04 = red */           put_dword(fp, 0x800000);
    /* 0x05 = magenta */       put_dword(fp, 0x800080);
    /* 0x06 = brown */         put_dword(fp, 0x808000);
    /* 0x07 = light gray */    put_dword(fp, 0xC0C0C0);
    /* 0x08 = dark gray */     put_dword(fp, 0x808080);
    /* 0x09 = bright blue */   put_dword(fp, 0x0000FF);
    /* 0x0A = bright green */  put_dword(fp, 0x00FF00);
    /* 0x0B = bright cyan */   put_dword(fp, 0x00FFFF);
    /* 0x0C = bright red */    put_dword(fp, 0xFF0000);
    /* 0x0D = bright magenta */put_dword(fp, 0xFF00FF);
    /* 0x0E = yellow */        put_dword(fp, 0xFFFF00);
    /* 0x0F = white */         put_dword(fp, 0xFFFFFF);
    /* pixel data bits */
    b = 0;

    //  for(i=size-1; i>=m; i--){
    //      memset(map, 0x0F, size);
    //        for (j = 0; j < size; ++j) {
    //            b «= 4;
    //            b |= (j < n ?  map[j] & 15 :  0);
    //            if (j & 1) put_byte(fp, b);
    //        }
    //  }

    for ( i = A->ROW-1; i >=0; i-- ) {

        for ( k = A->nb-1; k >=0; k-- ) {

            memset(map, 0x0F, size);

            for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
                for ( l = 0; l < A->nb; l++ ) {

                    col =  A->JA[j]*nb + l;
                    val = A->val[ A->JA[j]*nb2 + k*nb + l];

                    if (map[col] != 0x0F)
                        map[col] = 0x0F;
                    else if ( val > 1e-20)
                        map[col] = 0x09; /* bright blue */
                    else if ( val < -1e-20)
                        map[col] = 0x0C; /* bright red */
                    else if (val == 0)
                        map[col] = 0x00; /* bright red */
                    else
                        map[col] = 0x06; /* brown */
                } // for l
            } // for j


            for (j = 0; j < size; ++j) {
                b «= 4;
                b |= (j < n ?  map[j] & 15 :  0);
                if (j & 1) put_byte(fp, b);
            }

        }
    }

    fflush(fp);
    if (ferror(fp)) {
        printf("### ERROR: Write error on ‘%s’!  [%s]\n", fname, __FUNCTION__);
    }

FINISH: if (fp != NULL) fclose(fp);

    fasp_mem_free(map); map = NULL;
}

void fasp_grid2d_plot (pgrid2d  pg,
                       int      level)
{
    FILE *datei;
    char buf[120];
    INT i;
    REAL xmid,ymid,xc,yc;

    sprintf(buf,"Grid_ref_level%d.eps",level);
    datei = fopen(buf,"w");
    if(datei==NULL) {
        printf("Opening file %s fails!\n", buf);
        return;
    }

    fprintf(datei, "%%!PS-Adobe-2.0-2.0 EPSF-2.0\n");
    fprintf(datei, "%%%%BoundingBox:  0 0 550 550\n");
    fprintf(datei, "25 dup translate\n");
    fprintf(datei, "%f setlinewidth\n",0.2);
    fprintf(datei, "/Helvetica findfont %f scalefont setfont\n",64.0*pow(0.5,level));
    fprintf(datei, "/b{0 setgray} def\n");
    fprintf(datei, "/r{1.0 0.6 0.6  setrgbcolor} def\n");
    fprintf(datei, "/u{0.1 0.7 0.1  setrgbcolor} def\n");
    fprintf(datei, "/d{0.1 0.1 1.0  setrgbcolor} def\n");
    fprintf(datei, "/cs{closepath stroke} def\n");
    fprintf(datei, "/m{moveto} def\n");
    fprintf(datei, "/l{lineto} def\n");

    fprintf(datei,"b\n");
    for (i=0; i<pg->triangles; ++i) {
        xc = (pg->p[pg->t[i][0]][0]+pg->p[pg->t[i][1]][0]+pg->p[pg->t[i][2]][0])*150.0;
        yc = (pg->p[pg->t[i][0]][1]+pg->p[pg->t[i][1]][1]+pg->p[pg->t[i][2]][1])*150.0;

        xmid = pg->p[pg->t[i][0]][0]*450.0;
        ymid = pg->p[pg->t[i][0]][1]*450.0;
        fprintf(datei,"%.1f %.1f m ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
        xmid = pg->p[pg->t[i][1]][0]*450.0;
        ymid = pg->p[pg->t[i][1]][1]*450.0;
        fprintf(datei,"%.1f %.1f l ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
        xmid = pg->p[pg->t[i][2]][0]*450.0;
        ymid = pg->p[pg->t[i][2]][1]*450.0;
        fprintf(datei,"%.1f %.1f l ",0.9*xmid+0.1*xc,0.9*ymid+0.1*yc);
        fprintf(datei,"cs\n");
    }
    fprintf(datei,"r\n");
    for(i=0; i<pg->vertices; ++i) {
        xmid = pg->p[i][0]*450.0;
        ymid = pg->p[i][1]*450.0;
        fprintf(datei,"%.1f %.1f m ",xmid,ymid);
        fprintf(datei,"(%d) show\n ",i);
    }
    fprintf(datei,"u\n");
    for(i=0; i<pg->edges; ++i) {
        xmid = 0.5*(pg->p[pg->e[i][0]][0]+pg->p[pg->e[i][1]][0])*450.0;
        ymid = 0.5*(pg->p[pg->e[i][0]][1]+pg->p[pg->e[i][1]][1])*450.0;
        fprintf(datei,"%.1f %.1f m ",xmid,ymid);
        fprintf(datei,"(%d) show\n ",i);

        xmid = pg->p[pg->e[i][0]][0]*450.0;
        ymid = pg->p[pg->e[i][0]][1]*450.0;
        fprintf(datei,"%.1f %.1f m ",xmid,ymid);
        xmid = pg->p[pg->e[i][1]][0]*450.0;
        ymid = pg->p[pg->e[i][1]][1]*450.0;
        fprintf(datei,"%.1f %.1f l ",xmid,ymid);
        fprintf(datei,"cs\n");
    }
    fprintf(datei,"d\n");
    for(i=0; i<pg->triangles; ++i) {
        xmid = (pg->p[pg->t[i][0]][0]+pg->p[pg->t[i][1]][0]+pg->p[pg->t[i][2]][0])*150.0;
        ymid = (pg->p[pg->t[i][0]][1]+pg->p[pg->t[i][1]][1]+pg->p[pg->t[i][2]][1])*150.0;
        fprintf(datei,"%.1f %.1f m ",xmid,ymid);
        fprintf(datei,"(%d) show\n ",i);
    }
    fprintf(datei, "showpage\n");
    fclose(datei);
}

/*----------------------------------*/
/*--      Private Functions      --*/
/*----------------------------------*/

static void put_byte (FILE       *fp,
                      const int   c)
{
    fputc(c, fp);
    return;
}

static void put_word (FILE       *fp,
                      const int   w)
{ /* big endian */
    put_byte(fp, w);
    put_byte(fp, w » 8);
    return;
}

static void put_dword (FILE       *fp,
                       const int   d)
{ /* big endian */
    put_word(fp, d);
    put_word(fp, d » 16);
    return;
}

static int write_bmp16 (const char   *fname,
                        const int     m,
                        const int     n,
                        const char    map[])
{
    FILE *fp;
    int offset, bmsize, i, j, b, ret = 1;

    if (!(1 <= m && m <= 32767))
        printf("### ERROR: %s invalid height %d\n", __FUNCTION__, m);

    if (!(1 <= n && n <= 32767))
        printf("### ERROR: %s invalid width %d\n", __FUNCTION__, n);

    fp = fopen(fname, "wb");
    if (fp == NULL) {
        printf("### ERROR: %s unable to create ‘%s’\n", __FUNCTION__, fname);
        ret = 0;
        goto FINISH;
    }
    offset = 14 + 40 + 16 * 4;
    bmsize = (4 * n + 31) / 32;
    /* struct BMPFILEHEADER (14 bytes) */
    /* UINT bfType */           put_byte(fp, ’B’); put_byte(fp, ’M’);
    /* DWORD bfSize */          put_dword(fp, offset + bmsize * 4);
    /* UINT bfReserved1 */      put_word(fp, 0);
    /* UINT bfReserved2 */      put_word(fp, 0);
    /* DWORD bfOffBits */       put_dword(fp, offset);
    /* struct BMPINFOHEADER (40 bytes) */
    /* DWORD biSize */          put_dword(fp, 40);
    /* LONG biWidth */          put_dword(fp, n);
    /* LONG biHeight */         put_dword(fp, m);
    /* WORD biPlanes */         put_word(fp, 1);
    /* WORD biBitCount */       put_word(fp, 4);
    /* DWORD biCompression */   put_dword(fp, 0 /* BI_RGB */);
    /* DWORD biSizeImage */     put_dword(fp, 0);
    /* LONG biXPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* LONG biYPelsPerMeter */ put_dword(fp, 2953 /* 75 dpi */);
    /* DWORD biClrUsed */       put_dword(fp, 0);
    /* DWORD biClrImportant */ put_dword(fp, 0);
    /* struct RGBQUAD (16 * 4 = 64 bytes) */
    /* CGA-compatible colors:  */
    /* 0x00 = black */          put_dword(fp, 0x000000);
    /* 0x01 = blue */           put_dword(fp, 0x000080);
    /* 0x02 = green */          put_dword(fp, 0x008000);
    /* 0x03 = cyan */           put_dword(fp, 0x008080);
    /* 0x04 = red */            put_dword(fp, 0x800000);
    /* 0x05 = magenta */        put_dword(fp, 0x800080);
    /* 0x06 = brown */          put_dword(fp, 0x808000);
    /* 0x07 = light gray */     put_dword(fp, 0xC0C0C0);
    /* 0x08 = dark gray */      put_dword(fp, 0x808080);
    /* 0x09 = bright blue */    put_dword(fp, 0x0000FF);
    /* 0x0A = bright green */   put_dword(fp, 0x00FF00);
    /* 0x0B = bright cyan */    put_dword(fp, 0x00FFFF);
    /* 0x0C = bright red */     put_dword(fp, 0xFF0000);
    /* 0x0D = bright magenta */put_dword(fp, 0xFF00FF);
    /* 0x0E = yellow */         put_dword(fp, 0xFFFF00);
    /* 0x0F = white */          put_dword(fp, 0xFFFFFF);
    /* pixel data bits */
    b = 0;
    for (i = m - 1; i >= 0; i--) {
        for (j = 0; j < ((n + 7) / 8) * 8; ++j) {
            b «= 4;
            b |= (j < n ?  map[i * n + j] & 15 :  0);
            if (j & 1) put_byte(fp, b);
        }
    }
    fflush(fp);

    if (ferror(fp)) {
        printf("### ERROR: %s write error on ‘%s’\n", __FUNCTION__, fname);
        ret = 0;
    }

FINISH: if (fp != NULL) fclose(fp);
    return ret;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.29 AuxInput.c File Reference

Read and check input parameters.

```
#include "fasp.h"
#include "fasp_functs.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

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

inparam	Input parameters

Returns

FASP_SUCCESS if successed; 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

fname	File name for input file
inparam	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.

```

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_param_check (input_param  *inparam)
{
    SHORT status = FASP_SUCCESS;

    if ( inparam->problem_num<0
        || inparam->solver_type<0
        || inparam->solver_type>50
        || inparam->precond_type<0
        || inparam->decoup_type<0
        || inparam->itsolver_tol<0
        || inparam->itsolver_maxit<0
        || inparam->stop_type<=0
        || inparam->stop_type>3
        || inparam->restart<0
        || inparam->ILU_type<=0
        || inparam->ILU_type>3
        || inparam->ILU_lfil<0
        || inparam->ILU_droptol<=0
        || inparam->ILU_relax<0
        || inparam->ILU_permtol<0
        || inparam->SWZ_mmsize<0
        || inparam->SWZ_maxlvl<0
        || inparam->SWZ_type<0
        || inparam->SWZ_blksolver<0
        || inparam->AMG_type<=0
        || inparam->AMG_type>3
        || inparam->AMG_cycle_type<=0
        || inparam->AMG_levels<0
        || inparam->AMG_ILU_levels<0
        || inparam->AMG_coarse_dof<=0
        || inparam->AMG_tol<0
        || inparam->AMG_maxit<0
        || inparam->AMG_coarsening_type<=0
        || inparam->AMG_coarsening_type>4
        || inparam->AMG_coarse_solver<0
        || inparam->AMG_interpolation_type<0
        || inparam->AMG_interpolation_type>5
        || inparam->AMG_smoother<0
        || inparam->AMG_smoother>20
        || inparam->AMG_strong_threshold<0.0
        || inparam->AMG_strong_threshold>0.9999
        || inparam->AMG_truncation_threshold<0.0
        || inparam->AMG_truncation_threshold>0.9999
        || inparam->AMG_max_row_sum<0.0
        || inparam->AMG_presmooth_iter<0
        || inparam->AMG_postsmooth_iter<0
        || inparam->AMG_amli_degree<0
        || inparam->AMG_aggressive_level<0
        || inparam->AMG_aggressive_path<0
        || inparam->AMG_aggregation_type<0
        || inparam->AMG_pair_number<0
        || inparam->AMG_strong_coupled<0
        || inparam->AMG_max_aggregation<=0
        || inparam->AMG_tentative_smooth<0
        || inparam->AMG_smooth_filter<0
        || inparam->AMG_smooth_restriction<0
        || inparam->AMG_smooth_restriction>1
        ) status = ERROR_INPUT_PAR;

    return status;
}

void fasp_param_input (const char   *fname,
                       input_param  *inparam)
{
    char     buffer[STRLEN]; // Note:  max number of char for each line!
    int      val;
    SHORT    status = FASP_SUCCESS;
    FILE    *fp;

    // set default input parameters
    fasp_param_input_init(inparam);

    // if input file is not specified, use the default values
    if (fname==NULL) return;

    fp = fopen(fname,"r");
    if (fp==NULL) fasp_chkerr(ERROR_OPEN_FILE, __FUNCTION__);

    while ( status == FASP_SUCCESS ) {
        int     ibuff;
        double  dbuff;
        char    sbuff[STRLEN];

        val = fscanf(fp,"%s",buffer);
        if (val==EOF) break;
        if (val!=1) { status = ERROR_INPUT_PAR; break; }
        if (buffer[0]=='[' || buffer[0]=='%' || buffer[0]=='|') {
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
            continue;
        }

        // match keyword and scan for value
        if (strcmp(buffer,"workdir")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",sbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            memcpy(inparam->workdir,sbuff,STRLEN);
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"problem_num")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->problem_num=ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"print_level")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->print_level = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"output_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->output_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"solver_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->solver_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"stop_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->stop_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"decoup_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->decoup_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"precond_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->precond_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"itsolver_tol")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->itsolver_tol = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"itsolver_maxit")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->itsolver_maxit = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_ILU_levels")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_ILU_levels = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_SWZ_levels")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_SWZ_levels = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"itsolver_restart")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->restart = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"C")==0)||(strcmp(buffer,"c")==0))
                inparam->AMG_type = CLASSIC_AMG;
            else if ((strcmp(buffer,"SA")==0)||(strcmp(buffer,"sa")==0))
                inparam->AMG_type = SA_AMG;
            else if ((strcmp(buffer,"UA")==0)||(strcmp(buffer,"ua")==0))
                inparam->AMG_type = UA_AMG;
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_strong_coupled")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_strong_coupled = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_max_aggregation")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_max_aggregation = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_tentative_smooth")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_tentative_smooth = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_smooth_filter")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"ON")==0)||(strcmp(buffer,"on")==0)||
                (strcmp(buffer,"On")==0)||(strcmp(buffer,"oN")==0)) {
                inparam->AMG_smooth_filter = ON;
            }
            else if ((strcmp(buffer,"OFF")==0)||(strcmp(buffer,"off")==0)||
                     (strcmp(buffer,"ofF")==0)||(strcmp(buffer,"oFf")==0)||
                     (strcmp(buffer,"Off")==0)||(strcmp(buffer,"oFF")==0)||
                     (strcmp(buffer,"OfF")==0)||(strcmp(buffer,"OFf")==0)) {
                inparam->AMG_smooth_filter = OFF;
            }
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_smooth_restriction")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"ON")==0)||(strcmp(buffer,"on")==0)||
                (strcmp(buffer,"On")==0)||(strcmp(buffer,"oN")==0)) {
                inparam->AMG_smooth_restriction = ON;
            }
            else if ((strcmp(buffer,"OFF")==0)||(strcmp(buffer,"off")==0)||
                     (strcmp(buffer,"ofF")==0)||(strcmp(buffer,"oFf")==0)||
                     (strcmp(buffer,"Off")==0)||(strcmp(buffer,"oFF")==0)||
                     (strcmp(buffer,"OfF")==0)||(strcmp(buffer,"OFf")==0)) {
                inparam->AMG_smooth_restriction = OFF;
            }
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_coarse_solver")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_coarse_solver = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_coarse_scaling")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"ON")==0)||(strcmp(buffer,"on")==0)||
                (strcmp(buffer,"On")==0)||(strcmp(buffer,"oN")==0)) {
                inparam->AMG_coarse_scaling = ON;
            }
            else if ((strcmp(buffer,"OFF")==0)||(strcmp(buffer,"off")==0)||
                     (strcmp(buffer,"ofF")==0)||(strcmp(buffer,"oFf")==0)||
                     (strcmp(buffer,"Off")==0)||(strcmp(buffer,"oFF")==0)||
                     (strcmp(buffer,"OfF")==0)||(strcmp(buffer,"OFf")==0)) {
                inparam->AMG_coarse_scaling = OFF;
            }
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_levels")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_levels = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_tol")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_tol = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_maxit")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_maxit = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_coarse_dof")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_coarse_dof = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_cycle_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"V")==0)||(strcmp(buffer,"v")==0))
                inparam->AMG_cycle_type = V_CYCLE;
            else if ((strcmp(buffer,"W")==0)||(strcmp(buffer,"w")==0))
                inparam->AMG_cycle_type = W_CYCLE;
            else if ((strcmp(buffer,"A")==0)||(strcmp(buffer,"a")==0))
                inparam->AMG_cycle_type = AMLI_CYCLE;
            else if ((strcmp(buffer,"NA")==0)||(strcmp(buffer,"na")==0))
                inparam->AMG_cycle_type = NL_AMLI_CYCLE;
            else if ((strcmp(buffer,"VW")==0)||(strcmp(buffer,"vw")==0))
                inparam->AMG_cycle_type = VW_CYCLE;
            else if ((strcmp(buffer,"WV")==0)||(strcmp(buffer,"wv")==0))
                inparam->AMG_cycle_type = WV_CYCLE;
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_smoother")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"JACOBI")==0)||(strcmp(buffer,"jacobi")==0))
```

```
                inparam->AMG_smoother = SMOOTHER_JACOBI;
            else if ((strcmp(buffer,"GS")==0)||(strcmp(buffer,"gs")==0))
                inparam->AMG_smoother = SMOOTHER_GS;
            else if ((strcmp(buffer,"SGS")==0)||(strcmp(buffer,"sgs")==0))
                inparam->AMG_smoother = SMOOTHER_SGS;
            else if ((strcmp(buffer,"CG")==0)||(strcmp(buffer,"cg")==0))
                inparam->AMG_smoother = SMOOTHER_CG;
            else if ((strcmp(buffer,"SOR")==0)||(strcmp(buffer,"sor")==0))
                inparam->AMG_smoother = SMOOTHER_SOR;
            else if ((strcmp(buffer,"SSOR")==0)||(strcmp(buffer,"ssor")==0))
                inparam->AMG_smoother = SMOOTHER_SSOR;
            else if ((strcmp(buffer,"GSOR")==0)||(strcmp(buffer,"gsor")==0))
                inparam->AMG_smoother = SMOOTHER_GSOR;
            else if ((strcmp(buffer,"SGSOR")==0)||(strcmp(buffer,"sgsor")==0))
                inparam->AMG_smoother = SMOOTHER_SGSOR;
            else if ((strcmp(buffer,"POLY")==0)||(strcmp(buffer,"poly")==0))
                inparam->AMG_smoother = SMOOTHER_POLY;
            else if ((strcmp(buffer,"L1DIAG")==0)||(strcmp(buffer,"l1diag")==0))
                inparam->AMG_smoother = SMOOTHER_L1DIAG;
            else if ((strcmp(buffer,"BLKOIL")==0)||(strcmp(buffer,"blkoil")==0))
                inparam->AMG_smoother = SMOOTHER_BLKOIL;
            else if ((strcmp(buffer,"SPETEN")==0)||(strcmp(buffer,"speten")==0))
                inparam->AMG_smoother = SMOOTHER_SPETEN;
            else
                { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_smooth_order")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%s",buffer);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }

            if ((strcmp(buffer,"NO")==0)||(strcmp(buffer,"no")==0))
                inparam->AMG_smooth_order = NO_ORDER;
            else if ((strcmp(buffer,"CF")==0)||(strcmp(buffer,"cf")==0))
                inparam->AMG_smooth_order = CF_ORDER;
            else
            { status = ERROR_INPUT_PAR; break; }
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_coarsening_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_coarsening_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_interpolation_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_interpolation_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_aggregation_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_aggregation_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_pair_number")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_pair_number = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_quality_bound")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_quality_bound = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_aggressive_level")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_aggressive_level = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_aggressive_path")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_aggressive_path = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_presmooth_iter")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_presmooth_iter = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_postsmooth_iter")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_postsmooth_iter = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_relaxation")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_relaxation=dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_polynomial_degree")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_polynomial_degree = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_strong_threshold")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_strong_threshold = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_truncation_threshold")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_truncation_threshold = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_max_row_sum")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_max_row_sum = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_amli_degree")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_amli_degree = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"AMG_nl_amli_krylov_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->AMG_nl_amli_krylov_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"ILU_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->ILU_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"ILU_lfil")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->ILU_lfil = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"ILU_droptol")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->ILU_droptol = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"ILU_relax")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->ILU_relax = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"ILU_permtol")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%lf",&dbuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->ILU_permtol = dbuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"SWZ_mmsize")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->SWZ_mmsize = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"SWZ_maxlvl")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) {status = ERROR_INPUT_PAR; break; }
            inparam->SWZ_maxlvl = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"SWZ_type")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->SWZ_type = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else if (strcmp(buffer,"SWZ_blksolver")==0) {
            val = fscanf(fp,"%s",buffer);
            if (val!=1 || strcmp(buffer,"=")!=0) {
                status = ERROR_INPUT_PAR; break;
            }
            val = fscanf(fp,"%d",&ibuff);
            if (val!=1) { status = ERROR_INPUT_PAR; break; }
            inparam->SWZ_blksolver = ibuff;
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }

        else {
            printf("### WARNING: Unknown input keyword %s!\n", buffer);
            if (fscanf(fp, "%*[^\n]")) {/* skip rest of line and do nothing */ };
        }
```

```
    }

    fclose(fp);

    // if meet unexpected input, stop the program
    fasp_chkerr(status, __FUNCTION__);

    // sanity checks
    status = fasp_param_check(inparam);

#if DEBUG_MODE > 1
    printf("### DEBUG: Reading input status = %d\n", status);
#endif

    fasp_chkerr(status, __FUNCTION__);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 wether 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.

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

size	Number of memory blocks
type	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

mem	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 wether a ILU_data has enough work space.

Parameters

iludata	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

oldmem	Pointer to the existing mem block
tsize	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_functs.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;
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);
```

```
#endif

    if ( tsize > 0 ) {

#if DLMALLOC
        mem = dlcalloc(size,type);
#elif NEDMALLOC
        mem = nedcalloc(size,type);
#else
        mem = calloc(size,type);
#endif

#if DEBUG_MODE > 1
        total_alloc_mem += tsize;
        total_alloc_count++;
#endif
    }

    if ( mem == NULL ) {
        printf("### WARNING: Trying to allocate %lldB RAM...\n", tsize);
        printf("### WARNING: Cannot allocate %.4fMB RAM!\n", (REAL)tsize/Million);
    }

    return mem;
}

void * fasp_mem_realloc (void           *oldmem,
                         const LONGLONG  tsize)
{
    void * mem = NULL;

#if DEBUG_MODE > 1
    printf("### DEBUG: Trying to allocate %.3lfMB RAM!\n", (REAL)tsize/Million);
#endif

    if ( tsize > 0 ) {

#if DLMALLOC
        mem = dlrealloc(oldmem,tsize);
#elif NEDMALLOC
        mem = nedrealloc(oldmem,tsize);
#else
        mem = realloc(oldmem,tsize);
#endif

    }

    if ( mem == NULL ) {
        printf("### WARNING: Trying to allocate %lldB RAM!\n", tsize);
        printf("### WARNING: Cannot allocate %.3lfMB RAM!\n", (REAL)tsize/Million);
    }

    return mem;
}

void fasp_mem_free (void *mem)
{
    if ( mem ) {
#if DLMALLOC
        dlfree(mem);
#elif NEDMALLOC
        nedfree(mem);
#else
        free(mem);
#endif

#if DEBUG_MODE > 1
        total_alloc_count--;
#endif
    }
    else {
#if DEBUG_MODE > 1
        printf("### WARNING: Trying to free an empty pointer!\n");
#endif
    }
}

void fasp_mem_usage ( void )
{
#if DEBUG_MODE > 1
    printf("### DEBUG: Number of alloc = %ld, allocated memory = %.3fMB.\n",
            total_alloc_count, (REAL)total_alloc_mem/Million);
#endif
}

SHORT fasp_mem_iludata_check (const ILU_data *iludata)
{
    const INT memneed = 2*iludata->row; // estimated memory usage

    if ( iludata->nwork >= memneed ) {
        return FASP_SUCCESS;
    }
    else {
        printf("### ERROR: ILU needs %d RAM, only %d available!\n",
               memneed, iludata->nwork);
        return ERROR_ALLOC_MEM;
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 prtlvl)

  Print level and complexity information of AMG.
- void fasp_amgcomplexity_bsr (const AMG_data_bsr *mgl, const SHORT prtlvl)

  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.

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

mgl	Multilevel hierachy for AMG
prtlvl	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

mgl	Multilevel hierachy for AMG
prtlvl	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

status	Error status
fctname	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

message	Some string to print out
cputime	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

ptrlvl	Level for output
stop_type	Type of stopping criteria
iter	Number of iterations

Parameters

relres	Relative residual of different kinds
absres	Absolute residual of different kinds
factor	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

ptrlvl	Level for output
message	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,
                  const REAL  absres,
                  const REAL  factor)
{
    if ( ptrlvl >= PRINT_SOME ) {

        if ( iter > 0 ) {
            printf("%6d | %13.6e   | %13.6e  | %10.4f\n", iter, relres, absres, factor);
        }
        else { // iter = 0:  initial guess
            printf("-----------------------------------------------------------\n");
            switch (stop_type) {
                case STOP_REL_RES:
                    printf("It Num |   ||r||/||b||   |     ||r||      |  Conv.  Factor\n");
                    break;
                case STOP_REL_PRECRES:
                    printf("It Num | ||r||_B/||b||_B |    ||r||_B     |  Conv.  Factor\n");
                    break;
                case STOP_MOD_REL_RES:
                    printf("It Num |   ||r||/||x||   |     ||r||      |  Conv.  Factor\n");
                    break;
            }
            printf("-----------------------------------------------------------\n");
            printf("%6d | %13.6e   | %13.6e  |     -.-- \n", iter, relres, absres);
        } // end if iter

    } // end if ptrlvl
}

void fasp_amgcomplexity (const AMG_data  *mgl,
                         const SHORT      prtlvl)
{
    const SHORT   max_levels = mgl->num_levels;
    SHORT         level;
    REAL          gridcom = 0.0, opcom = 0.0;

    if ( prtlvl >= PRINT_SOME ) {

        printf("-----------------------------------------------------------\n");
        printf("  Level   Num of rows   Num of nonzeros   Avg.  NNZ / row   \n");
        printf("-----------------------------------------------------------\n");

        for ( level = 0; level < max_levels; ++level) {
            const REAL AvgNNZ = (REAL) mgl[level].A.nnz/mgl[level].A.row;
            printf("%5d %13d %17d %14.2f\n",
                   level, mgl[level].A.row, mgl[level].A.nnz, AvgNNZ);
            gridcom += mgl[level].A.row;
            opcom   += mgl[level].A.nnz;

#if 0 // Save coarser linear systems for debugging purposes --Chensong
            char matA[max_levels], rhsb[max_levels];
            if (level > 0) {
                sprintf(matA, "A%d.coo", level);
                sprintf(rhsb, "b%d.coo", level);
                fasp_dcsrvec_write2(matA, rhsb, &(mgl[level].A), &(mgl[level].b));
            }
#endif
        }
        printf("-----------------------------------------------------------\n");

        gridcom /= mgl[0].A.row;
        opcom   /= mgl[0].A.nnz;
        printf("  Grid complexity = %.3f  |", gridcom);
        printf("  Operator complexity = %.3f\n", opcom);

        printf("-----------------------------------------------------------\n");
    }
}

void fasp_amgcomplexity_bsr (const AMG_data_bsr  *mgl,
                             const SHORT          prtlvl)
{
    const SHORT  max_levels = mgl->num_levels;
    SHORT        level;
    REAL         gridcom = 0.0, opcom = 0.0;

    if ( prtlvl >= PRINT_SOME ) {

        printf("-----------------------------------------------------------\n");
        printf("  Level   Num of rows   Num of nonzeros   Avg.  NNZ / row   \n");
        printf("-----------------------------------------------------------\n");

        for ( level = 0; level < max_levels; ++level ) {
            const REAL AvgNNZ = (REAL) mgl[level].A.NNZ/mgl[level].A.ROW;
            printf("%5d  %13d  %17d  %14.2f\n",
                   level,mgl[level].A.ROW, mgl[level].A.NNZ, AvgNNZ);
            gridcom += mgl[level].A.ROW;
            opcom   += mgl[level].A.NNZ;
        }
        printf("-----------------------------------------------------------\n");

        gridcom /= mgl[0].A.ROW;
        opcom   /= mgl[0].A.NNZ;
        printf("  Grid complexity = %.3f  |", gridcom);
        printf("  Operator complexity = %.3f\n", opcom);

        printf("-----------------------------------------------------------\n");

    }
}

void fasp_cputime (const char  *message,
                   const REAL   cputime)
{
    printf("%s costs %.4f seconds\n", message, cputime);
}

void fasp_message (const INT    ptrlvl,
                   const char  *message)
{
    if ( ptrlvl > PRINT_NONE ) printf("%s", message);
}

void fasp_chkerr (const SHORT  status,
                  const char  *fctname)
{
    if ( status >= 0 ) return; // No error found!!!

    switch ( status ) {
        case ERROR_READ_FILE:
            printf("### ERROR: Cannot read file!  [%s]\n", fctname);
            break;
        case ERROR_OPEN_FILE:
            printf("### ERROR: Cannot open file!  [%s]\n", fctname);
            break;
        case ERROR_WRONG_FILE:
            printf("### ERROR: Unknown file format!  [%s]\n", fctname);
            break;
        case ERROR_INPUT_PAR:
            printf("### ERROR: Unknown input argument!  [%s]\n", fctname);
            break;
        case ERROR_REGRESS:
            printf("### ERROR: Regression test failed!  [%s]\n", fctname);
            break;
        case ERROR_ALLOC_MEM:
            printf("### ERROR: Cannot allocate memory!  [%s]\n", fctname);
            break;
        case ERROR_NUM_BLOCKS:
            printf("### ERROR: Unexpected number of blocks!  [%s]\n", fctname);
            break;
        case ERROR_DATA_STRUCTURE:
            printf("### ERROR: Wrong data structure!  [%s]\n", fctname);
            break;
        case ERROR_DATA_ZERODIAG:
            printf("### ERROR: Matrix has zero diagonal entries!  [%s]\n", fctname);
            break;
        case ERROR_DUMMY_VAR:
            printf("### ERROR: Unknown input argument!  [%s]\n", fctname);
            break;
        case ERROR_AMG_INTERP_TYPE:
            printf("### ERROR: Unknown AMG interpolation type!  [%s]\n", fctname);
            break;
        case ERROR_AMG_COARSE_TYPE:
            printf("### ERROR: Unknown AMG coarsening type!  [%s]\n", fctname);
            break;
        case ERROR_AMG_SMOOTH_TYPE:
            printf("### ERROR: Unknown AMG smoother type!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_TYPE:
            printf("### ERROR: Unknown solver type!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_PRECTYPE:
            printf("### ERROR: Unknown preconditioner type!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_STAG:
            printf("### ERROR: Solver stagnation!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_SOLSTAG:
            printf("### ERROR: Solution close to zero!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_TOLSMALL:
            printf("### ERROR: Convergence tolerance too small!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_ILUSETUP:
            printf("### ERROR: ILU setup failed!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_MAXIT:
            printf("### ERROR: Max iteration number reached!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_EXIT:
            printf("### ERROR: Iterative solver failed!  [%s]\n", fctname);
            break;
        case ERROR_SOLVER_MISC:
            printf("### ERROR: Unknown solver runtime error!  [%s]\n", fctname);
            break;
        case ERROR_MISC:
            printf("### ERROR: Miscellaneous error!  [%s]\n", fctname);
            break;
        case ERROR_QUAD_TYPE:
            printf("### ERROR: Unknown quadrature rules!  [%s]\n", fctname);
            break;
        case ERROR_QUAD_DIM:
            printf("### ERROR: Num of quad points not supported!  [%s]\n", fctname);
            break;
        case ERROR_UNKNOWN:
            printf("### ERROR: Unknown error!  [%s]\n", fctname);
            break;
        default:
            break;
    }

    exit(status);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.35 AuxParam.c File Reference

Initialize, set, or print input data and parameters.

```
#include <stdio.h>
#include "fasp.h"
#include "fasp_functs.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 precond_data with AMG_param.
- void fasp_param_prec_to_amg (AMG_param ∗amgparam, const precond_data ∗pcdata)

    Set AMG_param with precond_data.
- void fasp_param_amg_to_precbsr (precond_data_bsr ∗pcdata, const AMG_param ∗amgparam)

    Set precond_data_bsr with AMG_param.
- void fasp_param_precbsr_to_amg (AMG_param ∗amgparam, const precond_data_bsr ∗pcdata)

    Set AMG_param with precond_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

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

amgparam	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

param	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

param	Parameters for AMG
iniparam	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 (
            precond_data * pcdata,
            const AMG_param * amgparam )
```

Set precond_data with AMG_param.

Parameters

pcdata	Preconditioning data structure
amgparam	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 (
            precond_data_bsr * pcdata,
            const AMG_param * amgparam )
```

Set precond_data_bsr with AMG_param.

Parameters

pcdata	Preconditioning data structure
amgparam	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

iluparam	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

param	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

iluparam	Parameters for ILU
iniparam	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

iniparam	Input parameters
itsparam	Iterative solver parameters
amgparam	AMG parameters
iluparam	ILU parameters
swzparam	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

iniparam	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 precond_data * pcdata )
```

Set AMG_param with precond_data.

Parameters

amgparam	Parameters for AMG
pcdata	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 precond_data_bsr * pcdata )
```

Set AMG_param with precond_data.

Parameters

amgparam	Parameters for AMG
pcdata	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

argc	Number of arg input
argv	Input arguments
iniparam	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

itsparam	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

param	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

itsparam	Parameters for iterative solvers
iniparam	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

swzparam	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

param	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

swzparam	Parameters for Schwarz method
iniparam	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.

```

#include <stdio.h>

#include "fasp.h"
#include "fasp_functs.h"

#if DEBUG_MODE > 1
unsigned long total_alloc_mem;
unsigned long total_alloc_count;
#endif

/*---------------------------------*/
```

```
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_param_set (const int     argc,
                     const char   *argv[],
                     input_param  *iniparam)
{
    int      arg_index   = 1;
    int      print_usage = FALSE;
    SHORT    status      = FASP_SUCCESS;

    // Option 1.  set default input parameters
    fasp_param_input_init(iniparam);

    while ( arg_index < argc ) {

        if ( strcmp(argv[arg_index], "-help") == 0 ) {
            print_usage = TRUE; break;
        }

        // Option 2.  Get parameters from an ini file
        else if ( strcmp(argv[arg_index], "-ini") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Missing ini filename!  [%s]\n", __FUNCTION__);
                print_usage = TRUE; break;
            }
            strcpy(iniparam->inifile, argv[arg_index]);
            fasp_param_input(iniparam->inifile,iniparam);
            if ( ++arg_index >= argc ) break;
        }

        // Option 3.  Get parameters from command line input
        else if ( strcmp(argv[arg_index], "-print") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting print level (from 0 to 10).\n");
                print_usage = TRUE; break;
            }
            iniparam->print_level = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-output") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting output type (0 or 1).\n");
                print_usage = TRUE; break;
            }
            iniparam->output_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-solver") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting solver type.\n");
                print_usage = TRUE; break;
            }
            iniparam->solver_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-precond") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting preconditioner type.\n");
                print_usage = TRUE; break;
            }
            iniparam->precond_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-maxit") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting max number of iterations.\n");
                print_usage = TRUE; break;
            }
            iniparam->itsolver_maxit = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-tol") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting tolerance for itsolver.\n");
                print_usage = TRUE; break;
            }
            iniparam->itsolver_tol = atof(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgmaxit") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting max num of iterations for AMG.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_maxit = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgtol") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting tolerance for AMG.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_tol = atof(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgtype") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG type (1, 2, 3).\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgcycle") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG cycle type (1, 2, 3, 12, 21).\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_cycle_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgcoarsening") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG coarsening type.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_coarsening_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amginterplation") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG interpolation type.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_interpolation_type = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgsmoother") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG smoother type.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_smoother = atoi(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgsthreshold") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG strong threshold.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_strong_threshold = atof(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else if ( strcmp(argv[arg_index], "-amgscouple") == 0 ) {
            arg_index++;
            if ( arg_index >= argc ) {
                printf("### ERROR: Expecting AMG strong coupled threshold.\n");
                print_usage = TRUE; break;
            }
            iniparam->AMG_strong_coupled = atof(argv[arg_index]);
            if ( ++arg_index >= argc ) break;
        }

        else {
            print_usage = TRUE;
            break;
        }

    }

    if ( print_usage ) {

        printf("FASP command line options:\n");

        printf("================================================================\n");
        printf("  -ini              [CharValue] :  Ini file name\n");
        printf("  -print            [IntValue]  :  Print level\n");
        printf("  -output           [IntValue]  :  Output to screen or a log file\n");
        printf("  -solver           [IntValue]  :  Solver type\n");
        printf("  -precond          [IntValue]  :  Preconditioner type\n");
        printf("  -maxit            [IntValue]  :  Max number of iterations\n");
        printf("  -tol              [RealValue] :  Tolerance for iterative solvers\n");
        printf("  -amgmaxit         [IntValue]  :  Max number of AMG iterations\n");
        printf("  -amgtol           [RealValue] :  Tolerance for AMG methods\n");
        printf("  -amgtype          [IntValue]  :  AMG type\n");
        printf("  -amgcycle         [IntValue]  :  AMG cycle type\n");
        printf("  -amgcoarsening    [IntValue]  :  AMG coarsening type\n");
        printf("  -amginterpolation [IntValue]  :  AMG interpolation type\n");
        printf("  -amgsmoother      [IntValue]  :  AMG smoother type\n");
        printf("  -amgsthreshold    [RealValue] :  AMG strong threshold\n");
        printf("  -amgscoupled      [RealValue] :  AMG strong coupled threshold\n");
        printf("  -help                         :  Brief help messages\n");

        exit(ERROR_INPUT_PAR);

    }

    // sanity checks
    status = fasp_param_check(iniparam);

    // if meet unexpected input, stop the program
    fasp_chkerr(status, __FUNCTION__);

}

void fasp_param_init (const input_param  *iniparam,
                      ITS_param          *itsparam,
                      AMG_param          *amgparam,
                      ILU_param          *iluparam,
                      SWZ_param          *swzparam)
{
#if DEBUG_MODE > 1
    total_alloc_mem   = 0; // initialize total memeory amount
    total_alloc_count = 0; // initialize alloc count
#endif

    if (itsparam) fasp_param_solver_init(itsparam);
    if (amgparam) fasp_param_amg_init(amgparam);
    if (iluparam) fasp_param_ilu_init(iluparam);
    if (swzparam) fasp_param_swz_init(swzparam);

    if (iniparam) {
        if (itsparam) fasp_param_solver_set(itsparam,iniparam);
        if (amgparam) fasp_param_amg_set(amgparam,iniparam);
        if (iluparam) fasp_param_ilu_set(iluparam,iniparam);
        if (swzparam) fasp_param_swz_set(swzparam,iniparam);
    }
    else {
        printf("### WARNING: No input given!  Use default values instead.\n");
    }

    // if using AMG as a solver, set min num of iterations = 50
    if ( (itsparam == NULL) && (amgparam != NULL) ) {
        amgparam->maxit = MAX(amgparam->maxit, 50);
    }
}

void fasp_param_input_init (input_param *iniparam)
{
    strcpy(iniparam->workdir,"../data/");

    // Input/output
    iniparam->print_level               = PRINT_SOME;
    iniparam->output_type               = 0;

    // Problem information
    iniparam->problem_num               = 10;
    iniparam->solver_type               = SOLVER_CG;
    iniparam->decoup_type               = 1;
    iniparam->precond_type              = PREC_AMG;
    iniparam->stop_type                 = STOP_REL_RES;

    // Solver parameters
    iniparam->itsolver_tol              = 1e-6;
    iniparam->itsolver_maxit            = 500;
    iniparam->restart                   = 25;

    // ILU method parameters
    iniparam->ILU_type                  = ILUk;
    iniparam->ILU_lfil                  = 0;
    iniparam->ILU_droptol               = 0.001;
    iniparam->ILU_relax                 = 0;
    iniparam->ILU_permtol               = 0.0;

    // Schwarz method parameters
    iniparam->SWZ_mmsize                = 200;
    iniparam->SWZ_maxlvl                = 2;
    iniparam->SWZ_type                  = 1;
    iniparam->SWZ_blksolver             = SOLVER_DEFAULT;

    // AMG method parameters
    iniparam->AMG_type                  = CLASSIC_AMG;
    iniparam->AMG_levels                = 20;
    iniparam->AMG_cycle_type            = V_CYCLE;
    iniparam->AMG_smoother              = SMOOTHER_GS;
    iniparam->AMG_smooth_order          = CF_ORDER;
    iniparam->AMG_presmooth_iter        = 1;
    iniparam->AMG_postsmooth_iter       = 1;
    iniparam->AMG_relaxation            = 1.0;
    iniparam->AMG_coarse_dof            = 500;
    iniparam->AMG_coarse_solver         = 0;
    iniparam->AMG_tol                   = 1e-6;
    iniparam->AMG_maxit                 = 1;
    iniparam->AMG_ILU_levels            = 0;
    iniparam->AMG_SWZ_levels            = 0;
    iniparam->AMG_coarse_scaling        = OFF; // Require investigation --Chensong
    iniparam->AMG_amli_degree           = 1;
    iniparam->AMG_nl_amli_krylov_type   = 2;

    // Classical AMG specific
    iniparam->AMG_coarsening_type       = 1;
    iniparam->AMG_interpolation_type    = 1;
    iniparam->AMG_max_row_sum           = 0.9;
    iniparam->AMG_strong_threshold      = 0.3;
    iniparam->AMG_truncation_threshold  = 0.2;
    iniparam->AMG_aggressive_level      = 0;
    iniparam->AMG_aggressive_path       = 1;

    // Aggregation AMG specific
    iniparam->AMG_aggregation_type      = PAIRWISE;
    iniparam->AMG_quality_bound         = 8.0;
    iniparam->AMG_pair_number           = 2;
    iniparam->AMG_strong_coupled        = 0.25;
    iniparam->AMG_max_aggregation       = 9;
    iniparam->AMG_tentative_smooth     = 0.67;
    iniparam->AMG_smooth_filter        = ON;
    iniparam->AMG_smooth_restriction   = ON;
}

void fasp_param_amg_init (AMG_param *amgparam)
{
    // General AMG parameters
    amgparam->AMG_type             = CLASSIC_AMG;
    amgparam->print_level          = PRINT_NONE;
    amgparam->maxit                = 1;
    amgparam->tol                  = 1e-6;
    amgparam->max_levels           = 20;
    amgparam->coarse_dof           = 500;
    amgparam->cycle_type           = V_CYCLE;
    amgparam->smoother             = SMOOTHER_GS;
    amgparam->smooth_order         = CF_ORDER;
    amgparam->presmooth_iter       = 1;
    amgparam->postsmooth_iter      = 1;
    amgparam->coarse_solver        = SOLVER_DEFAULT;
    amgparam->relaxation           = 1.0;
    amgparam->polynomial_degree    = 3;
    amgparam->coarse_scaling       = OFF;
    amgparam->amli_degree          = 2;
    amgparam->amli_coef            = NULL;
    amgparam->nl_amli_krylov_type  = SOLVER_GCG;

    // Classical AMG specific
    amgparam->coarsening_type      = COARSE_RS;
    amgparam->interpolation_type   = INTERP_DIR;
    amgparam->max_row_sum          = 0.9;
    amgparam->strong_threshold     = 0.3;
    amgparam->truncation_threshold = 0.2;
    amgparam->aggressive_level     = 0;
    amgparam->aggressive_path      = 1;

    // Aggregation AMG specific
    amgparam->aggregation_type     = PAIRWISE;
    amgparam->quality_bound        = 10.0;
    amgparam->pair_number          = 2;
    amgparam->strong_coupled       = 0.08;
    amgparam->max_aggregation      = 20;
    amgparam->tentative_smooth     = 0.67;
    amgparam->smooth_filter        = ON;
    amgparam->smooth_restriction   = ON;

    // ILU smoother parameters
    amgparam->ILU_type             = ILUk;
    amgparam->ILU_levels           = 0;
    amgparam->ILU_lfil             = 0;
    amgparam->ILU_droptol          = 0.001;
    amgparam->ILU_relax            = 0;

    // Schwarz smoother parameters
    amgparam->SWZ_levels           = 0; // levels will use Schwarz smoother
    amgparam->SWZ_mmsize           = 200;
    amgparam->SWZ_maxlvl           = 3; // vertices with smaller distance
    amgparam->SWZ_type             = 1;
    amgparam->SWZ_blksolver        = SOLVER_DEFAULT;
}

void fasp_param_solver_init (ITS_param *itsparam)
{
    itsparam->print_level    = PRINT_NONE;
    itsparam->itsolver_type = SOLVER_CG;
    itsparam->decoup_type    = 1;
    itsparam->precond_type   = PREC_AMG;
    itsparam->stop_type      = STOP_REL_RES;
    itsparam->maxit          = 500;
    itsparam->restart        = 25;
    itsparam->tol            = 1e-6;
}

void fasp_param_ilu_init (ILU_param *iluparam)
{
    iluparam->print_level  = PRINT_NONE;
    iluparam->ILU_type     = ILUk;
    iluparam->ILU_lfil     = 2;
    iluparam->ILU_droptol  = 0.001;
    iluparam->ILU_relax    = 0;
    iluparam->ILU_permtol  = 0.01;
```

```
}

void fasp_param_swz_init (SWZ_param *swzparam)
{
    swzparam->print_level   = PRINT_NONE;
    swzparam->SWZ_type      = 3;
    swzparam->SWZ_maxlvl    = 2;
    swzparam->SWZ_mmsize    = 200;
    swzparam->SWZ_blksolver = 0;
}

void fasp_param_amg_set (AMG_param          *param,
                         const input_param  *iniparam)
{
    param->AMG_type    = iniparam->AMG_type;
    param->print_level = iniparam->print_level;

    if (iniparam->solver_type == SOLVER_AMG) {
        param->maxit = iniparam->itsolver_maxit;
        param->tol   = iniparam->itsolver_tol;
    }
    else if (iniparam->solver_type == SOLVER_FMG) {
        param->maxit = iniparam->itsolver_maxit;
        param->tol   = iniparam->itsolver_tol;
    }
    else {
        param->maxit = iniparam->AMG_maxit;
        param->tol   = iniparam->AMG_tol;
    }

    param->max_levels           = iniparam->AMG_levels;
    param->cycle_type           = iniparam->AMG_cycle_type;
    param->smoother             = iniparam->AMG_smoother;
    param->smooth_order         = iniparam->AMG_smooth_order;
    param->relaxation           = iniparam->AMG_relaxation;
    param->coarse_solver        = iniparam->AMG_coarse_solver;
    param->polynomial_degree    = iniparam->AMG_polynomial_degree;
    param->presmooth_iter       = iniparam->AMG_presmooth_iter;
    param->postsmooth_iter      = iniparam->AMG_postsmooth_iter;
    param->coarse_dof           = iniparam->AMG_coarse_dof;
    param->coarse_scaling       = iniparam->AMG_coarse_scaling;
    param->amli_degree          = iniparam->AMG_amli_degree;
    param->amli_coef            = NULL;
    param->nl_amli_krylov_type  = iniparam->AMG_nl_amli_krylov_type;

    param->coarsening_type      = iniparam->AMG_coarsening_type;
    param->interpolation_type   = iniparam->AMG_interpolation_type;
    param->strong_threshold     = iniparam->AMG_strong_threshold;
    param->truncation_threshold = iniparam->AMG_truncation_threshold;
    param->max_row_sum          = iniparam->AMG_max_row_sum;
    param->aggressive_level     = iniparam->AMG_aggressive_level;
    param->aggressive_path      = iniparam->AMG_aggressive_path;

    param->aggregation_type     = iniparam->AMG_aggregation_type;
    param->pair_number          = iniparam->AMG_pair_number;
    param->quality_bound        = iniparam->AMG_quality_bound;
    param->strong_coupled       = iniparam->AMG_strong_coupled;
    param->max_aggregation      = iniparam->AMG_max_aggregation;
    param->tentative_smooth     = iniparam->AMG_tentative_smooth;
    param->smooth_filter        = iniparam->AMG_smooth_filter;
    param->smooth_restriction   = iniparam->AMG_smooth_restriction;

    param->ILU_levels           = iniparam->AMG_ILU_levels;
    param->ILU_type             = iniparam->ILU_type;
    param->ILU_lfil             = iniparam->ILU_lfil;
    param->ILU_droptol          = iniparam->ILU_droptol;
    param->ILU_relax            = iniparam->ILU_relax;
    param->ILU_permtol          = iniparam->ILU_permtol;

    param->SWZ_levels           = iniparam->AMG_SWZ_levels;
    param->SWZ_mmsize           = iniparam->SWZ_mmsize;
    param->SWZ_maxlvl           = iniparam->SWZ_maxlvl;
    param->SWZ_type             = iniparam->SWZ_type;
}

void fasp_param_ilu_set (ILU_param          *iluparam,
                         const input_param  *iniparam)
{
    iluparam->print_level = iniparam->print_level;
    iluparam->ILU_type    = iniparam->ILU_type;
    iluparam->ILU_lfil    = iniparam->ILU_lfil;
    iluparam->ILU_droptol = iniparam->ILU_droptol;
    iluparam->ILU_relax   = iniparam->ILU_relax;
    iluparam->ILU_permtol = iniparam->ILU_permtol;
}

void fasp_param_swz_set (SWZ_param          *swzparam,
                         const input_param  *iniparam)
{
    swzparam->print_level   = iniparam->print_level;
    swzparam->SWZ_type      = iniparam->SWZ_type;
    swzparam->SWZ_maxlvl    = iniparam->SWZ_maxlvl;
    swzparam->SWZ_mmsize    = iniparam->SWZ_mmsize;
    swzparam->SWZ_blksolver = iniparam->SWZ_blksolver;
}

void fasp_param_solver_set (ITS_param          *itsparam,
                            const input_param  *iniparam)
{
    itsparam->print_level    = iniparam->print_level;
    itsparam->itsolver_type  = iniparam->solver_type;
    itsparam->decoup_type    = iniparam->decoup_type;
    itsparam->precond_type   = iniparam->precond_type;
    itsparam->stop_type      = iniparam->stop_type;
    itsparam->restart        = iniparam->restart;

    if ( itsparam->itsolver_type == SOLVER_AMG ) {
        itsparam->tol   = iniparam->AMG_tol;
        itsparam->maxit = iniparam->AMG_maxit;
    }
    else {
        itsparam->tol   = iniparam->itsolver_tol;
        itsparam->maxit = iniparam->itsolver_maxit;
    }
}

void fasp_param_amg_to_prec (precond_data     *pcdata,
                             const AMG_param  *amgparam)
{
    pcdata->AMG_type            = amgparam->AMG_type;
    pcdata->print_level         = amgparam->print_level;
    pcdata->maxit               = amgparam->maxit;
    pcdata->max_levels          = amgparam->max_levels;
    pcdata->tol                 = amgparam->tol;
    pcdata->cycle_type          = amgparam->cycle_type;
    pcdata->smoother            = amgparam->smoother;
    pcdata->smooth_order        = amgparam->smooth_order;
    pcdata->presmooth_iter      = amgparam->presmooth_iter;
    pcdata->postsmooth_iter     = amgparam->postsmooth_iter;
    pcdata->coarsening_type     = amgparam->coarsening_type;
    pcdata->coarse_solver       = amgparam->coarse_solver;
    pcdata->relaxation          = amgparam->relaxation;
    pcdata->polynomial_degree   = amgparam->polynomial_degree;
    pcdata->coarse_scaling      = amgparam->coarse_scaling;
    pcdata->amli_degree         = amgparam->amli_degree;
    pcdata->amli_coef           = amgparam->amli_coef;
    pcdata->nl_amli_krylov_type = amgparam->nl_amli_krylov_type;
    pcdata->tentative_smooth    = amgparam->tentative_smooth;
}

void fasp_param_prec_to_amg (AMG_param           *amgparam,
                             const precond_data  *pcdata)
{
    amgparam->AMG_type            = pcdata->AMG_type;
    amgparam->print_level         = pcdata->print_level;
    amgparam->cycle_type          = pcdata->cycle_type;
    amgparam->smoother            = pcdata->smoother;
    amgparam->smooth_order        = pcdata->smooth_order;
    amgparam->presmooth_iter      = pcdata->presmooth_iter;
    amgparam->postsmooth_iter     = pcdata->postsmooth_iter;
    amgparam->relaxation          = pcdata->relaxation;
    amgparam->polynomial_degree   = pcdata->polynomial_degree;
    amgparam->coarse_solver       = pcdata->coarse_solver;
    amgparam->coarse_scaling      = pcdata->coarse_scaling;
    amgparam->amli_degree         = pcdata->amli_degree;
    amgparam->amli_coef           = pcdata->amli_coef;
    amgparam->nl_amli_krylov_type = pcdata->nl_amli_krylov_type;
    amgparam->tentative_smooth    = pcdata->tentative_smooth;
    amgparam->ILU_levels          = pcdata->mgl_data->ILU_levels;
}

void fasp_param_amg_to_precbsr (precond_data_bsr  *pcdata,
```

```c
                                const AMG_param   *amgparam)
{
    pcdata->AMG_type            = amgparam->AMG_type;
    pcdata->print_level         = amgparam->print_level;
    pcdata->maxit               = amgparam->maxit;
    pcdata->max_levels          = amgparam->max_levels;
    pcdata->tol                 = amgparam->tol;
    pcdata->cycle_type          = amgparam->cycle_type;
    pcdata->smoother            = amgparam->smoother;
    pcdata->smooth_order        = amgparam->smooth_order;
    pcdata->presmooth_iter      = amgparam->presmooth_iter;
    pcdata->postsmooth_iter     = amgparam->postsmooth_iter;
    pcdata->coarse_solver       = amgparam->coarse_solver;
    pcdata->coarsening_type     = amgparam->coarsening_type;
    pcdata->relaxation          = amgparam->relaxation;
    pcdata->coarse_scaling      = amgparam->coarse_scaling;
    pcdata->amli_degree         = amgparam->amli_degree;
    pcdata->amli_coef           = amgparam->amli_coef;
    pcdata->nl_amli_krylov_type = amgparam->nl_amli_krylov_type;
    pcdata->tentative_smooth    = amgparam->tentative_smooth;
}

void fasp_param_precbsr_to_amg (AMG_param               *amgparam,
                                const precond_data_bsr  *pcdata)
{
    amgparam->AMG_type            = pcdata->AMG_type;
    amgparam->print_level         = pcdata->print_level;
    amgparam->cycle_type          = pcdata->cycle_type;
    amgparam->smoother            = pcdata->smoother;
    amgparam->smooth_order        = pcdata->smooth_order;
    amgparam->presmooth_iter      = pcdata->presmooth_iter;
    amgparam->postsmooth_iter     = pcdata->postsmooth_iter;
    amgparam->relaxation          = pcdata->relaxation;
    amgparam->coarse_solver       = pcdata->coarse_solver;
    amgparam->coarse_scaling      = pcdata->coarse_scaling;
    amgparam->amli_degree         = pcdata->amli_degree;
    amgparam->amli_coef           = pcdata->amli_coef;
    amgparam->nl_amli_krylov_type = pcdata->nl_amli_krylov_type;
    amgparam->tentative_smooth    = pcdata->tentative_smooth;
    amgparam->ILU_levels          = pcdata->mgl_data->ILU_levels;
}

void fasp_param_amg_print (const AMG_param *param)
{

    if ( param ) {

        printf("\n       Parameters in AMG_param\n");
        printf("-----------------------------------------------\n");

        printf("AMG print level:                                      %d\n", param->print_level);
        printf("AMG max num of iter:                              %d\n", param->maxit);
        printf("AMG type:                                                    %d\n", param->AMG_type);
        printf("AMG tolerance:                                          %.2e\n", param->tol);
        printf("AMG max levels:                                        %d\n", param->max_levels);
        printf("AMG cycle type:                                        %d\n", param->cycle_type);
        printf("AMG coarse solver type:                        %d\n", param->coarse_solver);
        printf("AMG scaling of coarse correction:    %d\n", param->coarse_scaling);
        printf("AMG smoother type:                                  %d\n", param->smoother);
        printf("AMG smoother order:                               %d\n", param->smooth_order);
        printf("AMG num of presmoothing:                      %d\n", param->presmooth_iter);
        printf("AMG num of postsmoothing:                    %d\n", param->postsmooth_iter);

        if ( param->smoother == SMOOTHER_SOR  ||
            param->smoother == SMOOTHER_SSOR ||
            param->smoother == SMOOTHER_GSOR ||
            param->smoother == SMOOTHER_SGSOR ) {
            printf("AMG relax factor:                                    %.4f\n",
                   param->relaxation);
        }

        if ( param->smoother == SMOOTHER_POLY ) {
            printf("AMG polynomial smoother degree:        %d\n",
                   param->polynomial_degree);
        }

        if ( param->cycle_type == AMLI_CYCLE ) {
            printf("AMG AMLI degree of polynomial:          %d\n",
                   param->amli_degree);
        }

        if ( param->cycle_type == NL_AMLI_CYCLE ) {
            printf("AMG Nonlinear AMLI Krylov type:        %d\n",
                   param->nl_amli_krylov_type);
        }

        switch (param->AMG_type) {
                case CLASSIC_AMG:
                printf("AMG coarsening type:                           %d\n",
                       param->coarsening_type);
                printf("AMG interpolation type:                        %d\n",
                       param->interpolation_type);
                printf("AMG dof on coarsest grid:                      %d\n",
                       param->coarse_dof);
                printf("AMG strong threshold:                          %.4f\n",
                       param->strong_threshold);
                printf("AMG truncation threshold:                      %.4f\n",
                       param->truncation_threshold);
                printf("AMG max row sum:                               %.4f\n",
                       param->max_row_sum);
                printf("AMG aggressive levels:                         %d\n",
                       param->aggressive_level);
                printf("AMG aggressive path:                           %d\n",
                       param->aggressive_path);
                break;

                default:  // SA_AMG or UA_AMG
                printf("Aggregation type:                              %d\n",
                       param->aggregation_type);
                if ( param->aggregation_type == PAIRWISE ) {
                    printf("Aggregation number of pairs:               %d\n",
                           param->pair_number);
                    printf("Aggregation quality bound:                 %.2f\n",
                           param->quality_bound);
                }
                if ( param->aggregation_type == VMB ) {
                    printf("Aggregation strong coupling:               %.4f\n",
                           param->strong_coupled);
                    printf("Aggregation max aggregation:               %d\n",
                           param->max_aggregation);
                    printf("Aggregation tentative smooth:              %.4f\n",
                           param->tentative_smooth);
                    printf("Aggregation smooth filter:                 %d\n",
                           param->smooth_filter);
                    printf("Aggregation smooth restriction:        %d\n",
                           param->smooth_restriction);

                }
                break;
        }

        if (param->ILU_levels>0) {
            printf("AMG ILU smoother level:                        %d\n", param->ILU_levels);
            printf("AMG ILU type:                                  %d\n", param->ILU_type);
            printf("AMG ILU level of fill-in:                      %d\n", param->ILU_lfil);
            printf("AMG ILU drop tol:                              %e\n", param->ILU_droptol);
            printf("AMG ILU relaxation:                            %f\n", param->ILU_relax);
        }

        if (param->SWZ_levels>0){
            printf("AMG Schwarz smoother level:                %d\n", param->SWZ_levels);
            printf("AMG Schwarz type:                                  %d\n", param->SWZ_type);
            printf("AMG Schwarz forming block level:      %d\n", param->SWZ_maxlvl);
            printf("AMG Schwarz maximal block size:        %d\n", param->SWZ_mmsize);
        }

        printf("-----------------------------------------------\n\n");

    }
    else {
        printf("### WARNING: AMG_param has not been set!\n");
    } // end if (param)

}

void fasp_param_ilu_print (const ILU_param *param)
{
    if ( param ) {

        printf("\n       Parameters in ILU_param\n");
        printf("-----------------------------------------------\n");
        printf("ILU print level:                               %d\n",   param->print_level);
        printf("ILU type:                                                    %d\n",   param->ILU_type);
        printf("ILU level of fill-in:                            %d\n",   param->ILU_lfil);
        printf("ILU relaxation factor:                          %.4f\n", param->ILU_relax);
        printf("ILU drop tolerance:                                %.2e\n", param->ILU_droptol);
        printf("ILU permutation tolerance:                  %.2e\n", param->ILU_permtol);
        printf("-----------------------------------------------\n\n");

    }
    else {
        printf("### WARNING: ILU_param has not been set!\n");
    }
}

void fasp_param_swz_print (const SWZ_param *param)
{
    if ( param ) {

        printf("\n       Parameters in SWZ_param\n");
        printf("-----------------------------------------------\n");
        printf("Schwarz print level:                              %d\n", param->print_level);
        printf("Schwarz type:                                            %d\n", param->SWZ_type);
        printf("Schwarz forming block level:              %d\n", param->SWZ_maxlvl);
        printf("Schwarz maximal block size:                %d\n", param->SWZ_mmsize);
        printf("Schwarz block solver type:                  %d\n", param->SWZ_blksolver);
        printf("-----------------------------------------------\n\n");

    }
    else {
        printf("### WARNING: SWZ_param has not been set!\n");
    }
}

void fasp_param_solver_print (const ITS_param *param)
{
    if ( param ) {

        printf("\n       Parameters in ITS_param\n");
        printf("-----------------------------------------------\n");

        printf("Solver print level:                                %d\n", param->print_level);
        printf("Solver type:                                              %d\n", param->itsolver_type);
        printf("Solver precond type:                              %d\n", param->precond_type);
        printf("Solver max num of iter:                        %d\n", param->maxit);
        printf("Solver tolerance:                                    %.2e\n", param->tol);
        printf("Solver stopping type:                            %d\n", param->stop_type);

        if (param->itsolver_type==SOLVER_GMRES ||
            param->itsolver_type==SOLVER_VGMRES) {
            printf("Solver restart number:                          %d\n", param->restart);
        }

        printf("-----------------------------------------------\n\n");

    }
    else {
        printf("### WARNING: ITS_param has not been set!\n");
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.37 AuxSort.c File Reference

Array sorting/merging and removing duplicated integers.

```
#include "fasp.h"
#include "fasp_functs.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

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

nlist	Length of the array list
list	Pointer to a set of values
value	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

a	Pointer to the array needed to be sorted
left	Starting index
right	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

a	Pointer to the array
left	Starting index
right	Ending index
index	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

a	Pointer to the array needed to be sorted
left	Starting index
right	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

a	Pointer to the array
left	Starting index
right	Ending index
index	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

numbers	Pointer to the array needed to be sorted
work	Pointer to the work array with same size as numbers
left	Starting index of array 1
mid	Starting index of array 2
right	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

numbers	Pointer to the array needed to be sorted
work	Pointer to the work array with same size as numbers
left	Starting index
right	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

numbers	Pointer to the array needed to be sorted (in/out)
size	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_functs.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
```

```
    while ((left <= left_end) && (mid <= right)) {

        if (numbers[left] <= numbers[mid]) // first branch <=
        {
            work[tmp_pos] = numbers[left];
            tmp_pos = tmp_pos + 1;
            left = left +1;
        }
        else // second branch >
        {
            work[tmp_pos] = numbers[mid];
            tmp_pos = tmp_pos + 1;
            mid = mid + 1;
        }
    }

    while (left <= left_end) {
        work[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1;
    }

    while (mid <= right) {
        work[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;
    }

    for (i = 0; i < num_elements; ++i) {
        numbers[right] = work[right];
        right = right - 1;
    }

}

void fasp_aux_msort (INT  numbers[],
                     INT  work[],
                     INT  left,
                     INT  right)
{
    INT mid;

    if (right > left) {
        mid = (right + left) / 2;
        fasp_aux_msort(numbers, work, left, mid);
        fasp_aux_msort(numbers, work, mid+1, right);
        fasp_aux_merge(numbers, work, left, mid+1, right);
    }

}

void fasp_aux_iQuickSort (INT  *a,
                          INT   left,
                          INT   right)
{
    INT i, last;

    if (left >= right) return;

    iSwapping(a, left, (left+right)/2);

    last = left;
    for (i = left+1; i <= right; ++i) {
        if (a[i] < a[left]) {
            iSwapping(a, ++last, i);
        }
    }

    iSwapping(a, left, last);

    fasp_aux_iQuickSort(a, left, last-1);
    fasp_aux_iQuickSort(a, last+1, right);
}

void fasp_aux_dQuickSort (REAL  *a,
                          INT   left,
                          INT   right)
{
    INT i, last;

    if (left >= right) return;

    dSwapping(a, left, (left+right)/2);

    last = left;
    for (i = left+1; i <= right; ++i) {
        if (a[i] < a[left]) {
            dSwapping(a, ++last, i);
        }
    }

    dSwapping(a, left, last);

    fasp_aux_dQuickSort(a, left, last-1);
    fasp_aux_dQuickSort(a, last+1, right);
}

void fasp_aux_iQuickSortIndex (INT  *a,
                               INT   left,
                               INT   right,
                               INT  *index)
{
    INT i, last;

    if (left >= right) return;

    iSwapping(index, left, (left+right)/2);

    last = left;
    for (i = left+1; i <= right; ++i) {
        if (a[index[i]] < a[index[left]]) {
            iSwapping(index, ++last, i);
        }
    }

    iSwapping(index, left, last);

    fasp_aux_iQuickSortIndex(a, left, last-1, index);
    fasp_aux_iQuickSortIndex(a, last+1, right, index);
}

void fasp_aux_dQuickSortIndex (REAL  *a,
                               INT    left,
                               INT    right,
                               INT   *index)
{
    INT i, last;

    if (left >= right) return;

    iSwapping(index, left, (left+right)/2);

    last = left;
    for (i = left+1; i <= right; ++i) {
        if (a[index[i]] < a[index[left]]) {
            iSwapping(index, ++last, i);
        }
    }

    iSwapping(index, left, last);

    fasp_aux_dQuickSortIndex(a, left, last-1, index);
    fasp_aux_dQuickSortIndex(a, last+1, right, index);
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void iSwapping (INT       *w,
                       const INT  i,
                       const INT  j)
{
    const INT temp = w[i];
    w[i] = w[j]; w[j] = temp;
}

static void dSwapping (REAL      *w,
                       const INT  i,
                       const INT  j)
{
    const REAL temp = w[i];
    w[i] = w[j]; w[j] = temp;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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_lGS =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.

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

procid	Index of thread
nprocs	Number of threads
n	Total workload
start	Pointer to the begin of each thread in total workload
end	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

mythreads	Total threads of solver
its	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;
```

```
    }
    else {
        end_loc = chunk_size;
        start_loc = end_loc * procid + mod;
    }
    end_loc = end_loc + start_loc;

    *start = start_loc;
    *end = end_loc;
}

INT THDs_AMG_GS=0;
INT THDs_CPR_lGS=0;
INT THDs_CPR_gGS=0;
void fasp_set_gs_threads (const INT mythreads,
                          const INT its)
{
#ifdef _OPENMP

#if 1

    if (its <=8) {
        THDs_AMG_GS =  mythreads;
        THDs_CPR_lGS = mythreads ;
        THDs_CPR_gGS = mythreads ;
    }
    else if (its <=12) {
        THDs_AMG_GS =  mythreads;
        THDs_CPR_lGS = (6 < mythreads) ?  6 :  mythreads;
        THDs_CPR_gGS = (4 < mythreads) ?  4 :  mythreads;
    }
    else if (its <=15) {
        THDs_AMG_GS =  (3 < mythreads) ?  3 :  mythreads;
        THDs_CPR_lGS = (3 < mythreads) ?  3 :  mythreads;
        THDs_CPR_gGS = (2 < mythreads) ?  2 :  mythreads;
    }
    else if (its <=18) {
        THDs_AMG_GS =  (2 < mythreads) ?  2 :  mythreads;
        THDs_CPR_lGS = (2 < mythreads) ?  2 :  mythreads;
        THDs_CPR_gGS = (1 < mythreads) ?  1 :  mythreads;
    }
    else {
        THDs_AMG_GS =  1;
        THDs_CPR_lGS = 1;
        THDs_CPR_gGS = 1;
    }

#else

    THDs_AMG_GS =  mythreads;
    THDs_CPR_lGS = mythreads ;
    THDs_CPR_gGS = mythreads ;

#endif

#endif // _OPENMP
}

/*----------------------------------*/
/*--        End of File          --*/
/*----------------------------------*/
```

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.

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

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

m	Number of rows
u	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

x	Pointer to dvector
y	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

m	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

u	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

u	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

x	Pointer to dvector
y	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

n	Size of the vector
x	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

n	Number of variables
x	Pointer to dvector
val	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

b	Pointer to dvector
diag	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,
            ivector * u )
```

Create vector data space of INT type.

Parameters

m	Number of rows
u	Pointer to ivector (OUTPUT)

Author

Chensong Zhang

Date

2010/04/06

Definition at line 125 of file AuxVector.c.

9.43.2.11 fasp_ivec_create()

```
ivector fasp_ivec_create (
            const INT m )
```

Create vector data space of INT type.

Parameters

m	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

u	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

n	Number of variables
m	Integer value of ivector
u	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.

```c

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_dvec_isnan (const dvector *u)
{
    INT i;

    for ( i = 0; i < u->row; i++ ) {
        if ( isnan(u->val[i]) ) return TRUE;
    }

    return FALSE;
}

dvector fasp_dvec_create (const INT m)
{
    dvector u;

    u.row = m;
    u.val = (REAL *)fasp_mem_calloc(m,sizeof(REAL));

    return u;
}

ivector fasp_ivec_create (const INT m)
{
    ivector u;

    u.row = m;
    u.val = (INT *)fasp_mem_calloc(m,sizeof(INT));

    return u;
}

void fasp_dvec_alloc (const INT  m,
                      dvector   *u)
{
    u->row = m;
    u->val = (REAL*)fasp_mem_calloc(m,sizeof(REAL));

    return;
}

void fasp_ivec_alloc (const INT  m,
                      ivector   *u)
{

    u->row = m;
    u->val = (INT*)fasp_mem_calloc(m,sizeof(INT));

    return;
}

void fasp_dvec_free (dvector *u)
{
    if ( u == NULL ) return;

    fasp_mem_free(u->val); u->val = NULL; u->row = 0;
}

void fasp_ivec_free (ivector *u)
{
    if ( u == NULL ) return;

    fasp_mem_free(u->val); u->val = NULL; u->row = 0;
}

void fasp_dvec_rand (const INT  n,
                     dvector   *x)
{
    const INT va = 0;
    const INT vb = n;

    INT s=1, i,j;

    srand(s);
    for ( i = 0; i < n; ++i ) {
        j = 1 + (INT) (((REAL)n)*rand()/(RAND_MAX+1.0));
        x->val[i] = (((REAL)j)-va)/(vb-va);
    }
    x->row = n;
}

void fasp_dvec_set (INT         n,
                    dvector    *x,
                    const REAL  val)
{
    INT   i;
    REAL *xpt = x->val;

    if ( n > 0 ) x->row = n;
    else n = x->row;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    if (val == 0.0) {

#ifdef _OPENMP
        if (n > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend)
            for (myid = 0; myid < nthreads; myid++ ) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                memset(&xpt[mybegin], 0x0, sizeof(REAL)*(myend-mybegin));
            }
        }
        else {
#endif
            memset(xpt, 0x0, sizeof(REAL)*n);
#ifdef _OPENMP
        }
#endif

    }

    else {

#ifdef _OPENMP
        if (n > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend)
            for (myid = 0; myid < nthreads; myid++ ) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                for (i=mybegin; i<myend; ++i) xpt[i]=val;
            }
        }
        else {
#endif
            for (i=0; i<n; ++i) xpt[i]=val;
#ifdef _OPENMP
        }
#endif

    }
}

void fasp_ivec_set (INT        n,
                    ivector   *u,
                    const INT  m)
{
    SHORT nthreads = 1, use_openmp = FALSE;
    INT   i;

    if ( n > 0 ) u->row = n;
    else n = u->row;

#ifdef _OPENMP
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
        INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) u->val[i] = m;
        }
    }
    else {
        for (i=0; i<n; ++i) u->val[i] = m;
    }
}

void fasp_dvec_cp (const dvector  *x,
                   dvector        *y)
{
    y->row = x->row;
    memcpy(y->val,x->val,x->row*sizeof(REAL));
}

REAL fasp_dvec_maxdiff (const dvector *x,
                        const dvector *y)
{
    const INT length = x->row;
    const REAL *xpt = x->val, *ypt = y->val;

    SHORT use_openmp = FALSE;
    INT   i;
    REAL  Linf = 0.0, diffi = 0.0;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( length > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if(use_openmp) {
#ifdef _OPENMP
        REAL temp = 0.;
#pragma omp parallel firstprivate(temp) private(myid, mybegin, myend, i, diffi)
        {
            myid = omp_get_thread_num();
            fasp_get_start_end(myid, nthreads, length, &mybegin, &myend);
            for(i=mybegin; i<myend; i++) {
                if ((diffi = ABS(xpt[i]-ypt[i])) > temp) temp = diffi;
            }
#pragma omp critical
            if(temp > Linf) Linf = temp;
        }
#endif
```

```
    }
    else {
        for (i=0; i<length; ++i) {
            if ((diffi = ABS(xpt[i]-ypt[i])) > Linf) Linf = diffi;
        }
    }

    return Linf;
}

void fasp_dvec_symdiagscale (dvector        *b,
                             const dvector  *diag)
{
    // information about dvector
    const INT    n = b->row;
    REAL      *val = b->val;

    // local variables
    SHORT use_openmp = FALSE;
    INT   i;

    if ( diag->row != n ) {
        printf("### ERROR: Sizes of diag = %d != dvector = %d!", diag->row, n);
        fasp_chkerr(ERROR_MISC, __FUNCTION__);
    }

#ifdef _OPENMP
    INT mybegin, myend, myid, nthreads;
    if ( n > OPENMP_HOLDS ){
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin,myend)
        for (myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) val[i] = val[i]/sqrt(diag->val[i]);
        }
#endif
    }
    else {
        for (i=0; i<n; ++i) val[i] = val[i]/sqrt(diag->val[i]);
    }

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.45 BlaArray.c File Reference

BLAS1 operations for arrays.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.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, length of x and y 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 arraies x and y.

9.45.1 Detailed Description

BLAS1 operations for arrays.

Note

This file contains Level-1 (Bla) functions. It requires: AuxThreads.c

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

n	Number of variables
a	Factor a
x	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

n	Number of variables
a	Factor a
x	Pointer to x
b	Factor b
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 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

a	REAL factor a
x	Pointer to the original array
y	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

a	REAL factor a
x	Pointer to the original array
y	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

a	REAL factor a
x	Pointer to the original array
y	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

n	Number of variables
a	Factor a
x	Pointer to x
y	Pointer to y
z	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

a	REAL factor a
x	Pointer to the original array 1
y	Pointer to the original array 2
z	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

a	REAL factor a
x	Pointer to the original array 1
y	Pointer to the original array 2
z	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

a	REAL factor a
x	Pointer to the original array 1
y	Pointer to the original array 2
z	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

a	REAL factor a
x	Pointer to the original array 1
y	Pointer to the original array 2
z	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.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
```

```
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_darray_ax (const INT    n,
                          const REAL   a,
                          REAL        *x)
{
    if ( a == 1.0 ) return; // do nothing

    {
        SHORT use_openmp = FALSE;
        INT   i;

#ifdef _OPENMP
        INT myid, mybegin, myend, nthreads;
        if ( n > OPENMP_HOLDS ) {
            use_openmp = TRUE;
            nthreads = fasp_get_num_threads();
        }
#endif

        if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
                for ( i = mybegin; i < myend; ++i ) x[i] *= a;
            }
#endif
        }
        else {
            for ( i = 0; i < n; ++i ) x[i] *= a;
        }
    }
}

void fasp_blas_darray_axpy (const INT    n,
                            const REAL   a,
                            const REAL  *x,
                            REAL        *y)
{
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if ( a == 1.0 ) {
        if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
                for ( i = mybegin; i < myend; ++i ) y[i] += x[i];
            }
#endif
        }
        else {
            for ( i = 0; i < n; ++i ) y[i] += x[i];
        }
    }

    else if ( a == -1.0 ) {
        if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
                for ( i = mybegin; i < myend; ++i ) y[i] -= x[i];
            }
#endif
        }
        else {
            for ( i = 0; i < n; ++i ) y[i] -= x[i];
        }
    }

    else {
        if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
                for ( i = mybegin; i < myend; ++i ) y[i] += a*x[i];
            }
#endif
        }
        else {
            for ( i = 0; i < n; ++i ) y[i] += a*x[i];
        }
    }
}

void fasp_blas_darray_axpy_nc2 (const REAL   a,
                                const REAL  *x,
                                REAL        *y)
{
    y[0] += a*x[0];
    y[1] += a*x[1];

    y[2] += a*x[2];
    y[3] += a*x[3];
}

void fasp_blas_darray_axpy_nc3 (const REAL   a,
                                const REAL  *x,
                                REAL        *y)
{
    y[0] += a*x[0];
    y[1] += a*x[1];
    y[2] += a*x[2];

    y[3] += a*x[3];
    y[4] += a*x[4];
    y[5] += a*x[5];

    y[6] += a*x[6];
    y[7] += a*x[7];
    y[8] += a*x[8];
}

void fasp_blas_darray_axpy_nc5 (const REAL   a,
                                const REAL  *x,
                                REAL        *y)
{
    y[0] += a*x[0];
    y[1] += a*x[1];
    y[2] += a*x[2];
    y[3] += a*x[3];
    y[4] += a*x[4];

    y[5] += a*x[5];
    y[6] += a*x[6];
    y[7] += a*x[7];
    y[8] += a*x[8];
    y[9] += a*x[9];

    y[10] += a*x[10];
    y[11] += a*x[11];
    y[12] += a*x[12];
    y[13] += a*x[13];
    y[14] += a*x[14];

    y[15] += a*x[15];
    y[16] += a*x[16];
    y[17] += a*x[17];
    y[18] += a*x[18];
    y[19] += a*x[19];

    y[20] += a*x[20];
    y[21] += a*x[21];
    y[22] += a*x[22];
    y[23] += a*x[23];
    y[24] += a*x[24];
}

void fasp_blas_darray_axpy_nc7 (const REAL   a,
                                const REAL  *x,
                                REAL        *y)
{
    y[0] += a*x[0];
    y[1] += a*x[1];
    y[2] += a*x[2];
    y[3] += a*x[3];
    y[4] += a*x[4];
    y[5] += a*x[5];
    y[6] += a*x[6];

    y[7]  += a*x[7];
    y[8]  += a*x[8];
    y[9]  += a*x[9];
    y[10] += a*x[10];
    y[11] += a*x[11];
    y[12] += a*x[12];
    y[13] += a*x[13];

    y[14] += a*x[14];
    y[15] += a*x[15];
    y[16] += a*x[16];
    y[17] += a*x[17];
    y[18] += a*x[18];
    y[19] += a*x[19];
    y[20] += a*x[20];

    y[21] += a*x[21];
    y[22] += a*x[22];
    y[23] += a*x[23];
    y[24] += a*x[24];
    y[25] += a*x[25];
    y[26] += a*x[26];
    y[27] += a*x[27];

    y[28] += a*x[28];
    y[29] += a*x[29];
    y[30] += a*x[30];
    y[31] += a*x[31];
    y[32] += a*x[32];
    y[33] += a*x[33];
    y[34] += a*x[34];

    y[35] += a*x[35];
    y[36] += a*x[36];
    y[37] += a*x[37];
    y[38] += a*x[38];
    y[39] += a*x[39];
    y[40] += a*x[40];
    y[41] += a*x[41];

    y[42] += a*x[42];
    y[43] += a*x[43];
    y[44] += a*x[44];
    y[45] += a*x[45];
    y[46] += a*x[46];
    y[47] += a*x[47];
    y[48] += a*x[48];
}

void fasp_blas_darray_axpyz (const INT    n,
                             const REAL   a,
                             const REAL  *x,
                             const REAL  *y,
                             REAL        *z)
{
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
            for ( i = mybegin; i < myend; ++i ) z[i] = a*x[i] + y[i];
        }
#endif
    }
    else {
        for ( i = 0; i < n; ++i ) z[i] = a*x[i] + y[i];
    }
}

void fasp_blas_darray_axpyz_nc2 (const REAL   a,
                                 const REAL  *x,
                                 const REAL  *y,
                                 REAL        *z)
{
    z[0] = a*x[0] + y[0];
    z[1] = a*x[1] + y[1];

    z[2] = a*x[2] + y[2];
    z[3] = a*x[3] + y[3];
}

void fasp_blas_darray_axpyz_nc3 (const REAL   a,
                                 const REAL  *x,
                                 const REAL  *y,
                                 REAL        *z)
{
    z[0] = a*x[0] + y[0];
    z[1] = a*x[1] + y[1];
    z[2] = a*x[2] + y[2];

    z[3] = a*x[3] + y[3];
    z[4] = a*x[4] + y[4];
    z[5] = a*x[5] + y[5];

    z[6] = a*x[6] + y[6];
    z[7] = a*x[7] + y[7];
    z[8] = a*x[8] + y[8];
}

void fasp_blas_darray_axpyz_nc5 (const REAL   a,
                                 const REAL  *x,
                                 const REAL  *y,
                                 REAL        *z)
{
    z[0] = a*x[0] + y[0];
    z[1] = a*x[1] + y[1];
    z[2] = a*x[2] + y[2];
    z[3] = a*x[3] + y[3];
    z[4] = a*x[4] + y[4];

    z[5] = a*x[5] + y[5];
    z[6] = a*x[6] + y[6];
    z[7] = a*x[7] + y[7];
    z[8] = a*x[8] + y[8];
    z[9] = a*x[9] + y[9];

    z[10] = a*x[10] + y[10];
    z[11] = a*x[11] + y[11];
    z[12] = a*x[12] + y[12];
    z[13] = a*x[13] + y[13];
    z[14] = a*x[14] + y[14];

    z[15] = a*x[15] + y[15];
    z[16] = a*x[16] + y[16];
    z[17] = a*x[17] + y[17];
    z[18] = a*x[18] + y[18];
    z[19] = a*x[19] + y[19];

    z[20] = a*x[20] + y[20];
    z[21] = a*x[21] + y[21];
    z[22] = a*x[22] + y[22];
    z[23] = a*x[23] + y[23];
    z[24] = a*x[24] + y[24];
}

void fasp_blas_darray_axpyz_nc7 (const REAL   a,
                                 const REAL  *x,
                                 const REAL  *y,
                                 REAL        *z)
{
    z[0] = a*x[0] + y[0];
    z[1] = a*x[1] + y[1];
    z[2] = a*x[2] + y[2];
    z[3] = a*x[3] + y[3];
    z[4] = a*x[4] + y[4];
    z[5] = a*x[5] + y[5];
    z[6] = a*x[6] + y[6];

    z[7] = a*x[7] + y[7];
    z[8] = a*x[8] + y[8];
    z[9] = a*x[9] + y[9];
    z[10] = a*x[10] + y[10];
    z[11] = a*x[11] + y[11];
    z[12] = a*x[12] + y[12];
    z[13] = a*x[13] + y[13];

    z[14] = a*x[14] + y[14];
    z[15] = a*x[15] + y[15];
    z[16] = a*x[16] + y[16];
    z[17] = a*x[17] + y[17];
    z[18] = a*x[18] + y[18];
    z[19] = a*x[19] + y[19];
    z[20] = a*x[20] + y[20];

    z[21] = a*x[21] + y[21];
    z[22] = a*x[22] + y[22];
    z[23] = a*x[23] + y[23];
    z[24] = a*x[24] + y[24];
    z[25] = a*x[25] + y[25];
    z[26] = a*x[26] + y[26];
    z[27] = a*x[27] + y[27];

    z[28] = a*x[28] + y[28];
    z[29] = a*x[29] + y[29];
    z[30] = a*x[30] + y[30];
    z[31] = a*x[31] + y[31];
    z[32] = a*x[32] + y[32];
    z[33] = a*x[33] + y[33];
    z[34] = a*x[34] + y[34];

    z[35] = a*x[35] + y[35];
    z[36] = a*x[36] + y[36];
    z[37] = a*x[37] + y[37];
    z[38] = a*x[38] + y[38];
    z[39] = a*x[39] + y[39];
    z[40] = a*x[40] + y[40];
    z[41] = a*x[41] + y[41];

    z[42] = a*x[42] + y[42];
    z[43] = a*x[43] + y[43];
    z[44] = a*x[44] + y[44];
    z[45] = a*x[45] + y[45];
    z[46] = a*x[46] + y[46];
    z[47] = a*x[47] + y[47];
    z[48] = a*x[48] + y[48];
}

void fasp_blas_darray_axpby (const INT   n,
                             const REAL  a,
                             const REAL *x,
                             const REAL  b,
                             REAL       *y)
{
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
            for ( i = mybegin; i < myend; ++i ) y[i] = a*x[i] + b*y[i];
        }
#endif
    }
    else {
        for ( i = 0; i < n; ++i ) y[i] = a*x[i] + b*y[i];
    }

}

REAL fasp_blas_darray_norm1 (const INT    n,
                             const REAL  *x)
{
    register REAL onenorm = 0.0;
    INT   i;

#ifdef _OPENMP
#pragma omp parallel for reduction(+:onenorm) private(i)
#endif
    for ( i = 0; i < n; ++i ) onenorm += ABS(x[i]);

    return onenorm;
}

REAL fasp_blas_darray_norm2 (const INT    n,
                             const REAL  *x)
{
    register REAL twonorm = 0.0;
    INT  i;

#ifdef _OPENMP
#pragma omp parallel for reduction(+:twonorm) private(i)
#endif
    for ( i = 0; i < n; ++i ) twonorm += x[i] * x[i];

    return sqrt(twonorm);
}

REAL fasp_blas_darray_norminf (const INT    n,
                               const REAL  *x)
{
    SHORT use_openmp = FALSE;
    register REAL infnorm = 0.0;
    INT   i;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if ( use_openmp ) {
#ifdef _OPENMP
        REAL infnorm_loc = 0.0;
#pragma omp parallel firstprivate(infnorm_loc) private(myid, mybegin, myend, i)
        {
            myid = omp_get_thread_num();
            fasp_get_start_end (myid, nthreads, n, &mybegin, &myend);
            for ( i = mybegin; i < myend; ++i )
                infnorm_loc = MAX( infnorm_loc, ABS(x[i]) );

            if ( infnorm_loc > infnorm ) {
#pragma omp critical
                infnorm = MAX( infnorm_loc, infnorm );
            }
        }
#endif
    }
    else {
        for ( i = 0; i < n; ++i ) infnorm = MAX( infnorm, ABS(x[i]) );
    }

    return infnorm;
}

REAL fasp_blas_darray_dotprod (const INT    n,
                               const REAL  *x,
                               const REAL  *y)
{
    SHORT use_openmp = FALSE;
    register REAL value = 0.0;
    INT   i;

#ifdef _OPENMP
    if ( n > OPENMP_HOLDS ) use_openmp = TRUE;
#endif

    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+:value) private(i)
#endif
        for ( i = 0; i < n; ++i ) value += x[i]*y[i];
    }
    else {
        for ( i = 0; i < n; ++i ) value += x[i]*y[i];
    }

    return value;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.47 BlaEigen.c File Reference

Computing the extreme eigenvalues.

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

Functions

- REAL fasp_dcsr_maxeig (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

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

A	Pointer to the dCSRmat matrix
tol	Tolerance for stopping the power method
maxit	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_functs.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_mxv(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
```

```
        // convergence test
        if ( (ABS(eigenvalue - temp)/ABS(temp)) < tol ) break;

        fasp_dvec_cp(&y, &x);
        temp = eigenvalue;
    }

    // clean up memory
    fasp_dvec_free(&x);
    fasp_dvec_free(&y);

    return eigenvalue;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.
- dCSRLmat ∗ fasp_format_dcsrl_dcsr (const dCSRmat ∗A)
  Convert a dCSRmat into a dCSRLmat.
- 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

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

Ab	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

B	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

B	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

A	Pointer to dCOOmat matrix
B	Pointer to dCSRmat matrix

Returns

FASP_SUCCESS if successed; 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

A	Pointer to the dCSRmat type matrix
nb	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

A	Pointer to dCSRmat matrix
B	Pointer to dCOOmat matrix

Returns

FASP_SUCCESS if successed; 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_dcsrl_dcsr()

```
dCSRLmat * fasp_format_dcsrl_dcsr (
                const dCSRmat * A )
```

Convert a dCSRmat into a dCSRLmat.

Parameters

A	Pointer to dCSRLmat matrix

Returns

Pointer to dCSRLmat 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

B	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

A	Pointer to dSTRmat matrix
B	Pointer to dCSRmat matrix

Returns

FASP_SUCCESS if successed; 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.

```c

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_format_dcoo_dcsr (const dCOOmat  *A,
                             dCSRmat        *B)
{
    const INT m=A->row, n=A->col, nnz=A->nnz;
    INT  iind, jind, i;

    fasp_dcsr_alloc(m,n,nnz,B);
    INT *ia = B->IA;

    INT *ind = (INT *) fasp_mem_calloc(m+1,sizeof(INT));
    memset(ind, 0, sizeof(INT)*(m+1)); // initialize ind
    for ( i=0; i<nnz; ++i ) ind[A->rowind[i]+1]++; // count nnz in each row

    ia[0] = 0; // first index starting from zero
    for ( i=1; i<=m; ++i ) {
        ia[i]  = ia[i-1]+ind[i]; // set row_idx
        ind[i] = ia[i];
    }

    // loop over nnz and set col_idx and val
    for ( i=0; i<nnz; ++i ) {
        iind = A->rowind[i]; jind = ind[iind];
        B->JA [jind] = A->colind[i];
        B->val[jind] = A->val[i];
        ind[iind]    = ++jind;
    }

    fasp_mem_free(ind); ind = NULL;

    return FASP_SUCCESS;
}

SHORT fasp_format_dcsr_dcoo (const dCSRmat  *A,
                             dCOOmat        *B)
{
    const INT m=A->row, nnz=A->nnz;
    INT i, j;

    B->rowind = (INT *)fasp_mem_calloc(nnz,sizeof(INT));
    B->colind = (INT *)fasp_mem_calloc(nnz,sizeof(INT));
    B->val    = (REAL *)fasp_mem_calloc(nnz,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(m>OPENMP_HOLDS) private(i, j)
#endif
    for (i=0;i<m;++i) {
        for (j=A->IA[i];j<A->IA[i+1];++j) B->rowind[j]=i;
    }

    memcpy(B->colind, A->JA, nnz*sizeof(INT));
    memcpy(B->val, A->val, nnz*sizeof(REAL));

    return FASP_SUCCESS;
}

SHORT fasp_format_dstr_dcsr (const dSTRmat  *A,
                             dCSRmat        *B)
{
    // some members of A
    const INT nc    = A->nc;
    const INT ngrid = A->ngrid;
    const INT nband = A->nband;
    const INT *offsets = A->offsets;

    REAL  *diag = A->diag;
    REAL **offdiag = A->offdiag;

    // some members of B
    const INT glo_row = nc*ngrid;
    INT glo_nnz;
    INT *ia = NULL;
    INT *ja = NULL;
    REAL *a = NULL;

    dCSRmat B_tmp;

    // local variables
    INT width;
    INT nc2 = nc*nc;
    INT BAND,ROW,COL;
    INT ncb,nci;
    INT row_start,col_start;
    INT block; // how many blocks in the current ROW
    INT i,j;
    INT pos;
    INT start;
    INT val_L_start,val_R_start;
    INT row;
    INT tmp_col;
    REAL tmp_val;

    // allocate for 'ia' array
    ia = (INT *)fasp_mem_calloc(glo_row+1,sizeof(INT));

    // Generate the 'ia' array
    ia[0] = 0;
    for (ROW = 0; ROW < ngrid; ++ROW) {
        block = 1; // diagonal block
        for (BAND = 0; BAND < nband; ++BAND) {
            width = offsets[BAND];
            COL   = ROW + width;
            if (width < 0) {
                if (COL >= 0) ++block;
            }
            else {
                if (COL < ngrid) ++block;
            }
        } // end for BAND

        ncb = nc*block;
        row_start = ROW*nc;

        for (i = 0; i < nc; i ++) {
            row = row_start + i;
            ia[row+1] = ia[row] + ncb;
        }
    } // end for ROW

    // allocate for 'ja' and 'a' arrays
    glo_nnz = ia[glo_row];
    ja = (INT *)fasp_mem_calloc(glo_nnz,sizeof(INT));
    a = (REAL *)fasp_mem_calloc(glo_nnz,sizeof(REAL));

    // Generate the 'ja' and 'a' arrays at the same time
    for (ROW = 0; ROW < ngrid; ++ROW) {
        row_start = ROW*nc;
        val_L_start = ROW*nc2;

        // deal with the diagonal band
        for (i = 0; i < nc; i ++) {
            nci   = nc*i;
            row   = row_start + i;
            start = ia[row];
            for (j = 0; j < nc; j ++) {
                pos     = start + j;
                ja[pos] = row_start + j;
                a[pos]  = diag[val_L_start+nci+j];
            }
        }
        block = 1;

        // deal with the off-diagonal bands
        for (BAND = 0; BAND < nband; ++BAND) {
            width     = offsets[BAND];
            COL       = ROW + width;
            ncb       = nc*block;
            col_start = COL*nc;

            if (width < 0) {
                if (COL >= 0) {
                    val_R_start = COL*nc2;
                    for (i = 0; i < nc; i ++) {
                        nci = nc*i;
                        row = row_start + i;
                        start = ia[row];
                        for (j = 0 ; j < nc; j ++) {
                            pos     = start + ncb + j;
                            ja[pos] = col_start + j;
                            a[pos]  = offdiag[BAND][val_R_start+nci+j];
                        }
                    }
                    ++block;
                }
            }
            else {
                if (COL < ngrid) {
                    for (i = 0; i < nc; i ++) {
                        nci = nc*i;
                        row = row_start + i;
                        start = ia[row];
                        for (j = 0; j < nc; j ++) {
                            pos = start + ncb + j;
                            ja[pos] = col_start + j;
                            a[pos]  = offdiag[BAND][val_L_start+nci+j];
                        }
                    }
                    ++block;
                }
            }
        }
    }

    // Reordering in such manner that every diagonal element
    // is firstly stored in the corresponding row
    if (nc > 1) {
        for (ROW = 0; ROW < ngrid; ++ROW) {
            row_start = ROW*nc;
            for (j = 1; j < nc; j ++) {
                row   = row_start + j;
                start = ia[row];
                pos   = start + j;

                // swap in 'ja'
                tmp_col   = ja[start];
                ja[start] = ja[pos];
                ja[pos]   = tmp_col;

                // swap in 'a'
                tmp_val  = a[start];
                a[start] = a[pos];
                a[pos]   = tmp_val;
            }
        }
    }

    /* fill all the members of B_tmp */
    B_tmp.row = glo_row;
    B_tmp.col = glo_row;
    B_tmp.nnz = glo_nnz;
    B_tmp.IA = ia;
    B_tmp.JA = ja;
    B_tmp.val = a;

    *B = B_tmp;

    return FASP_SUCCESS;
}

dCSRmat fasp_format_dblc_dcsr (const dBLCmat *Ab)
{
    const INT mb=Ab->brow, nb=Ab->bcol, nbl=mb*nb;
    dCSRmat **blockptr=Ab->blocks, *blockptrij, A;

    INT i,j,ij,ir,i1,length,ilength,start,irmrow,irmrow1;
    INT *row, *col;
    INT m=0,n=0,nnz=0;

    row = (INT *)fasp_mem_calloc(mb+1,sizeof(INT));
    col = (INT *)fasp_mem_calloc(nb+1,sizeof(INT));

    // count the size of A
    row[0]=0; col[0]=0;
    for (i=0;i<mb;++i) { m+=blockptr[i*nb]->row; row[i+1]=m; }
    for (i=0;i<nb;++i) { n+=blockptr[i]->col;    col[i+1]=n; }

#ifdef _OPENMP
#pragma omp parallel for reduction(+:nnz) if (nbl>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<nbl;++i) { nnz+=blockptr[i]->nnz; }

    // memory space allocation
    A = fasp_dcsr_create(m,n,nnz);

    // set dCSRmat for A
    A.IA[0]=0;
    for (i=0;i<mb;++i) {

        for (ir=row[i];ir<row[i+1];ir++) {

            for (length=j=0;j<nb;++j) {
                ij=i*nb+j; blockptrij=blockptr[ij];
                if (blockptrij->nnz>0) {
                    start=A.IA[ir]+length;
                    irmrow=ir-row[i]; irmrow1=irmrow+1;
                    ilength=blockptrij->IA[irmrow1]-blockptrij->IA[irmrow];
                    if (ilength>0) {

memcpy(&(A.val[start]),&(blockptrij->val[blockptrij->IA[irmrow]]),ilength*sizeof(REAL));
                        memcpy(&(A.JA[start]), &(blockptrij->JA[blockptrij->IA[irmrow]]),
ilength*sizeof(INT));
                        for (i1=0;i1<ilength;i1++) A.JA[start+i1]+=col[j];
                        length+=ilength;
                    }
                }
            } // end for j

            A.IA[ir+1]=A.IA[ir]+length;
        } // end for ir

    } // end for i

    fasp_mem_free(row); row = NULL;
    fasp_mem_free(col); col = NULL;

    return(A);
}

dCSRLmat * fasp_format_dcsrl_dcsr (const dCSRmat *A)
{
    REAL    *DATA         = A -> val;
    INT     *IA           = A -> IA;
    INT     *JA           = A -> JA;
    INT     num_rows     = A -> row;
    INT     num_cols     = A -> col;
    INT     num_nonzeros = A -> nnz;

    dCSRLmat *B        = NULL;
    INT       dif;
    INT      *nzdifnum = NULL;
    INT      *rowstart = NULL;
    INT      *rowindex = (INT *)fasp_mem_calloc(num_rows, sizeof(INT));
    INT      *ja       = (INT *)fasp_mem_calloc(num_nonzeros, sizeof(INT));
    REAL     *data     = (REAL *)fasp_mem_calloc(num_nonzeros, sizeof(REAL));

    /* auxiliary arrays */
    INT *nzrow    = (INT *)fasp_mem_calloc(num_rows, sizeof(INT));
    INT *counter  = NULL;
    INT *invnzdif = NULL;

    INT i,j,k,cnt,maxnzrow;

    //-----------------------------------------
    //  Generate 'nzrow' and 'maxnzrow'
    //-----------------------------------------

    maxnzrow = 0;
    for (i = 0; i < num_rows; i ++) {
        nzrow[i] = IA[i+1] - IA[i];
        if (nzrow[i] > maxnzrow) {
            maxnzrow = nzrow[i];
        }
    }
    /* generate 'counter' */
    counter = (INT *)fasp_mem_calloc(maxnzrow + 1, sizeof(INT));

    for (i = 0; i < num_rows; i ++) {
        counter[nzrow[i]] ++;
    }

    //--------------------------------------------
    //  Determine 'dif'
    //--------------------------------------------

    for (dif = 0, i = 0; i < maxnzrow + 1; i ++) {
        if (counter[i] > 0) dif ++;
    }

    //--------------------------------------------
    //  Generate the 'nzdifnum' and 'rowstart'
    //--------------------------------------------

    nzdifnum = (INT *)fasp_mem_calloc(dif, sizeof(INT));
    invnzdif = (INT *)fasp_mem_calloc(maxnzrow + 1, sizeof(INT));
    rowstart = (INT *)fasp_mem_calloc(dif + 1, sizeof(INT));
    rowstart[0] = 0;
    for (cnt = 0, i = 0; i < maxnzrow + 1; i ++) {
        if (counter[i] > 0) {
            nzdifnum[cnt] = i;
            invnzdif[i] = cnt;
            rowstart[cnt+1] = rowstart[cnt] + counter[i];
            cnt ++;
        }
    }

    //--------------------------------------------
    //  Generate the 'rowindex'
    //--------------------------------------------

    for (i = 0; i < num_rows; i ++) {
        j = invnzdif[nzrow[i]];
        rowindex[rowstart[j]] = i;
        rowstart[j] ++;
    }
    /* recover 'rowstart' */
    for (i = dif; i > 0; i --) {
        rowstart[i] = rowstart[i-1];
    }
    rowstart[0] = 0;

    //--------------------------------------------
    //  Generate the 'data' and 'ja'
    //--------------------------------------------

    for (cnt = 0, i = 0; i < num_rows; i ++) {
        k = rowindex[i];
        for (j = IA[k]; j < IA[k+1]; j ++) {
            data[cnt] = DATA[j];
            ja[cnt] = JA[j];
            cnt ++;
        }
    }

    //-----------------------------------------------------------
    //  Create and fill a dCSRLmat B
    //-----------------------------------------------------------

    B = fasp_dcsrl_create(num_rows, num_cols, num_nonzeros);
    B -> dif      = dif;
    B -> nz_diff  = nzdifnum;
    B -> index    = rowindex;
    B -> start    = rowstart;
    B -> ja       = ja;
    B -> val      = data;

    //----------------------------
    //  Free the auxiliary arrays
    //----------------------------

    free(nzrow);
    free(counter);
    free(invnzdif);

    return B;
}

dCSRmat fasp_format_dbsr_dcsr (const dBSRmat *B)
{
    dCSRmat A;

    /* members of B */
    INT     ROW = B->ROW;
    INT     COL = B->COL;
    INT     NNZ = B->NNZ;
    INT     nb  = B->nb;
    INT    *IA  = B->IA;
    INT    *JA  = B->JA;
    REAL   *val = B->val;

    INT     storage_manner = B->storage_manner;

    INT jump = nb*nb;
    INT rowA = ROW*nb;
    INT colA = COL*nb;
    INT nzA  = NNZ*jump;

    INT     *ia = NULL;
    INT     *ja = NULL;
    REAL    *a  = NULL;

    INT i,j,k;
    INT mr,mc;
    INT rowstart0,rowstart,colstart0,colstart;
    INT colblock,nzperrow;

    REAL  *vp = NULL;
    REAL  *ap = NULL;
    INT  *jap = NULL;

    SHORT use_openmp = FALSE;

#ifdef _OPENMP
    INT stride_i,mybegin,myend,myid,nthreads;
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    //--------------------------------------------------------
    // Create a CSR Matrix
    //--------------------------------------------------------
    A  = fasp_dcsr_create(rowA, colA, nzA);
    ia = A.IA;
    ja = A.JA;
    a  = A.val;

    //-----------------------------------------------------------------------------
    // Compute the number of nonzeros per row, and after this loop,
    // ia[i],i=1:rowA, will be the number of nonzeros of the (i-1)-th row.
    //-----------------------------------------------------------------------------

    if (use_openmp) {
#ifdef _OPENMP
        stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend, i, rowstart, colblock, nzperrow, j)
        {
            myid = omp_get_thread_num();
            mybegin = myid*stride_i;
            if(myid < nthreads-1) myend = mybegin+stride_i;
            else myend = ROW;
            for (i=mybegin; i<myend; ++i)
            {
                rowstart = i*nb + 1;
                colblock = IA[i+1] - IA[i];
                nzperrow = colblock*nb;
                for (j = 0; j < nb; ++j)
                {
                    ia[rowstart+j] = nzperrow;
                }
            }
        }
#endif
    }
    else {
        for (i = 0; i < ROW; ++i)
        {
            rowstart = i*nb + 1;
            colblock = IA[i+1] - IA[i];
            nzperrow = colblock*nb;
            for (j = 0; j < nb; ++j)
            {
                ia[rowstart+j] = nzperrow;
            }
        }
    }

    //-----------------------------------------------------
    // Generate the real 'ia' for CSR of A
    //-----------------------------------------------------

    ia[0] = 0;
    for (i = 1; i <= rowA; ++i)
    {
        ia[i] += ia[i-1];
    }

    //-----------------------------------------------------
    // Generate 'ja' and 'a' for CSR of A
    //-----------------------------------------------------

    switch (storage_manner)
    {
        case 0:  // each non-zero block elements are stored in row-major order
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, k, j, rowstart, colstart, vp, mr, ap, jap, mc)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if(myid < nthreads-1) myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i<myend; ++i)
                    {
                        for (k = IA[i]; k < IA[i+1]; ++k)
                        {
                            j = JA[k];
                            rowstart = i*nb;
                            colstart = j*nb;
                            vp = &val[k*jump];
                            for (mr = 0; mr < nb; mr ++)
                            {
                                ap  = &a[ia[rowstart]];
                                jap = &ja[ia[rowstart]];
                                for (mc = 0; mc < nb; mc ++)
                                {
                                    *ap = *vp;
                                    *jap = colstart + mc;
                                    vp ++; ap ++; jap ++;
                                }
                                ia[rowstart] += nb;
                                rowstart ++;
                            }
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i)
                {
                    for (k = IA[i]; k < IA[i+1]; ++k)
                    {
                        j = JA[k];
                        rowstart = i*nb;
                        colstart = j*nb;
                        vp = &val[k*jump];
                        for (mr = 0; mr < nb; mr ++)
                        {
                            ap  = &a[ia[rowstart]];
                            jap = &ja[ia[rowstart]];
                            for (mc = 0; mc < nb; mc ++)
                            {
                                *ap = *vp;
                                *jap = colstart + mc;
                                vp ++; ap ++; jap ++;
                            }
                            ia[rowstart] += nb;
                            rowstart ++;
                        }
                    }
                }
            }
        }
            break;

        case 1:  // each non-zero block elements are stored in column-major order
        {
            for (i = 0; i < ROW; ++i)
            {
                for (k = IA[i]; k < IA[i+1]; ++k)
                {
                    j = JA[k];
                    rowstart0 = i*nb;
                    colstart0 = j*nb;
                    vp = &val[k*jump];
                    for (mc = 0; mc < nb; mc ++)
                    {
                        rowstart = rowstart0;
                        colstart = colstart0 + mc;
                        for (mr = 0; mr < nb; mr ++)
                        {
                            a[ia[rowstart]] = *vp;
                            ja[ia[rowstart]] = colstart;
                            vp ++; ia[rowstart]++; rowstart++;
                        }
                    }
                }
            }
        }
            break;
    }

    //-----------------------------------------------------
    // Map back the real 'ia' for CSR of A
    //-----------------------------------------------------

    for (i = rowA; i > 0; i --) {
        ia[i] = ia[i-1];
    }
    ia[0] = 0;

    return (A);
}

dBSRmat fasp_format_dcsr_dbsr (const dCSRmat  *A,
                                const INT       nb)
{
    INT i, j, k, ii, jj, kk, l, mod, nnz;
    INT row    = A->row/nb;
    INT col    = A->col/nb;
    INT nb2    = nb*nb;
    INT *IA    = A->IA;
    INT *JA    = A->JA;
    REAL *val = A->val;

    dBSRmat B;   // Safe-guard check
    INT *col_flag, *ia, *ja;
    REAL *bval;

    if ((A->row)%nb!=0) {
        printf("### ERROR: A.row=%d is not a multiplication of nb=%d!\n",
               A->row, nb);
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    if ((A->col)%nb!=0) {
        printf("### ERROR: A.col=%d is not a multiplication of nb=%d!\n",
               A->col, nb);
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    B.ROW = row;
    B.COL = col;
    B.nb  = nb;
    B.storage_manner = 0;

    // allocate memory for B
    col_flag = (INT *)fasp_mem_calloc(col, sizeof(INT));
    ia = (INT *) fasp_mem_calloc(row+1, sizeof(INT));

    fasp_iarray_set(col, col_flag, -1);

    // Get ia for BSR format
    nnz = 0;
    for (i=0; i<row; ++i) {
        ii = nb*i;
        for (j=0; j<nb; ++j) {
            jj = ii+j;
            for (k=IA[jj]; k<IA[jj+1]; ++k) {
                kk = JA[k]/nb;
                if (col_flag[kk]!=0) {
                    col_flag[kk] = 0;
                    //ja[nnz] = kk;
                    nnz ++;
                }
            }
        }
        ia[i+1] = nnz;
        fasp_iarray_set(col, col_flag, -1);
    }

    // set NNZ
    B.NNZ = nnz;

    // allocate ja and bval
    ja = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    bval = (REAL*)fasp_mem_calloc(nnz*nb2, sizeof(REAL));

    // Get ja for BSR format
    nnz = 0;
    for (i=0; i<row; ++i) {
        ii = nb*i;
        for(j=0; j<nb; ++j) {
            jj = ii+j;
            for(k=IA[jj]; k<IA[jj+1]; ++k) {
                kk = JA[k]/nb;
                if (col_flag[kk]!=0) {
                    col_flag[kk] = 0;
                    ja[nnz] = kk;
                    nnz ++;
                }
            }
        }
        ia[i+1] = nnz;
        fasp_iarray_set(col, col_flag, -1);
    }

    // Get non-zeros of BSR
    for (i=0; i<row; ++i) {
        ii = nb*i;
        for (j=0; j<nb; ++j) {
            jj = ii+j;
            for (k=IA[jj]; k<IA[jj+1]; ++k) {
                for (l=ia[i]; l<ia[i+1]; ++l) {
                    if (JA[k]/nb ==ja[l]) {
                        mod = JA[k]%nb;
                        bval[l*nb2+j*nb+mod] = val[k];
                        break;
                    }
                }
            }
        }
    }

    B.IA = ia;
    B.JA = ja;
    B.val = bval;

    fasp_mem_free(col_flag); col_flag = NULL;

    return B;
}

dBSRmat fasp_format_dstr_dbsr (const dSTRmat *B)
{
    // members of 'B'
    INT      nc      = B->nc;
    INT      ngrid   = B->ngrid;
    REAL    *diag    = B->diag;
    INT      nband   = B->nband;
    INT     *offsets = B->offsets;
    REAL   **offdiag = B->offdiag;

    // members of 'A'
    dBSRmat  A;
    INT      NNZ;
    INT     *IA  = NULL;
    INT     *JA  = NULL;
    REAL    *val = NULL;

    // local variables
    INT i,j,k,m;
    INT nc2 = nc*nc;
    INT ngridplus1 = ngrid + 1;

    // compute NNZ
    NNZ = ngrid;
    for (i = 0; i < nband; ++i) {
        NNZ += (ngrid - abs(offsets[i]));
    }

    // Create and Initialize a dBSRmat 'A'
    A = fasp_dbsr_create(ngrid, ngrid, NNZ, nc, 0);
    IA = A.IA;
    JA = A.JA;
    val = A.val;

    // Generate 'IA'
    for (i = 1; i < ngridplus1; ++i) IA[i] = 1; // take the diagonal blocks into account
    for (i = 0; i < nband; ++i) {
        k = offsets[i];
        if (k < 0) {
            for (j = -k+1; j < ngridplus1; ++j) {
                IA[j] ++;
            }
        }
        else {
            m = ngridplus1 - k;
            for (j = 1; j < m; ++j)
            {
                IA[j] ++;
            }
        }
    }
    IA[0] = 0;
    for (i = 1; i < ngridplus1; ++i) {
        IA[i] += IA[i-1];
    }

    // Generate 'JA' and 'val' at the same time
    for (i = 0 ; i < ngrid; ++i) {
        memcpy(val + IA[i]*nc2, diag + i*nc2, nc2*sizeof(REAL));
        JA[IA[i]] = i;
        IA[i] ++;
    }

    for (i = 0; i < nband; ++i) {
        k = offsets[i];
        if (k < 0) {
            for (j = -k; j < ngrid; ++j) {
                m = j + k;
                memcpy(val+IA[j]*nc2, offdiag[i]+m*nc2, nc2*sizeof(REAL));
                JA[IA[j]] = m;
                IA[j] ++;
            }
        }
        else {
            m = ngrid - k;
            for (j = 0; j < m; ++j) {
                memcpy(val + IA[j]*nc2, offdiag[i] + j*nc2, nc2*sizeof(REAL));
                JA[IA[j]] = k + j;
                IA[j] ++;
            }
        }
    }

    // Map back the real 'IA' for BSR of A
    for (i = ngrid; i > 0; i --) {
        IA[i] = IA[i-1];
    }
    IA[0] = 0;

    return (A);
}

dCOOmat * fasp_format_dbsr_dcoo (const dBSRmat *B)
{
    /* members of B */
    INT     ROW = B->ROW;
    INT     COL = B->COL;
    INT     NNZ = B->NNZ;
    INT     nb  = B->nb;
    INT    *IA  = B->IA;
    INT    *JA  = B->JA;
    REAL   *val = B->val;

    dCOOmat *A = NULL;
    INT      nb2 = nb*nb;
    INT      num_nonzeros = NNZ*nb2;
    INT     *rowA = NULL;
    INT     *colA = NULL;
    REAL    *valA = NULL;

    INT      i,j,k,inb;
    INT      row_start, col_start;
    INT      cnt,mr,mc;
    REAL    *pt = NULL;

    // Create and Initialize a dCOOmat 'A'
    A         = (dCOOmat *)fasp_mem_calloc(1, sizeof(dCOOmat));
    A->row    = ROW*nb;
    A->col    = COL*nb;
    A->nnz    = num_nonzeros;
    rowA      = (INT *)fasp_mem_calloc(num_nonzeros, sizeof(INT));
    colA      = (INT *)fasp_mem_calloc(num_nonzeros, sizeof(INT));
    valA      = (REAL *)fasp_mem_calloc(num_nonzeros, sizeof(REAL));
    A->rowind = rowA;
    A->colind = colA;
    A->val    = valA;

    cnt = 0;
    for (i = 0; i < ROW; ++i) {
        inb = i*nb;
        for (k = IA[i]; k < IA[i+1]; ++k) {
            j  = JA[k];
            pt = &val[k*nb2];
            row_start = inb;
            col_start = j*nb;
```

```
                for (mr = 0; mr < nb; mr ++) {
                    for (mc = 0; mc < nb; mc ++) {
                        rowA[cnt] = row_start;
                        colA[cnt] = col_start + mc;
                        valA[cnt] = (*pt);
                        pt ++;
                        cnt ++;
                    }
                    row_start ++;
                }
            }
        }

        return (A);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.51 BlaILU.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

Definition in file BlaILU.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

n	row number of A
a	nonzero entries of A
ja	integer array of column for A
ia	integer array of row pointers for A
lfil	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.
alu	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.
jlu	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
iwk	integer. The minimum length of arrays alu, jlu, and levs.
ierr	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.
nzlu	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 BlaILU.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

n	row number of A
a	nonzero entries of A
ja	integer array of column for A
ia	integer array of row pointers for A
lfil	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.
droptol	real*8. Sets the threshold for dropping small terms in the factorization. See below for details on dropping strategy.
alu	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.
jlu	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
iwk	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.
ierr	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.
nz	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 BlaILU.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

n	row number of A
a	nonzero entries of A
ja	integer array of column for A
ia	integer array of row pointers for A
lfil	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.
droptol	real∗8. Sets the threshold for dropping small terms in the factorization. See below for details on dropping strategy.
permtol	tolerance ratio used to determne whether or not to permute two columns. At step i columns i and j are permuted when abs(a(i,j))∗permtol .gt. abs(a(i,i)) [0 --> never permute; good values 0.1 to 0.01]
mbloc	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.
alu	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.
jlu	integer array of length n containing the pointers to the beginning of each row of U in the matrix alu,jlu.
iperm	permutation arrays
iwk	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.
ierr	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.
nz	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 BlaILU.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

n	row number of A
colind	integer array of column for A
rwptr	integer array of row pointers for A
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.
nzlu	integer pointer. Return number of nonzero entries for alu and jlu
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	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: ∗ _______________________ ∗ | | | ∗ | | | ∗ | j,j | j,k | ∗ | | | ∗ |__________|___________| ∗ | | | ∗ | | | ∗ | i,j | i,k | ∗ | | | ∗ |__________|___________| ∗

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: ∗ _______________________ ∗ | | | ∗ | | | ∗ | level 0 | level s2 | ∗ | | | ∗ |__________|_____↩
______| ∗ | | | ∗ | | | ∗ | level s1 | | ∗ | | | ∗ |__________|___________| ∗

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. rowll = integer array of length n containing pointers to implement a linked list for the fill-in elements.

external functions:

ifix, float, min0, srtr

Definition at line 1372 of file BlaILU.c.

9.52 BlaILU.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 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
```

```

iwk     = integer.  The minimum length of arrays alu, jlu, and levs.

On return:
===========

alu,jlu = 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.

jlu     = integer array of length n containing the pointers to
the beginning of each row of U in the matrix alu,jlu.

levs    = integer (work) array of size iwk -- which contains the
levels of each element in alu, jlu.

ierr    = integer.  Error message with the following meaning.
ierr  = 0    --> successful return.
ierr .gt.  0  --> zero pivot encountered at step number ierr.
ierr  = -1   --> Error.  input matrix may be wrong.
(The elimination process has generated a
row in L or U whose length is .gt.    n.)
ierr  = -2   --> The matrix L overflows the array al.
ierr  = -3   --> The matrix U overflows the array alu.
ierr  = -4   --> Illegal value for lfil.
ierr  = -5   --> zero row encountered in A or U.

work arrays:
=============
jw      = integer work array of length 3*n.
w       = real work array of length n

----------------------------------------------------------------------
w, ju (1:n) store the working array [1:ii-1 = L-part, ii:n = U-part]
jw(n+1:2n)  stores the nonzero indicator.

Notes:
------
All the diagonal elements of the input matrix must be nonzero.
---------------------------------------------------------------------- */

    // locals
    INT ju0, k, j1, j2, j, ii, i, lenl, lenu, jj, jrow, jpos, n2, jlev, NE;
    REAL t, s, fact;
    SHORT cinindex=0;
    REAL *w;
    INT *ju, *jw, *levs;

    if (lfil  <  0) goto F998;

    w = (REAL *)fasp_mem_calloc(n, sizeof(REAL));
    ju = (INT *)fasp_mem_calloc(n, sizeof(INT));
    jw = (INT *)fasp_mem_calloc(3*n, sizeof(INT));
    levs = (INT *)fasp_mem_calloc(iwk, sizeof(INT));

    --jw;
    --w;
    --ju;
    --jlu;
    --alu;
    --ia;
    --ja;
    --a;
    --levs;

    /*----------------------------------------------------------------------
shift index for C routines
-----------------------------------------------------------------------*/
    if (ia[1]   ==  0) cinindex=1 ;
    if (cinindex)
    {
        NE = n + 1; //modify by chunsheng 2012, Sep, 1;
        for (i=1; i<=NE; ++i)  ++ia[i];
        NE = ia[n+1] - 1;
        for (i=1; i<=NE; ++i)  ++ja[i];
    }

    /*----------------------------------------------------------------------
initialize ju0 (points to next element to be added to alu,jlu)
```

```
and pointer array.
-----------------------------------------------------------------------*/
    n2 = n + n;
    ju0 = n + 2;
    jlu[1] = ju0;

    // initialize nonzero indicator array + levs array --
    for(j = 1; j<=2*n; ++j) jw[j] = 0;

    /*-----------------------------------------------------------------------
beginning of main loop.
-----------------------------------------------------------------------*/
    for(ii = 1; ii <= n; ++ii)  {  //500
        j1 = ia[ii];
        j2 = ia[ii + 1] - 1;

        //   unpack L-part and U-part of row of A in arrays w
        lenu = 1;
        lenl = 0;
        jw[ii] = ii;
        w[ii] = 0.0;
        jw[n + ii] = ii;

        //
        for(j = j1; j <= j2; ++j)   { //170
            k = ja[j];
            t = a[j];
            if (t  ==  0.0) continue;  //goto g170;
            if (k  <  ii)   {
                ++lenl;
                jw[lenl] = k;
                w[lenl] = t;
                jw[n2 + lenl] = 0;
                jw[n + k] = lenl;
            } else if (k  ==  ii) {
                w[ii] = t;
                jw[n2 + ii] = 0;
            } else  {
                ++lenu;
                jpos = ii + lenu - 1;
                jw[jpos] = k;
                w[jpos] = t;
                jw[n2 + jpos] = 0;
                jw[n + k] = jpos;
            }

        }  //170

        jj = 0;
        //   eliminate previous rows

    F150:
        ++jj;
        if (jj  >  lenl) goto F160;

        /*-----------------------------------------------------------------------
in order to do the elimination in the correct order we must select
the smallest column index among jw(k), k=jj+1, ..., lenl.
-----------------------------------------------------------------------*/

        jrow = jw[jj];
        k = jj;

        //   determine smallest column index
        for(j = jj + 1; j <= lenl; ++j)     { //151
            if (jw[j]  <  jrow) {
                jrow = jw[j];
                k = j;
            }
        } //151

        if (k  !=  jj) {
            //     exchange in jw
            j = jw[jj];
            jw[jj] = jw[k];
            jw[k] = j;
            //     exchange in jw(n+  (pointers/ nonzero indicator).
            jw[n + jrow] = jj;
            jw[n + j] = k;
            //     exchange in jw(n2+  (levels)
            j = jw[n2 + jj];
            jw[n2 + jj] = jw[n2 + k];
            jw[n2 + k] = j;
            //     exchange in w
            s = w[jj];
            w[jj] = w[k];
            w[k] = s;
        }

        //   zero out element in row by resetting jw(n+jrow) to zero.
        jw[n + jrow] = 0;

        //   get the multiplier for row to be eliminated (jrow) + its level
        fact = w[jj]*alu[jrow];
        jlev = jw[n2 + jj];
        if (jlev  >  lfil) goto F150;

        //   combine current row and row jrow
        for(k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k ) { // 203
            s = fact*alu[k];
            j = jlu[k];
            jpos = jw[n + j];
            if (j  >=  ii) {
                //   dealing with upper part.
                if (jpos  ==  0) {
                    //   this is a fill-in element
                    ++lenu;
                    if (lenu  >  n) goto F995;
                    i = ii + lenu - 1;
                    jw[i] = j;
                    jw[n + j] = i;
                    w[i] = -s;
                    jw[n2 + i] = jlev + levs[k] + 1;
                } else  {
                    //   this is not a fill-in element
                    w[jpos] = w[jpos] - s;
                    jw[n2 + jpos] = MIN(jw[n2 + jpos], jlev + levs[k] + 1);
                }
            } else  {
                //   dealing with lower part.
                if (jpos  ==  0)    {
                    //   this is a fill-in element
                    ++lenl;
                    if (lenl  >  n) goto F995;
                    jw[lenl] = j;
                    jw[n + j] = lenl;
                    w[lenl] = -s;
                    jw[n2 + lenl] = jlev + levs[k] + 1;
                } else {
                    //   this is not a fill-in element
                    w[jpos] = w[jpos] - s;
                    jw[n2 + jpos] = MIN(jw[n2 + jpos], jlev + levs[k] + 1);
                }
            }

        } //203
        w[jj] = fact;
        jw[jj] = jrow;
        goto F150;

    F160:
        //  reset double-pointer to zero (U-part)
        for(k = 1; k <= lenu; ++k)  jw[n + jw[ii + k - 1]] = 0;

        //   update l-matrix
        for(k = 1; k <= lenl; ++k ) {   //204
            if (ju0  >  iwk) goto F996;
            if (jw[n2 + k]  <=  lfil)  {
                alu[ju0] = w[k];
                jlu[ju0] = jw[k];
                ++ju0;
            }
        } //204

        //   save pointer to beginning of row ii of U
        ju[ii] = ju0;

        //   update u-matrix
        for(k = ii + 1; k <= ii + lenu - 1; ++k ) {  //302
            if (ju0  >  iwk) goto F997;

            if (jw[n2 + k]  <=  lfil) {
                jlu[ju0] = jw[k];
                alu[ju0] = w[k];
                levs[ju0] = jw[n2 + k];
                ++ju0;
            }

        } //302

        if (w[ii]  ==  0.0) goto F999;
        //
        alu[ii] = 1.0/w[ii];

        //   update pointer to beginning of next row of U.
        jlu[ii + 1] = ju0;
        /*-----------------------------------------------------------------------
end main loop
-----------------------------------------------------------------------*/
    } //500

    *nzlu = ju[n] - 1;

    if (cinindex)  {
        for ( i = 1; i <= *nzlu; ++i )  --jlu[i];
    }

    *ierr = 0;

F100:
    ++jw;
    ++w;
    ++ju;
    ++jlu;
    ++alu;
    ++ia;
    ++ja;
    ++a;
    ++levs;

    fasp_mem_free(w);     w    = NULL;
    fasp_mem_free(ju);    ju   = NULL;
    fasp_mem_free(jw);    jw   = NULL;
    fasp_mem_free(levs);  levs = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;

    // incomprehensible error.  Matrix must be wrong.
F995:
    printf("### ERROR: Incomprehensible error.  [%s]\n", __FUNCTION__);
    *ierr = -1;
    goto F100;

    // insufficient storage in L.
F996:
    printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
    *ierr = -2;
    goto F100;

    // insufficient storage in U.
F997:
    printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
    *ierr = -3;
    goto F100;

    // illegal lfil entered.
F998:
    printf("### ERROR: Illegal lfil entered.  [%s]\n", __FUNCTION__);
    *ierr = -4;
    return;

    // zero row encountered in A or U.
F999:
    printf("### ERROR: Zero row encountered in A or U. [%s]\n", __FUNCTION__);
    *ierr = -5;
    goto F100;
    /*----------------end-of-iluk---------------------------------------------
---------------------------------------------------------------------- */
}
```

```

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)
{
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /*----------------------------------------------------------------------*
*** ILUT preconditioner ***                                           *
incomplete LU factorization with dual truncation mechanism       *
----------------------------------------------------------------------*
Author:  Yousef Saad *May, 5, 1990, Latest revision, August 1996  *
----------------------------------------------------------------------*
PARAMETERS
-----------

on entry:
==========
n       = integer.  The row dimension of the matrix A. The matrix

a,ja,ia = matrix stored in Compressed Sparse Row format.

lfil    = 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.

droptol = real*8.  Sets the threshold for dropping small terms in the
factorization.  See below for details on dropping strategy.

iwk     = 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.

On return:
===========

alu,jlu = 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.

ju      = integer array of length n containing the pointers to
the beginning of each row of U in the matrix alu,jlu.

ierr    = integer.  Error message with the following meaning.
ierr  = 0    --> successful return.
ierr .gt.  0  --> zero pivot encountered at step number ierr.
ierr  = -1   --> Error.  input matrix may be wrong.
(The elimination process has generated a
row in L or U whose length is .gt.    n.)
ierr  = -2   --> The matrix L overflows the array al.
ierr  = -3   --> The matrix U overflows the array alu.
ierr  = -4   --> Illegal value for lfil.
ierr  = -5   --> zero row encountered.

work arrays:
=============
jw      = integer work array of length 2*n.
w       = real work array of length n+1.

----------------------------------------------------------------------
w, ju (1:n) store the working array [1:ii-1 = L-part, ii:n = u]
jw(n+1:2n)  stores nonzero indicators

Notes:
------
The diagonal elements of the input matrix must be nonzero (at least
'structurally').

----------------------------------------------------------------------*
---- Dual drop strategy works as follows.                                    *
*
1) Theresholding in L and U as set by droptol.  Any element whose *
magnitude is less than some tolerance (relative to the abs       *
value of diagonal element in u) is dropped.                                  *
*
2) Keeping only the largest lfil elements in the i-th row of L   *
and the largest lfil elements in the i-th row of U (excluding    *
diagonal elements).
*
*
Flexibility:  one can use droptol=0 to get a strategy based on    *
keeping the largest elements in each row of L and U. Taking      *
droptol .ne.  0 but lfil=n will give the usual threshold strategy *
(however, fill-in is then unpredictible).                                  *
-------------------------------------------------------------------- */

    // locals
    INT ju0, k, j1, j2, j, ii, i, lenl, lenu, jj, jrow, jpos, NE, len;
    REAL t, s, fact, tmp;
    SHORT cinindex=0;
    REAL *w, *tnorm;
    INT  *ju, *jw;

    if (lfil  <  0) goto F998;

    ju = (INT *)fasp_mem_calloc(n, sizeof(INT));
    jw = (INT *)fasp_mem_calloc(2*n, sizeof(INT));
    w = (REAL *)fasp_mem_calloc(n+1, sizeof(REAL));
    tnorm = (REAL *)fasp_mem_calloc(n, sizeof(REAL));

    --jw;
    --ju;
    --w;
    --tnorm;
    --jlu;
    --alu;
    --ia;
    --ja;
    --a;

    if (ia[1]  ==  0) cinindex=1 ;

    if (cinindex)
    {
        NE = n + 1; //modify by chunsheng 2012, Sep, 1;
        for (i=1; i<=NE; ++i)  ++ia[i];
        NE = ia[n+1] - 1;
        for (i=1; i<=NE; ++i)  ++ja[i];
    }

    /*----------------------------------------------------------------------
initialize ju0 (points to next element to be added to alu,jlu)
and pointer array.
--------------------------------------------------------------------*/
    ju0 = n + 2;
    jlu[1] = ju0;

    // initialize nonzero indicator array.
    for (j = 1; j<=n; ++j)  jw[n + j] = 0;

    /*----------------------------------------------------------------------
beginning of main loop.
--------------------------------------------------------------------*/
    for (ii = 1; ii <= n; ++ii ) {
        j1 = ia[ii];
        j2 = ia[ii + 1] - 1;
        tmp = 0.0;
        for ( k = j1; k<= j2; ++k)  tmp = tmp + ABS(a[k]);
        tmp = tmp/(REAL)(j2 - j1 + 1);
        tnorm[ii] = tmp*droptol;;
    }

    for (ii = 1; ii<=n; ++ii) {
        j1 = ia[ii];
        j2 = ia[ii + 1] - 1;

        //   unpack L-part and U-part of row of A in arrays w
        lenu = 1;
        lenl = 0;
        jw[ii] = ii;
        w[ii] = 0.0;
        jw[n + ii] = ii;

        for(j = j1; j<=j2; ++j) {
            k = ja[j];
            t = a[j];
            if (k  <  ii) {
                ++lenl;
                jw[lenl] = k;
                w[lenl] = t;
                jw[n + k] = lenl;
            } else if (k == ii) {
                w[ii] = t;
            } else  {
                ++lenu ;
                jpos = ii + lenu - 1;
                jw[jpos] = k;
                w[jpos] = t;
                jw[n + k] = jpos;
            }
        }
        jj = 0;
        len = 0;

        //      eliminate previous rows
    F150:
        ++jj;
        if (jj  >  lenl) goto F160;

        /*-----------------------------------------------------------------------
in order to do the elimination in the correct order we must select
the smallest column index among jw(k), k=jj+1, ..., lenl.
-----------------------------------------------------------------------*/
        jrow = jw[jj];
        k = jj;

        /*
determine smallest column index
*/
        for(j = jj + 1; j<=lenl; ++j)   {  //151
            if (jw[j]  <  jrow) {
                jrow = jw[j];
                k = j;
            }
        }    //151

        if (k  !=  jj) {
            // exchange in jw
            j = jw[jj];
            jw[jj] = jw[k];
            jw[k] = j;
            // exchange in jr
            jw[n + jrow] = jj;
            jw[n + j] = k;
            // exchange in w
            s = w[jj];
            w[jj] = w[k];
            w[k] = s;
        }

        // zero out element in row by setting jw(n+jrow) to zero.
        jw[n + jrow] = 0;

        // get the multiplier for row to be eliminated (jrow).
        fact = w[jj]*alu[jrow];

        if (ABS(fact)  <=  droptol) goto F150;

        // combine current row and row jrow
        for ( k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k) {   //203
            s = fact*alu[k];
            j = jlu[k];
            jpos = jw[n + j];
            if (j  >=  ii)  {
                //      dealing with upper part.
                if (jpos  ==  0)
                {
                    //      this is a fill-in element
                    ++lenu;
                    if (lenu  >  n) goto F995;
                    i = ii + lenu - 1;
                    jw[i] = j;
                    jw[n + j] = i;
                    w[i] = -s;
                } else  {
                    //    this is not a fill-in element
                    w[jpos] = w[jpos] - s;
                }
            } else {
                //     dealing  with lower part.
                if (jpos  ==  0) {
                    //     this is a fill-in element
                    ++lenl;
                    if (lenl  >  n) goto F995;
                    jw[lenl] = j;
                    jw[n + j] = lenl;
                    w[lenl] = -s;
                } else  {
                    //    this is not a fill-in element
                    w[jpos] = w[jpos] - s;
                }
            }
        }  //203

        /*
store this pivot element -- (from left to right -- no danger of
overlap with the working elements in L (pivots).
*/
        ++len;
        w[len] = fact;
        jw[len] = jrow;
        goto F150;

    F160:
        // reset double-pointer to zero (U-part)
        for (k = 1; k <= lenu; ++k ) jw[n + jw[ii + k - 1]] = 0;  //308

        // update L-matrix
        lenl = len;
        len = MIN(lenl, lfil);

        // sort by quick-split
        fasp_qsplit(&w[1], &jw[1], lenl, len);

        // store L-part
        for (k = 1; k <= len; ++k )     {   //204
            if (ju0  >  iwk) goto F996;
            alu[ju0] = w[k];
            jlu[ju0] = jw[k];
            ++ju0;
        }

        // save pointer to beginning of row ii of U
        ju[ii] = ju0;

        // update U-matrix -- first apply dropping strategy
        len = 0;
        for (k = 1; k <= lenu - 1; ++k) {
            //      if ( ABS(w[ii + k])  >  droptol*tnorm )
            if ( ABS(w[ii + k])  >  tnorm[ii] ) {
                ++len;
                w[ii + len] = w[ii + k];
                jw[ii + len] = jw[ii + k];
            }
        }

        lenu = len + 1;
        len = MIN(lenu, lfil);

        fasp_qsplit(&w[ii + 1], &jw[ii + 1], lenu - 1, len);

        // copy
        t = ABS(w[ii]);
        if (len + ju0  >  iwk) goto F997;
        for (k = ii + 1; k<=ii + len - 1; ++k)  {  //302
            jlu[ju0] = jw[k];
            alu[ju0] = w[k];
            t = t + ABS(w[k]);
            ++ju0;
        }

        // store inverse of diagonal element of u
        // if (w(ii) .eq.  0.0) w(ii) = (0.0001 + droptol)*tnorm
        if (w[ii]  ==  0.0) w[ii] = tnorm[ii];

        alu[ii] = 1.0/w[ii];

        // update pointer to beginning of next row of U.
        jlu[ii + 1] = ju0;
        /*----------------------------------------------------------------------
end main loop
----------------------------------------------------------------------- */
    }

    *nz = ju[n] - 1;

    if (cinindex) {
        for(i = 1; i <= *nz; ++i)  --jlu[i];
    }

    *ierr = 0;

F100:
    ++jw;
    ++ju;
    ++w;
    ++tnorm;
    ++jlu;
    ++alu;
    ++ia;
    ++ja;
    ++a;

    fasp_mem_free(ju);      ju    = NULL;
    fasp_mem_free(jw);      jw    = NULL;
    fasp_mem_free(w);       w     = NULL;
    fasp_mem_free(tnorm);   tnorm = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;

F995:         // incomprehensible error.  Matrix must be wrong.
    printf("### ERROR: Input matrix may be wrong.  [%s]\n", __FUNCTION__);
    *ierr = -1;
    goto F100;

F996:         // insufficient storage in L.
    printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
    *ierr = -2;
    goto F100;

F997:         // insufficient storage in U.
    printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
    *ierr = -3;
    goto F100;

F998:         // illegal lfil entered.
    *ierr = -4;
    printf("### ERROR: Illegal lfil entered.  [%s]\n", __FUNCTION__);
    return;
    /*----------------end-of-ilut--------------------------------------------
-----------------------------------------------------------------------*/
}

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)
{
#if DEBUG_MODE > 0
```

```
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /*----------------------------------------------------------------------*
*** ILUTP preconditioner -- ILUT with pivoting  ***               *
incomplete LU factorization with dual truncation mechanism       *
----------------------------------------------------------------------*
author Yousef Saad *Sep 8, 1993 -- Latest revision, August 1996.  *
----------------------------------------------------------------------*
on entry:
==========
n       = integer.  The dimension of the matrix A.

a,ja,ia = matrix stored in Compressed Sparse Row format.
ON RETURN THE COLUMNS OF A ARE PERMUTED. SEE BELOW FOR
DETAILS.

lfil    = 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.
** WARNING: THE MEANING OF LFIL HAS CHANGED WITH RESPECT TO
EARLIER VERSIONS.

droptol = real*8.  Sets the threshold for dropping small terms in the
factorization.  See below for details on dropping strategy.

lfil    = integer.  The fill-in parameter.  Each row of L and
each row of U will have a maximum of lfil elements.

permtol = tolerance ratio used to  determne whether or not to permute
two columns.    At step i columns i and j are permuted when

abs(a(i,j))*permtol .gt.  abs(a(i,i))

[0 --> never permute; good values 0.1 to 0.01]

mbloc   = 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.

iwk     = 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.

On return:
===========

alu,jlu = 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.

ju      = integer array of length n containing the pointers to
the beginning of each row of U in the matrix alu,jlu.

iperm   = contains the permutation arrays.
iperm(1:n) = old numbers of unknowns
iperm(n+1:2*n) = reverse permutation = new unknowns.

ierr    = integer.  Error message with the following meaning.
ierr   = 0    --> successful return.
ierr .gt.  0  --> zero pivot encountered at step number ierr.
ierr   = -1   --> Error.  input matrix may be wrong.
(The elimination process has generated a
row in L or U whose length is .gt.    n.)
ierr   = -2   --> The matrix L overflows the array al.
ierr   = -3   --> The matrix U overflows the array alu.
ierr   = -4   --> Illegal value for lfil.
ierr   = -5   --> zero row encountered.

work arrays:
=============
jw      = integer work array of length 2*n.
w       = real work array of length n

IMPORTANR NOTE:
--------------
TO AVOID PERMUTING THE SOLUTION VECTORS ARRAYS FOR EACH LU-SOLVE,
THE MATRIX A IS PERMUTED ON RETURN. [all column indices are
changed].  SIMILARLY FOR THE U MATRIX.
```

```
To permute the matrix back to its original state use the loop:

do k=ia(1), ia(n+1)-1
ja(k) = iperm(ja(k))
enddo

-----------------------------------------------------------------------*/

    // local variables
    INT k, i, j, jrow, ju0, ii, j1, j2, jpos, len, imax, lenu, lenl, jj, icut,NE;
    REAL s, tmp, tnorm, xmax, xmax0, fact, t;
    SHORT cinindex=0;
    REAL  *w;
    INT  *ju, *jw;

    if (lfil  <  0) goto F998;

    ju = (INT *) fasp_mem_calloc(n, sizeof(INT));
    jw = (INT *) fasp_mem_calloc(2*n, sizeof(INT));
    w  = (REAL *)fasp_mem_calloc(n+1, sizeof(REAL));

    --ju;
    --jw;
    --iperm;
    --w;
    --jlu;
    --alu;
    --ia;
    --ja;
    --a;

    /*-----------------------------------------------------------------------
shift index for C routines
-----------------------------------------------------------------------*/
    if (ia[1]  ==  0) cinindex=1 ;

    if (cinindex)
    {
        NE = n + 1; //modify by chunsheng 2012, Sep, 1;
        for (i=1; i<=NE; ++i)  ++ia[i];
        NE = ia[n+1] - 1;
        for (i=1; i<=NE; ++i)  ++ja[i];
    }

    /*-----------------------------------------------------------------------
initialize ju0 (points to next element to be added to alu,jlu)
and pointer array.
-----------------------------------------------------------------------*/
    ju0 = n + 2;
    jlu[1] = ju0;


    //   integer double pointer array.
    for ( j = 1; j <= n; ++j ) { //1
        jw[n + j] = 0;
        iperm[j] = j;
        iperm[n + j] = j;
    } //1

    /*-----------------------------------------------------------------------
beginning of main loop.
-----------------------------------------------------------------------*/
    for (ii = 1; ii <= n; ++ii )    { //500
        j1 = ia[ii];
        j2 = ia[ii + 1] - 1;

        tnorm = 0.0;
        for (k = j1; k <= j2; ++k )     tnorm = tnorm + ABS( a[k] ); //501
        if (tnorm  ==  0.0) goto F999;
        tnorm = tnorm/(REAL)(j2 - j1 + 1);

        // unpack L-part and U-part of row of A in arrays  w  --
        lenu = 1;
        lenl = 0;
        jw[ii] = ii;
        w[ii] = 0.0;
        jw[n + ii] = ii;
        //
        for (j = j1; j <= j2; ++j )  { // 170
            k = iperm[n + ja[j]];
            t = a[j];
            if (k  <  ii) {
                ++lenl;
                jw[lenl] = k;
                w[lenl] = t;
                jw[n + k] = lenl;
            } else if (k  ==  ii) {
                w[ii] = t;
            } else {
                ++lenu;
                jpos = ii + lenu - 1;
                jw[jpos] = k;
                w[jpos] = t;
                jw[n + k] = jpos;
            }
        }  //170

        jj = 0;
        len = 0;


        // eliminate previous rows
    F150:
        ++jj;
        if (jj  >  lenl) goto F160;

        /*-----------------------------------------------------------------------
in order to do the elimination in the correct order we must select
the smallest column index among jw(k), k=jj+1, ..., lenl.
-----------------------------------------------------------------------*/
        jrow = jw[jj];
        k = jj;

        // determine smallest column index
        for (j = jj + 1; j <= lenl; ++j) {  //151
            if (jw[j]  <  jrow) {
                jrow = jw[j];
                k = j;
            }
        }

        if (k  !=  jj)  {
            // exchange in jw
            j = jw[jj];
            jw[jj] = jw[k];
            jw[k] = j;
            // exchange in jr
            jw[n + jrow] = jj;
            jw[n + j] = k;
            // exchange in w
            s = w[jj];
            w[jj] = w[k];
            w[k] = s;
        }

        // zero out element in row by resetting jw(n+jrow) to zero.
        jw[n + jrow] = 0;

        // get the multiplier for row to be eliminated:  jrow
        fact = w[jj]*alu[jrow];

        // drop term if small
        if (ABS(fact)  <=  droptol) goto F150;

        // combine current row and row jrow

        for ( k = ju[jrow]; k <= jlu[jrow + 1] - 1; ++k ) {  //203
            s = fact*alu[k];
            // new column number
            j = iperm[n + jlu[k]];
            jpos = jw[n + j];
            if (j  >=  ii) {
                // dealing with upper part.
                if (jpos  ==  0) {
                    //   this is a fill-in element
                    ++lenu;
                    i = ii + lenu - 1;
                    if (lenu  >  n) goto F995;
                    jw[i] = j;
                    jw[n + j] = i;
                    w[i] = -s;
                } else {
                    //     no fill-in element --
                    w[jpos] = w[jpos] - s;
                }

            } else {
                // dealing with lower part.
                if (jpos  ==  0) {
                    //   this is a fill-in element
                    ++lenl;
                    if (lenl  >  n) goto F995;
                    jw[lenl] = j;
                    jw[n + j] = lenl;
                    w[lenl] = -s;
                } else {
                    //   this is not a fill-in element
                    w[jpos] = w[jpos] - s;
                }
            }
        }  //203

        /*
store this pivot element -- (from left to right -- no danger of
overlap with the working elements in L (pivots).
*/

        ++len;
        w[len] = fact;
        jw[len] = jrow;
        goto F150;

    F160:
        // reset double-pointer to zero (U-part)
        for ( k = 1; k <= lenu; ++k ) jw[n + jw[ii + k - 1]] = 0;  //308

        // update L-matrix
        lenl = len;
        len = MIN(lenl, lfil);

        // sort by quick-split
        fasp_qsplit(&w[1], &jw[1], lenl, len);

        // store L-part -- in original coordinates ..
        for ( k = 1; k <= len; ++k ) {  // 204
            if (ju0  >  iwk) goto F996;
            alu[ju0] = w[k];
            jlu[ju0] = iperm[jw[k]];
            ++ju0;
        }  //204

        // save pointer to beginning of row ii of U
        ju[ii] = ju0;

        // update U-matrix -- first apply dropping strategy
        len = 0;
        for(k = 1; k <= lenu - 1; ++k ) {
            if ( ABS(w[ii + k])  >  droptol*tnorm) {
                ++len;
                w[ii + len] = w[ii + k];
                jw[ii + len] = jw[ii + k];
            }
        }

        lenu = len + 1;
        len = MIN(lenu, lfil);
        fasp_qsplit(&w[ii + 1], &jw[ii + 1], lenu-1, len);

        // determine next pivot --
        imax = ii;
        xmax = ABS(w[imax]);
        xmax0 = xmax;
        icut = ii - 1 + mbloc - (ii - 1)%mbloc;

        for ( k = ii + 1; k <= ii + len - 1; ++k ) {
            t = ABS(w[k]);
            if ((t  >  xmax) && (t*permtol  >  xmax0) && (jw[k]  <=  icut)) {
                imax = k;
                xmax = t;
            }
        }

        // exchange w's
        tmp = w[ii];
        w[ii] = w[imax];
        w[imax] = tmp;

        // update iperm and reverse iperm
        j = jw[imax];
        i = iperm[ii];
        iperm[ii] = iperm[j];
        iperm[j] = i;

        // reverse iperm
        iperm[n + iperm[ii]] = ii;
        iperm[n + iperm[j]] = j;

        //-----------------------------------------------------------------------
        if (len + ju0  >  iwk) goto F997;


        // copy U-part in original coordinates
        for ( k = ii + 1; k <= ii + len - 1; ++k ) { //302
            jlu[ju0] = iperm[jw[k]];
            alu[ju0] = w[k];
            ++ju0;
        }

        // store inverse of diagonal element of u
        if (w[ii]  ==  0.0) w[ii] = (1.0e-4 + droptol)*tnorm;
        alu[ii] = 1.0/w[ii];

        // update pointer to beginning of next row of U.
        jlu[ii + 1] = ju0;

        /*-----------------------------------------------------------------------
end main loop
-----------------------------------------------------------------------*/
    }  //500

    // permute all column indices of LU ...
    for ( k = jlu[1]; k <= jlu[n + 1] - 1; ++k )    jlu[k] = iperm[n + jlu[k]];

    // ...and of A
    for ( k = ia[1]; k <= ia[n + 1] - 1; ++k )  ja[k] = iperm[n + ja[k]];

    *nz = ju[n]- 1;

    if (cinindex)  {
        for (i = 1; i <= *nz; ++i ) --jlu[i];
    }

    *ierr = 0;

F100:
    ++jw;
    ++ju;
    ++iperm;
    ++w;
    ++jlu;
    ++alu;
    ++ia;
    ++ja;
    ++a;

    fasp_mem_free(ju);  ju = NULL;
    fasp_mem_free(jw);  jw = NULL;
    fasp_mem_free(w);   w  = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;

F995:        // incomprehensible error.  Matrix must be wrong.
    printf("### ERROR: Input matrix may be wrong.  [%s]\n", __FUNCTION__);
    *ierr = -1;
    goto F100;

F996:        // insufficient storage in L.
    printf("### ERROR: Insufficient storage in L. [%s]\n", __FUNCTION__);
    *ierr = -2;
    goto F100;
```

```

F997:         // insufficient storage in U.
    printf("### ERROR: Insufficient storage in U. [%s]\n", __FUNCTION__);
    *ierr = -3;
    goto F100;

F998:         // illegal lfil entered.
    printf("### ERROR: Illegal lfil entered.  [%s]\n", __FUNCTION__);
    *ierr = -4;
    // goto F100;
    return;

F999:         // zero row encountered
    printf("### ERROR: Zero row encountered.  [%s]\n", __FUNCTION__);
    *ierr = -5;
    goto F100;
    //----------------end-of-ilutp------------------------------------------
}

void fasp_symbfactor (INT   n,
                      INT  *colind,
                      INT  *rwptr,
                      INT   levfill,
                      INT   nzmax,
                      INT  *nzlu,
                      INT  *ijlu,
                      INT  *uptr,
                      INT  *ierr)
{
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    INT icolindj, ijlum, i, j, k, m, ibegin, iend, Ujbeg, Ujend,NE;
    INT head, prev, lm, actlev, lowct, k1, k2, levp1, lmk, nzi, rowct;
    SHORT cinindex=0;
    INT *rowll, *lastcol, *levels;

    rowll = (INT *)fasp_mem_calloc(n, sizeof(INT));
    lastcol = (INT *)fasp_mem_calloc(n, sizeof(INT));
    levels = (INT *)fasp_mem_calloc(nzmax, sizeof(INT));

    //========================================================================
    //       Beginning of Executable Statements
    //========================================================================

    /*----------------------------------------------------------------------
shift index for C routines
-----------------------------------------------------------------------*/
    --rowll;
    --lastcol;
    --levels;
    --colind;
    --rwptr;
    --ijlu;
    --uptr;

    if (rwptr[1]  ==  0) cinindex=1 ;
    if (cinindex) {
        NE = n + 1;
        for (i=1; i<=NE; ++i)  ++rwptr[i];
        NE = rwptr[n+1] - 1;
        for (i=1; i<=NE; ++i)  ++colind[i];
    }

    // -------------------------------------------------------------
    // Because the first row of the factor contains no strictly lower
    // triangular parts (parts of L), uptr(1) = ijlu(1) = n+2:
    // -------------------------------------------------------------
    ijlu[1] = n + 2;
    uptr[1] = n + 2;

    // --------------------------------------------------------
    // The storage for the nonzeros of LU must be at least n+1,
    // for a diagonal matrix:
    // --------------------------------------------------------
    *nzlu = n + 1;

    //  ------------------------------------------------------------------
    //  Number of allowed levels plus 1; used for the test of accept/reject.
    //  See the notes about the methodology above.
    //  ---------------------------------------------------------------------
    levp1 = levfill + 1;

    //  ------------------------------------------------------------
    //  Initially, for all columns there were no nonzeros in the rows
    //  above, because there are no rows above the first one.
    //  ------------------------------------------------------------
    for (i = 1; i<=n; ++i) lastcol[i] = 0;

    //  -------------------
    //  Proceed row by row:
    //  -------------------
    for (i = 1; i <= n; ++i) { // 100

        // ----------------------------------------------------------
        // Because the matrix diagonal entry is nonzero, the level of
        // fill for that diagonal entry is zero:
        // ----------------------------------------------------------
        levels[i] = 0;

        // ----------------------------------------------------------
        // ibegin and iend are the beginning of rows i and i+1, resp.
        // ----------------------------------------------------------
        ibegin = rwptr[i];
        iend = rwptr[i + 1];

        //  ------------------------------------------------------------
        //  Number of offdiagonal nonzeros in the original matrix's row i
        //  ------------------------------------------------------------
        nzi = iend - ibegin;

        //  --------------------------------------------------------
        //  If only the diagonal entry in row i is nonzero, skip the
        //  fancy stuff; nothing need be done:
        //  --------------------------------------------------------
        if (nzi  >  1) {
            //  ----------------------------------------------------------
            //  Decrement iend, so that it can be used as the ending index
            //  in icolind of row i:
            //  ----------------------------------------------------------
            iend = iend - 1;

            //  ---------------------------------------------------------
            //  rowct keeps count of the number of nondiagonal entries in
            //  the current row:
            //  ---------------------------------------------------------
            rowct = 0;

            //  ------------------------------------------------------------
            //  For nonzeros in the current row from the original matrix A,
            //  set lastcol to be the current row number, and the levels of
            //  the entry to be 1.    Note that this is really the true level
            //  of the element, plus 1.    At the same time, load up the work
            //  array rowll with the column numbers for the original entries
            //  from row i:
            //  ------------------------------------------------------------
#if DEBUG_MODE > 0
            printf("### DEBUG: %s %d row\n", __FUNCTION__, i);
#endif

            for ( j = ibegin; j <= iend; ++j) {
                icolindj = colind[j];
                lastcol[icolindj] = i;
                if (icolindj !=  i) {
                    levels[icolindj] = 1;
                    rowct = rowct + 1;
                    rowll[rowct] = icolindj;
                }
#if DEBUG_MODE > 0
                printf("### DEBUG: %d\n", icolindj);
#endif
            }

            //  ---------------------------------------------------------
            //  Sort the entries in rowll, so that the row has its column
            //  entries in increasing order.
            //  ---------------------------------------------------------
            fasp_sortrow(nzi - 1, &rowll[1]);

            //check col index
            fasp_check_col_index(i, nzi-1, &rowll[1]);
            //  ---------------------------------------------------------
            //  Now set up rowll as a linked list containing the original
            //  nonzero column numbers, as described in the methods section:
            //  ---------------------------------------------------------
            head = rowll[1];
            k1 = n + 1;
            for (j = nzi - 1; j >= 1;  --j) {
                k2 = rowll[j];
                rowll[k2] = k1;
                k1 = k2;
            }

            //  ------------------------------------------------------------
            //  Increment count of nonzeros in the LU factors by the number
            //  of nonzeros in the original matrix's row i.    Further
            //  incrementing will be necessary if any fill-in actually occurs
            //  ------------------------------------------------------------
            *nzlu = *nzlu + nzi - 1;

            //   ------------------------------------------------------------
            //   The integer j will be used as a pointer to track through the
            //   linked list rowll:
            //   ------------------------------------------------------------
            j = head;

            //   ------------------------------------------------------------
            //   The integer lowct is used to keep count of the number of
            //   nonzeros in the current row's strictly lower triangular part,
            //   for setting uptr pointers to indicate where in ijlu the upperc
            //   triangular part starts.
            //   ------------------------------------------------------------
            lowct = 0;

            //   ------------------------------------------------------------
            //   Fill-in could only have resulted from rows preceding row i,
            //   so we only need check those rows with index j < i.
            //   Furthermore, if the current row has a zero in column j,
            //   there is no need to check the preceding rows; there clearly
            //   could not be any fill-in from those rows to this entry.
            //   ------------------------------------------------------------
            while (j  <  i) {  //80
                //  ------------------------------------------------------------
                //  Increment lower triangular part count, since in this case
                //  (j<i) we got another entry in L:
                //  ------------------------------------------------------------
                lowct = lowct + 1;

                //   ---------------------------------------------------------
                //   If the fill level is zero, there is no way to get fill in
                //   occuring.
                //   ---------------------------------------------------------
                if (levfill !=  0) {

                    //   -----------------------------------------------------
                    //   Ujbeg is beginning index of strictly upper triangular
                    //   part of U's j-th row, and Ujend is the ending index
                    //   of it, in ijlu().
                    //   -----------------------------------------------------
                    Ujbeg = uptr[j];
                    Ujend = ijlu[j + 1] - 1;

                    //   -----------------------------------------------------
                    //   Need to set pointer to previous entry before working
                    //   segment of rowll, because if fill occurs that will be
                    //   a moving segment.
                    //   -----------------------------------------------------
                    prev = j;

                    //  -----------------------------------------------------
                    //  lm is the next nonzero pointer in linked list rowll:
                    //  -----------------------------------------------------
                    lm = rowll[j];

                    //  -------------------------------------------------------
                    //  lmk is the fill level in this row, caused by
                    //  eliminating column entry j.    That is, level s1 from the
                    //  methodology explanation above.
                    //  -------------------------------------------------------
                    lmk = levels[j];

                    //  -------------------------------------------------------
```

```
                        //  Now proceed through the j-th row of U, because in the
                        //  elimination we add a multiple of it to row i to zero
                        //  out entry (i,j).    If a column entry in row j of U is
                        //  zero, there is no need to worry about fill, because it
                        //  cannot cause a fill in the corresponding entry of row i
                        //  -------------------------------------------------------
                        for (m = Ujbeg; m <= Ujend; ++m) { //60
                            //  --------------------------------------------------
                            //  ijlum is the column number of the current nonzero in
                            //  row j of U:
                            //  --------------------------------------------------
                            ijlum = ijlu[m];

                            //  -------------------------------------------------
                            //  actlev is the actual level (plus 1) of column entry
                            //  j in row i, from summing the level contributions
                            //  s1 and s2 as explained in the methods section.
                            //  Note that the next line could reasonably be
                            //  replaced by, e.g., actlev = max(lmk, levels(m)),
                            //  but this would cause greater fill-in:
                            //  -------------------------------------------------
                            actlev = lmk + levels[m];

                            //  -------------------------------------------------
                            //  If lastcol of the current column entry in U is not
                            //  equal to the current row number i, then the current
                            //  row has a zero in column j, and the earlier row j
                            //  in U has a nonzero, so possible fill can occur.
                            //  -------------------------------------------------
                            if (lastcol[ijlum]  !=  i) {

                                //  -------------------------------------------------
                                //  If actlev < levfill + 1, then the new entry has an
                                //  acceptable fill level and needs to be added to the
                                //  data structure.
                                //  -------------------------------------------------
                                if (actlev  <=  levp1) {

                                    //  ------------------------------------------
                                    //  Since the column entry ijlum in the current
                                    //  row i is to be filled, we need to update
                                    //  lastcol for that column number.    Also, the
                                    //  level number of the current entry needs to be
                                    //  set to actlev.    Note that when we finish
                                    //  processing this row, the n-vector levels(1:n)
                                    //  will be copied over to the corresponding
                                    //  trailing part of levels, so that it can be
                                    //  used in subsequent rows:
                                    //  ------------------------------------------
                                    lastcol[ijlum] = i;
                                    levels[ijlum] = actlev;

                                    //  ------------------------------------------
                                    //  Now find location in the linked list rowll
                                    //  where the fillin entry should be placed.
                                    //  Chase through the linked list until the next
                                    //  nonzero column is to the right of the fill
                                    //  column number.
                                    //  ------------------------------------------
                                    while (lm  <=  ijlum) { //50
                                        prev = lm;
                                        lm = rowll[lm];
                                    }  //50

                                    //  ------------------------------------------
                                    //  Insert new entry into the linked list for
                                    //  row i, and increase the nonzero count for LU
                                    //  ------------------------------------------
                                    rowll[prev] = ijlum;
                                    rowll[ijlum] = lm;
                                    prev = ijlum;
                                    *nzlu = *nzlu + 1;
                                }

                                //  -------------------------------------------------
                                //  Else clause is for when lastcol(ijlum) = i.    In
                                //  this case, the current column has a nonzero, but
                                //  it resulted from an earlier fill-in or from an
                                //  original matrix entry.    In this case, need to
                                //  update the level number for this column to be the
                                //  smaller of the two possible fill contributors,
                                //  the current fill number or the computed one from
                                //  updating this entry from a previous row.
                                //  -------------------------------------------------
                            } else  {
                                levels[ijlum] = MIN(levels[ijlum], actlev);
                            }

                            //  -------------------------------------------------
                            //  Now go and pick up the next column entry from row
                            //  j of U:
                            //  -------------------------------------------------

                        } //60
                        // -------------------------------------------
                        // End if clause for levfill not equal to zero
                        // -------------------------------------------
                    }

                    // ------------------------------------------------------
                    // Pick up next nonzero column index from the linked
                    // list, and continue processing the i-th row's nonzeros.
                    // This ends the first while loop (j < i).
                    // ------------------------------------------------------
                    j = rowll[j];
                }  //80

                //  -----------------------------------------------------------
                //  Check to see if we have exceeded the allowed memory
                //  storage before storing the results of computing row i's
                //  sparsity pattern into the ijlu and uptr data structures.
                //  -----------------------------------------------------------
                if (*nzlu  >  nzmax) {
                    printf("### ERROR: More storage needed!  [%s]\n", __FUNCTION__);
                    *ierr = 1;
                    goto F100;
                }

                // ----------------------------------------------------------
                // Storage is adequate, so update ijlu data structure.
                // Row i ends at nzlu + 1:
                // ----------------------------------------------------------
                ijlu[i + 1] = *nzlu + 1;

                //  ----------------------------------------------------------
                //  ...  and the upper triangular part of LU begins at
                //  lowct entries to right of where row i begins.
                //  ----------------------------------------------------------
                uptr[i] = ijlu[i] + lowct;

                //  -----------------------------------------------------
                //  Now chase through linked list for row i, recording
                //  information into ijlu.   At same time, put level data
                //  into the levels array for use on later rows:
                //  -----------------------------------------------------
                j = head;
                k1 = ijlu[i];
                for (k = k1; k <= *nzlu; ++k) {
                    ijlu[k] = j;
                    levels[k] = levels[j];
                    j = rowll[j];
                }

            } else  {

                // ---------------------------------------------------------
                // This else clause ends the (nzi > 1) if.    If nzi = 1, then
                // the update of ijlu and uptr is trivial:
                // ---------------------------------------------------------
                ijlu[i + 1] = *nzlu + 1;
                uptr[i] = ijlu[i];
            }

            // ----------------------------------------------
            // And you thought we would never get through....
            // ----------------------------------------------
        }  //100

        if (cinindex) {
            for ( i = 1; i <= *nzlu; ++i ) --ijlu[i];
            for ( i = 1; i <= n; ++i )     --uptr[i];
            NE = rwptr[n + 1] - 1;
            for ( i = 1; i <= NE; ++i )     --colind[i];
            NE = n + 1;
            for ( i = 1; i <= NE; ++i )     --rwptr[i];
        }

        *ierr = 0;

    F100:
        ++rowll;
        ++lastcol;
        ++levels;
        ++colind;
        ++rwptr;
        ++ijlu;
        ++uptr;

        fasp_mem_free(rowll);    rowll   = NULL;
        fasp_mem_free(lastcol);  lastcol = NULL;
        fasp_mem_free(levels);   levels  = NULL;

    #if DEBUG_MODE > 0
        printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
    #endif

        return;
        //======================== End of symbfac ==============================
    }

    /*----------------------------------*/
    /*--      Private Functions       --*/
    /*----------------------------------*/

    static void fasp_qsplit (REAL   *a,
                             INT    *ind,
                             INT     n,
                             INT     ncut)
    {
        /*-----------------------------------------------------------------------
    does a quick-sort split of a real array.
    on input a(1:n).  is a real array
    on output a(1:n) is permuted such that its elements satisfy:
    abs(a(i)) .ge.  abs(a(ncut)) for i .lt.  ncut and
    abs(a(i)) .le.  abs(a(ncut)) for i .gt.  ncut
    ind(1:n) is an integer array which permuted in the same way as a(*).
    ----------------------------------------------------------------------*/
        REAL tmp, abskey;
        INT  itmp, first, last, mid, j;

        /* Parameter adjustments */
        --ind;
        --a;

        first = 1;
        last = n;
        if ((ncut < first) || (ncut > last)) return;

        // outer loop -- while mid .ne.  ncut do
    F161:
        mid = first;
        abskey = ABS(a[mid]);
        for (j = first + 1; j <= last; ++j ) {
            if (ABS(a[j])  >  abskey) {
                ++mid;
                //     interchange
                tmp = a[mid];
                itmp = ind[mid];
                a[mid] = a[j];
                ind[mid] = ind[j];
                a[j] = tmp;
                ind[j] = itmp;
            }
        }

        // interchange
        tmp = a[mid];
        a[mid] = a[first];
        a[first] = tmp;
        //
        itmp = ind[mid];
        ind[mid] = ind[first];
        ind[first] = itmp;

        // test for while loop
        if (mid  ==  ncut) {
            ++ind;
            ++a;
            return;
        }

        if (mid  >  ncut)  {
            last = mid - 1;
        } else    {
            first = mid + 1;
        }

        goto F161;
        /*----------------end-of-qsplit------------------------------------------*/
    }

    static void fasp_sortrow (INT   num,
                              INT   *q)
    {
    #if DEBUG_MODE > 0
        printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    #endif
        INT key, icn, ih, ii, i, j, jj;
        INT iinc[6] = {0,1, 4, 13, 40, 121};
        //data iinc/1, 4, 13, 40, 121/;

        --q;
        if (num  ==  0)
            icn = 0;
        else if (num  <  14)
            icn = 1;
        else if (num  <  41)
            icn = 2;
        else if (num  <  122)
            icn = 3;
        else if (num  <  365)
            icn = 4;
        else
            icn = 5;

        for(ii = 1; ii <= icn; ++ii) { // 40
            ih = iinc[icn + 1 - ii];
            for(j = ih + 1; j <= num; ++j) { // 30
                i = j - ih;
                key = q[j];
                for(jj = 1; jj <= j - ih; jj += ih) { // 10
                    if (key  >=  q[i]) {
                        goto F20;
                    } else {
                        q[i + ih] = q[i];
                        i = i - ih;
                    }
                }  // 10
            F20:
                q[i + ih] = key;
            }  // 30
        } // 40

        ++q;

    #if DEBUG_MODE > 0
        printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
    #endif
        return;
    }

    static void fasp_check_col_index (INT row,
                                      INT num,
                                      INT  *q)
    {
    #if DEBUG_MODE > 0
        printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    #endif

        INT ii;
        INT num_1 = num - 1;

        for ( ii = 0; ii < num_1; ++ii ) {
            if ( q[ii] == q[ii+1] ) {
                printf("### ERROR: Multiple entries with same col indices!\n");
                printf("### ERROR: row = %d, col = %d, %d!\n", row, q[ii], q[ii+1]);
                fasp_chkerr(ERROR_SOLVER_ILUSETUP, __FUNCTION__);
            }
        }

    #if DEBUG_MODE > 0
        printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
    #endif

        return;
    }

    /*---------------------------------*/
    /*--        End of File          --*/
    /*---------------------------------*/
```

9.53 BlaILUSetupBSR.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, BlaILU.c, BlaSmallMat.c, BlaSmallMatInv.c, BlaSparseBSR.c, BlaSparseCSR.c, BlaSpmvCSR.c, and PreDataInit.c

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

Definition in file BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param

Returns

FASP_SUCCESS if successed; 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 inited.

Definition at line 55 of file BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param
step	Step in ILU factorization

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
Ap	Pointer to dCSRmat matrix which provides sparsity pattern
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupBSR.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

A	Pointer to dBSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param
step	Step in ILU factorization

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupBSR.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

mgl	Pointer to input matrix
gslvl	Used to specify levels of AMG using multicolor smoothing

Author

Zheng Li, Chunsheng Feng

Date

12/04/2016

Definition at line 1909 of file BlaILUSetupBSR.c.

9.53.2.8 topologic_sort_ILU()

```
void topologic_sort_ILU (
            ILU_data * iludata )
```

Reordering vertices according to level schedule strategy.

Parameters

iludata	Pointer to iludata

Author

Zheng Li, Chensong Zhang

Date

12/04/2016

Definition at line 1827 of file BlaILUSetupBSR.c.

9.54 BlaILUSetupBSR.c

Go to the documentation of this file.

```

#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static INT numfactor (dBSRmat *, REAL *, INT *, INT *);
static INT numfactor_mulcol (dBSRmat *, REAL *, INT *, INT *, INT, INT *, INT *);
static INT numfactor_levsch (dBSRmat *, REAL *, INT *, INT *, INT, INT *, INT *);
static void generate_S_theta(dCSRmat *, iCSRmat *, REAL);
// static void topologic_sort_ILU (ILU_data *);
// static void mulcol_independ_set (AMG_data *, INT);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_ilu_dbsr_setup(dBSRmat *A,
                          ILU_data *iludata,
                          ILU_param *iluparam)
{

    const SHORT  prtlvl = iluparam->print_level;
    const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;

    // local variables
    INT     lfil = iluparam->ILU_lfil;
    INT     ierr, iwk, nzlu, nwork, *ijlu, *uptr;
    SHORT   status = FASP_SUCCESS;
    REAL    setup_start, setup_end, setup_duration;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
#endif

    fasp_gettime(&setup_start);

    // Expected amount of memory for ILU needed and allocate memory
    iwk = (lfil+2)*nnz;

#if DEBUG_MODE > 0
    if (iluparam->ILU_type == ILUtp) {
        printf("### WARNING: iludata->type = %d not supported!\n",
               iluparam->ILU_type);
    }
#endif

    // setup preconditioner
    iludata->type  = 0; // Must be initialized
    iludata->iperm = NULL;
    iludata->A     = NULL; // No need for BSR matrix
    iludata->row   = iludata->col = n;
    iludata->nb    = nb;
    iludata->ilevL = iludata->jlevL = NULL;
    iludata->ilevU = iludata->jlevU = NULL;

    ijlu = (INT*)fasp_mem_calloc(iwk,sizeof(INT));
    uptr = (INT*)fasp_mem_calloc(A->ROW,sizeof(INT));

#if DEBUG_MODE > 1
    printf("### DEBUG: symbolic factorization ...  \n");
#endif

    // ILU decomposition
    // (1) symbolic factoration
    fasp_symbfactor(A->ROW,A->JA,A->IA,lfil,iwk,&nzlu,ijlu,uptr,&ierr);

    if ( ierr != 0 ) {
        printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));

#if DEBUG_MODE > 1
    printf("### DEBUG: numerical factorization ...  \n");
#endif

    // (2) numerical factoration
    status = numfactor(A, iludata->luval, ijlu, uptr);

    if ( status < 0 ) {
        printf("### ERROR: ILU factorization failed!  [%s]\n", __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    //nwork = 6*nzlu*nb;
    nwork = 20*A->ROW*A->nb;
    iludata->nzlu  = nzlu;
    iludata->nwork = nwork;
    iludata->ijlu  = (INT*)fasp_mem_calloc(nzlu, sizeof(INT));

    memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
    iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
    // Check:  Is the work space too large?  --Xiaozhe

#if DEBUG_MODE > 1
    printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
    printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif

    if ( iwk < nzlu ) {
        printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;
        printf("BSR ILU(%d)-seq setup costs %f seconds.\n", lfil, setup_duration);
    }

FINISHED:
    fasp_mem_free(ijlu);  ijlu = NULL;
    fasp_mem_free(uptr);  uptr = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

SHORT fasp_ilu_dbsr_setup_step (dBSRmat    *A,
                                ILU_data   *iludata,
                                ILU_param  *iluparam,
                                INT step)
{

    const SHORT  prtlvl = iluparam->print_level;
    const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;

    // local variables
       INT     lfil = iluparam->ILU_lfil;
    static INT     ierr, iwk, nzlu, nwork, *ijlu, *uptr;
    SHORT   status = FASP_SUCCESS;

    REAL    setup_start, setup_end, setup_duration;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
#endif

    fasp_gettime(&setup_start);

    if (step==1) {
        // Expected amount of memory for ILU needed and allocate memory
        iwk = (lfil+2)*nnz;

#if DEBUG_MODE > 0
        if (iluparam->ILU_type == ILUtp) {
            printf("### WARNING: iludata->type = %d not supported!\n",
                iluparam->ILU_type);
        }
#endif

        // setup preconditioner
        iludata->type  = 0; // Must be initialized
        iludata->iperm = NULL;
        iludata->A     = NULL; // No need for BSR matrix
        iludata->row   = iludata->col = n;
        iludata->nb    = nb;
        iludata->ilevL = iludata->jlevL = NULL;
        iludata->ilevU = iludata->jlevU = NULL;

        ijlu = (INT*)fasp_mem_calloc(iwk,sizeof(INT));

        if (uptr != NULL)   fasp_mem_free(uptr);
        uptr = (INT*)fasp_mem_calloc(A->ROW,sizeof(INT));

#if DEBUG_MODE > 1
        printf("### DEBUG: symbolic factorization ...  \n");
#endif

        // ILU decomposition
        // (1) symbolic factoration
        fasp_symbfactor(A->ROW,A->JA,A->IA,lfil,iwk,&nzlu,ijlu,uptr,&ierr);

        iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));


#if DEBUG_MODE > 1
        printf("### DEBUG: numerical factorization ...  \n");
#endif

        //nwork = 6*nzlu*nb;
        nwork = 5*A->ROW*A->nb;
        iludata->nwork = nwork;
        iludata->nzlu  = nzlu;
        iludata->ijlu  = (INT*)fasp_mem_calloc(nzlu, sizeof(INT));

        memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
        fasp_mem_free(ijlu);  ijlu = NULL;

        iludata->work = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
        // Check:  Is the work space too large?  --Xiaozhe

#if DEBUG_MODE > 1
        printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
        printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif

        if ( ierr != 0 ) {
            printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
            status = ERROR_SOLVER_ILUSETUP;
            goto FINISHED;
        }

        if ( iwk < nzlu ) {
            printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
            status = ERROR_SOLVER_ILUSETUP;
            goto FINISHED;
        }
    }
    else if (step==2) {
        // (2) numerical factoration
        numfactor(A, iludata->luval, iludata->ijlu, uptr);

    } else {

FINISHED:
            fasp_mem_free(uptr);  uptr = NULL;
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;
        printf("BSR ILU(%d) setup costs %f seconds.\n", lfil, setup_duration);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

SHORT fasp_ilu_dbsr_setup_omp (dBSRmat    *A,
                               ILU_data   *iludata,
                               ILU_param  *iluparam)
{

    const SHORT  prtlvl = iluparam->print_level;
    const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;

    // local variables
    INT     lfil = iluparam->ILU_lfil;
    INT     ierr, iwk, nzlu, nwork, *ijlu, *uptr;
    SHORT   status = FASP_SUCCESS;

    REAL    setup_start, setup_end, setup_duration;
    REAL    symbolic_start, symbolic_end, numfac_start, numfac_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m = %d, n = %d, nnz = %d\n", A->ROW, n, nnz);
#endif

    fasp_gettime(&setup_start);

    // Expected amount of memory for ILU needed and allocate memory
    iwk = (lfil+2)*nnz;

#if DEBUG_MODE > 0
    if (iluparam->ILU_type == ILUtp) {
        printf("### WARNING: iludata->type = %d not supported any more!\n",
               iluparam->ILU_type);
    }
#endif

    // setup preconditioner
    iludata->type  = 0; // Must be initialized
    iludata->iperm = NULL;
    iludata->A     = NULL; // No need for BSR matrix
    iludata->row   = iludata->col = n;
    iludata->nb    = nb;

    ijlu = (INT *) fasp_mem_calloc(iwk,    sizeof(INT));
    uptr = (INT *) fasp_mem_calloc(A->ROW,sizeof(INT));

#if DEBUG_MODE > 1
    printf("### DEBUG: symbolic factorization ...  \n");
#endif

    // ILU decomposition
    // (1) symbolic factoration
    fasp_gettime(&symbolic_start);

    fasp_symbfactor(A->ROW,A->JA,A->IA,lfil,iwk,&nzlu,ijlu,uptr,&ierr);

    fasp_gettime(&symbolic_end);

#if prtlvl > PRINT_MIN
    printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
#endif

    nwork = 5*A->ROW*A->nb;
    iludata->nzlu  = nzlu;
    iludata->nwork = nwork;
    iludata->ijlu  = (INT*)fasp_mem_calloc(nzlu,sizeof(INT));
    iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));
    iludata->work  = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
    memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
    fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);

#if DEBUG_MODE > 1
    printf("### DEBUG: numerical factorization ...  \n");
#endif

    // (2) numerical factoration
    fasp_gettime(&numfac_start);

    numfactor_mulcol(A, iludata->luval, ijlu, uptr, iludata->nlevL,
                     iludata->ilevL, iludata->jlevL);

    fasp_gettime(&numfac_end);

#if prtlvl > PRINT_MIN
    printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
#endif

#if DEBUG_MODE > 1
    printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
    printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif

```

```
    if ( ierr != 0 ) {
        printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if ( iwk < nzlu ) {
        printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;
        printf("BSR ILU(%d)-mc setup costs %f seconds.\n", lfil, setup_duration);
    }

FINISHED:
    fasp_mem_free(ijlu);  ijlu = NULL;
    fasp_mem_free(uptr);  uptr = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

SHORT fasp_ilu_dbsr_setup_levsch_omp (dBSRmat    *A,
                                      ILU_data   *iludata,
                                      ILU_param  *iluparam)
{
    const SHORT  prtlvl = iluparam->print_level;
    const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;

    // local variables
    INT lfil = iluparam->ILU_lfil;
    INT ierr, iwk, nzlu, nwork, *ijlu, *uptr;
    SHORT   status = FASP_SUCCESS;

    REAL    setup_start, setup_end, setup_duration;
    REAL    symbolic_start, symbolic_end, numfac_start, numfac_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->ROW, n, nnz);
#endif

    fasp_gettime(&setup_start);

    // Expected amount of memory for ILU needed and allocate memory
    iwk = (lfil+2)*nnz;

#if DEBUG_MODE > 0
    if (iluparam->ILU_type == ILUtp) {
        printf("### WARNING: iludata->type = %d not supported!\n",
               iluparam->ILU_type);
    }
#endif

    // setup preconditioner
    iludata->type  = 0; // Must be initialized
    iludata->iperm = NULL;
    iludata->A     = NULL; // No need for BSR matrix
    iludata->row   = iludata->col=n;
    iludata->nb    = nb;

    ijlu = (INT*)fasp_mem_calloc(iwk,sizeof(INT));
    uptr = (INT*)fasp_mem_calloc(A->ROW,sizeof(INT));

#if DEBUG_MODE > 1
    printf("### DEBUG: symbolic factorization ...  \n");
#endif

    fasp_gettime(&symbolic_start);

    // ILU decomposition
    // (1) symbolic factoration
    fasp_symbfactor(A->ROW,A->JA,A->IA,lfil,iwk,&nzlu,ijlu,uptr,&ierr);

    fasp_gettime(&symbolic_end);

#if prtlvl > PRINT_MIN
    printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
#endif

    nwork = 5*A->ROW*A->nb;
    iludata->nzlu  = nzlu;
    iludata->nwork = nwork;
    iludata->ijlu  = (INT*)fasp_mem_calloc(nzlu,sizeof(INT));
    iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));
    iludata->work  = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
    memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
    fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);
    iludata->uptr = NULL; iludata->ic = NULL; iludata->icmap = NULL;

    topologic_sort_ILU(iludata);

#if DEBUG_MODE > 1
    printf("### DEBUG: numerical factorization ...  \n");
#endif

    fasp_gettime(&numfac_start);

    // (2) numerical factoration
    numfactor_levsch(A, iludata->luval, ijlu, uptr, iludata->nlevL,
                     iludata->ilevL, iludata->jlevL);

    fasp_gettime(&numfac_end);

#if prtlvl > PRINT_MIN
    printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
#endif

#if DEBUG_MODE > 1
    printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
    printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif

    if ( ierr != 0 ) {
        printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if ( iwk < nzlu ) {
        printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;
        printf("BSR ILU(%d)-ls setup costs %f seconds.\n", lfil, setup_duration);
    }

FINISHED:
    fasp_mem_free(ijlu);  ijlu = NULL;
    fasp_mem_free(uptr);  uptr = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

SHORT fasp_ilu_dbsr_setup_levsch_step (dBSRmat    *A,
                                       ILU_data   *iludata,
                                       ILU_param  *iluparam,
                                       INT step)
{
    const SHORT  prtlvl = iluparam->print_level;
    const INT    n = A->COL, nnz = A->NNZ, nb = A->nb, nb2 = nb*nb;

    // local variables
    INT lfil = iluparam->ILU_lfil;
    static INT ierr, iwk, nzlu, nwork, *ijlu, *uptr;
    SHORT   status = FASP_SUCCESS;

    REAL    setup_start, setup_end, setup_duration;
    REAL    symbolic_start, symbolic_end, numfac_start, numfac_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->ROW, n, nnz);
    printf("### DEBUG: step=%d(1:  symbolic factoration, 2:  numerical factoration)\n", step);// zhaoli
2021.03.24
#endif

    fasp_gettime(&setup_start);
   if (step==1) {
        // Expected amount of memory for ILU needed and allocate memory
        iwk = (lfil+2)*nnz;

#if DEBUG_MODE > 0
        if (iluparam->ILU_type == ILUtp) {
            printf("### WARNING: iludata->type = %d not supported!\n",
                iluparam->ILU_type);
        }
#endif

        // setup preconditioner
        iludata->type  = 0; // Must be initialized
        iludata->iperm = NULL;
        iludata->A     = NULL; // No need for BSR matrix
        iludata->row   = iludata->col=n;
        iludata->nb    = nb;

        fasp_mem_free(ijlu);
        ijlu = (INT*)fasp_mem_calloc(iwk,sizeof(INT));

        fasp_mem_free(uptr);
        uptr = (INT*)fasp_mem_calloc(A->ROW,sizeof(INT));

#if DEBUG_MODE > 1
        printf("### DEBUG: symbolic factorization ...  \n");
#endif

        fasp_gettime(&symbolic_start);

        // ILU decomposition
        // (1) symbolic factoration
        fasp_symbfactor(A->ROW,A->JA,A->IA,lfil,iwk,&nzlu,ijlu,uptr,&ierr);

        fasp_gettime(&symbolic_end);

#if prtlvl > PRINT_MIN
        printf("ILU symbolic factorization time = %f\n", symbolic_end-symbolic_start);
#endif

        nwork = 5*A->ROW*A->nb;
        iludata->nzlu  = nzlu;
        iludata->nwork = nwork;
        iludata->ijlu  = (INT*)fasp_mem_calloc(nzlu,sizeof(INT));
        iludata->luval = (REAL*)fasp_mem_calloc(nzlu*nb2,sizeof(REAL));
        iludata->work  = (REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
        memcpy(iludata->ijlu,ijlu,nzlu*sizeof(INT));
        fasp_mem_free(ijlu);  ijlu = NULL;

        fasp_darray_set(nzlu*nb2, iludata->luval, 0.0);
        iludata->uptr = NULL; iludata->ic = NULL; iludata->icmap = NULL;

        topologic_sort_ILU(iludata);
#if DEBUG_MODE > 1
        printf("### DEBUG: fill-in = %d, nwork = %d\n", lfil, nwork);
        printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif

        if ( ierr != 0 ) {
            printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
            status = ERROR_SOLVER_ILUSETUP;
            goto FINISHED;
        }

        if ( iwk < nzlu ) {
            printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
            status = ERROR_SOLVER_ILUSETUP;
            goto FINISHED;
        }
   } else if (step==2) {
```

```

#if DEBUG_MODE > 1
    printf("### DEBUG: numerical factorization ...  \n");
#endif

        fasp_gettime(&numfac_start);

        // (2) numerical factoration
        numfactor_levsch(A, iludata->luval, iludata->ijlu, uptr, iludata->nlevL,
                         iludata->ilevL, iludata->jlevL);
        fasp_gettime(&numfac_end);

#if prtlvl > PRINT_MIN
    printf("ILU numerical factorization time = %f\n", numfac_end-numfac_start);
#endif
    } else {

FINISHED:
//    fasp_mem_free(ijlu);  ijlu = NULL;
        fasp_mem_free(uptr);  uptr = NULL;
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;
        printf("BSR ILU(%d)-ls setup costs %f seconds.\n", lfil, setup_duration);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif


    return status;
}

SHORT fasp_ilu_dbsr_setup_mc_omp (dBSRmat    *A,
                                  dCSRmat    *Ap,
                                  ILU_data   *iludata,
                                  ILU_param  *iluparam)
{
    INT status;
    AMG_data *mgl=fasp_amg_data_create(1);
    dCSRmat pp, Ap1;
    dBSRmat A_LU;

    if (iluparam->ILU_lfil==0) {  //for ILU0
        mgl[0].A = fasp_dcsr_sympart(Ap);
    }
    else if (iluparam->ILU_lfil==1) {  // for ILU1
        Ap1 = fasp_dcsr_create(Ap->row,Ap->col, Ap->nnz);
        fasp_dcsr_cp(Ap, &Ap1);
        fasp_blas_dcsr_mxm (Ap,&Ap1,&pp);
        mgl[0].A = fasp_dcsr_sympart(&pp);
        fasp_dcsr_free(&Ap1);
        fasp_dcsr_free(&pp);
    }

    mgl->num_levels = 20;

    mulcol_independ_set(mgl, 1);

    A_LU = fasp_dbsr_perm(A, mgl[0].icmap);

    // hold color info with nlevl, ilevL and jlevL.
    iludata->nlevL = mgl[0].colors;
    iludata->ilevL = mgl[0].ic;
    iludata->jlevL = mgl[0].icmap;
    iludata->nlevU = 0;
    iludata->ilevU = NULL;
    iludata->jlevU = NULL;
    iludata->A     = NULL; // No need for BSR matrix

#if DEBUG_MODE > 0
    if (iluparam->ILU_type == ILUtp) {
        printf("### WARNING: iludata->type = %d not supported!\n",
               iluparam->ILU_type);
    }
#endif

    // setup preconditioner
    iludata->type  = 0; // Must be initialized
    iludata->iperm = NULL;

    status = fasp_ilu_dbsr_setup_omp(&A_LU,iludata,iluparam);

    fasp_dcsr_free(&mgl[0].A);
    fasp_dbsr_free(&A_LU);

    return status;
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static INT numfactor (dBSRmat   *A,
                      REAL      *luval,
                      INT       *jlu,
                      INT       *uptr)
{
    INT n=A->ROW,nb=A->nb, nb2=nb*nb, ib, ibstart,ibstart1;
    INT k, indj, inds, indja,jluj, jlus, ijaj;
    REAL  *mult,*mult1;
    INT *colptrs;
    INT status=FASP_SUCCESS;

    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));

    //for (k=0;k<n;k++) colptrs[k]=0;
    memset(colptrs, 0, sizeof(INT)*n);

    switch (nb) {

        case 1:

            for (k = 0; k < n; ++k) {

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                    colptrs[jlu[indj]] = indj;
                    ibstart=indj*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                }

                colptrs[k] =  k;

                for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                    ijaj = A->JA[indja];
                    ibstart=colptrs[ijaj]*nb2;
                    ibstart1=indja*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                }

                for (indj = jlu[k]; indj < uptr[k]; ++indj) {

                    jluj = jlu[indj];

                    luval[indj] = luval[indj]*luval[jluj];
                    mult[0] = luval[indj];

                    for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                        jlus = jlu[inds];
                        if (colptrs[jlus] != 0)
                            luval[colptrs[jlus]] = luval[colptrs[jlus]] - mult[0]*luval[inds];
                    }

                }

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;

                colptrs[k] =  0;
                luval[k] = 1.0/luval[k];
            }

            break;

        case 3:

            for (k = 0; k < n; ++k) {

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                    colptrs[jlu[indj]] = indj;
                    ibstart=indj*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                }

                colptrs[k] =  k;

                for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                    ijaj = A->JA[indja];
                    ibstart=colptrs[ijaj]*nb2;
                    ibstart1=indja*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                }

                for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                    jluj = jlu[indj];

                    ibstart=indj*nb2;
                    fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];

                    for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                        jlus = jlu[inds];
                        if (colptrs[jlus] != 0) {
                            fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
                            ibstart=colptrs[jlus]*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                        }
                    }

                }

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;

                colptrs[k] =  0;

                fasp_smat_inv_nc3(&(luval[k*nb2]));
            }

            break;

        case -5:

            for (k = 0; k < n; ++k) {

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                    colptrs[jlu[indj]] = indj;
                    ibstart=indj*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                }

                colptrs[k] =  k;

                for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                    ijaj = A->JA[indja];
                    ibstart=colptrs[ijaj]*nb2;
                    ibstart1=indja*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                }

                for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                    jluj = jlu[indj];

                    ibstart=indj*nb2;
                    fasp_blas_smat_mul_nc5(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];

                    for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                        jlus = jlu[inds];
                        if (colptrs[jlus] != 0) {
                            fasp_blas_smat_mul_nc5(mult,&(luval[inds*nb2]),mult1);
                            ibstart=colptrs[jlus]*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                        }
                    }

                }

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;

                colptrs[k] =  0;

                // fasp_smat_inv_nc5(&(luval[k*nb2])); // not numerically stable --zcs 04/26/2021
                status = fasp_smat_invp_nc(&(luval[k*nb2]), 5);
            }

            break;

        case -7:

            for (k = 0; k < n; ++k) {

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                    colptrs[jlu[indj]] = indj;
                    ibstart=indj*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                }

                colptrs[k] =  k;

                for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                    ijaj = A->JA[indja];
                    ibstart=colptrs[ijaj]*nb2;
                    ibstart1=indja*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                }

                for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                    jluj = jlu[indj];

                    ibstart=indj*nb2;
                    fasp_blas_smat_mul_nc7(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];

                    for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                        jlus = jlu[inds];
                        if (colptrs[jlus] != 0) {
                            fasp_blas_smat_mul_nc7(mult,&(luval[inds*nb2]),mult1);
                            ibstart=colptrs[jlus]*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                        }
                    }

                }

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;

                colptrs[k] =  0;

                // fasp_smat_inv(&(luval[k*nb2]),nb); // not numerically stable --zcs 04/26/2021
                status = fasp_smat_invp_nc(&(luval[k*nb2]), nb);
            }

            break;

        default:

            for (k=0;k<n;k++) {

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                    colptrs[jlu[indj]] = indj;
                    ibstart=indj*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                }

                colptrs[k] =  k;

                for (indja = A->IA[k]; indja < A->IA[k+1]; indja++) {
                    ijaj = A->JA[indja];
                    ibstart=colptrs[ijaj]*nb2;
                    ibstart1=indja*nb2;
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                }

                for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                    jluj = jlu[indj];

                    ibstart=indj*nb2;
                    fasp_blas_smat_mul(&(luval[ibstart]),&(luval[jluj*nb2]),mult,nb);
                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];

                    for (inds = uptr[jluj]; inds < jlu[jluj+1]; inds++) {
                        jlus = jlu[inds];
                        if (colptrs[jlus] != 0) {
                            fasp_blas_smat_mul(mult,&(luval[inds*nb2]),mult1,nb);
                            ibstart=colptrs[jlus]*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                        }
                    }

                }

                for (indj = jlu[k]; indj < jlu[k+1]; ++indj)
                    colptrs[jlu[indj]] = 0;

                colptrs[k] =  0;

                //fasp_smat_inv(&(luval[k*nb2]),nb); // not numerically stable --zcs 04/26/2021
                status = fasp_smat_invp_nc(&(luval[k * nb2]), nb);
            }
    }

    fasp_mem_free(colptrs);  colptrs = NULL;
    fasp_mem_free(mult);     mult    = NULL;
    fasp_mem_free(mult1);    mult1   = NULL;

    return status;
}

static INT numfactor_mulcol (dBSRmat   *A,
                             REAL      *luval,
                             INT       *jlu,
                             INT       *uptr,
                             INT        ncolors,
                             INT       *ic,
                             INT       *icmap)
{
    INT status = FASP_SUCCESS;

#ifdef _OPENMP
    INT   n = A->ROW, nb = A->nb, nb2 = nb*nb;
    INT   ib, ibstart,ibstart1;
    INT   k, i, indj, inds, indja,jluj, jlus, ijaj, tmp;
    REAL  *mult, *mult1;
    INT   *colptrs;

    switch (nb) {

        case 1:
            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,colptrs,tmp)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
#pragma omp for
                    for (k = ic[i]; k < ic[i+1]; ++k) {
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            luval[indj] = luval[indj]*luval[jluj];
                            tmp = luval[indj];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0)
                                    luval[colptrs[jlus]] = luval[colptrs[jlus]] - tmp*luval[inds];
                            }

                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        luval[k] = 1.0/luval[k];
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                }
            }

            break;

        case 2:

            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (k = ic[i]; k < ic[i+1]; ++k) {
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc2(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc2(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc2(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        case 3:

            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (k = ic[i]; k < ic[i+1]; ++k) {
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc3(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        default:
        {
            if (nb > 3) printf("Multi-thread ILU numerical decomposition for %d\
components has not been implemented!!!", nb);
            exit(0);
        }
    }

#endif

    return status;
}

static INT numfactor_levsch (dBSRmat *A,
                             REAL *luval,
                             INT *jlu,
                             INT *uptr,
                             INT ncolors,
                             INT *ic,
                             INT *icmap)
{
    INT status = FASP_SUCCESS;

#ifdef _OPENMP
    INT n = A->ROW, nb = A->nb, nb2 = nb*nb;
    INT ib, ibstart,ibstart1;
    INT k, i, indj, inds, indja, jluj, jlus, ijaj, tmp, ii;
    REAL *mult, *mult1;
    INT  *colptrs;

    switch (nb) {

        case 1:
            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,colptrs,tmp)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
#pragma omp for
                    for (k = ic[i]; k < ic[i+1]; ++k) {
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            luval[indj] = luval[indj]*luval[jluj];
                            tmp = luval[indj];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0)
```

```
                                    luval[colptrs[jlus]] = luval[colptrs[jlus]] - tmp*luval[inds];
                            }

                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        luval[k] = 1.0/luval[k];
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                }
            }

            break;
        case 2:

            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (ii = ic[i]; ii < ic[i+1]; ++ii) {
                        k = icmap[ii];
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc2(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc2(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc2(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        case 3:

            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (ii = ic[i]; ii < ic[i+1]; ++ii) {
                        k = icmap[ii];
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc3(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc3(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc3(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        case 4:

            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (ii = ic[i]; ii < ic[i+1]; ++ii) {
                        k = icmap[ii];
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc4(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc4(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc4(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        case 5:

```

```
            for (i = 0; i < ncolors; ++i) {
#pragma omp parallel private(k,indj,ibstart,ib,indja,ijaj,ibstart1,jluj,inds,jlus,mult,mult1,colptrs,ii)
                {
                    colptrs=(INT*)fasp_mem_calloc(n,sizeof(INT));
                    memset(colptrs, 0, sizeof(INT)*n);
                    mult=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
                    mult1=(REAL*)fasp_mem_calloc(nb2,sizeof(REAL));
#pragma omp for
                    for (ii = ic[i]; ii < ic[i+1]; ++ii) {
                        k = icmap[ii];
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) {
                            colptrs[jlu[indj]] = indj;
                            ibstart=indj*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = 0;
                        }
                        colptrs[k] =  k;
                        for (indja = A->IA[k]; indja < A->IA[k+1]; ++indja) {
                            ijaj = A->JA[indja];
                            ibstart=colptrs[ijaj]*nb2;
                            ibstart1=indja*nb2;
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib] = A->val[ibstart1+ib];
                        }
                        for (indj = jlu[k]; indj < uptr[k]; ++indj) {
                            jluj = jlu[indj];
                            ibstart=indj*nb2;
                            fasp_blas_smat_mul_nc5(&(luval[ibstart]),&(luval[jluj*nb2]),mult);
                            for (ib=0;ib<nb2;++ib) luval[ibstart+ib]=mult[ib];
                            for (inds = uptr[jluj]; inds < jlu[jluj+1]; ++inds) {
                                jlus = jlu[inds];
                                if (colptrs[jlus] != 0) {
                                    fasp_blas_smat_mul_nc5(mult,&(luval[inds*nb2]),mult1);
                                    ibstart=colptrs[jlus]*nb2;
                                    for (ib=0;ib<nb2;++ib) luval[ibstart+ib]-=mult1[ib];
                                }
                            }
                        }
                        for (indj = jlu[k]; indj < jlu[k+1]; ++indj) colptrs[jlu[indj]] = 0;
                        colptrs[k] =  0;
                        fasp_smat_inv_nc5(&(luval[k*nb2]));
                    }
                    fasp_mem_free(colptrs); colptrs = NULL;
                    fasp_mem_free(mult);    mult    = NULL;
                    fasp_mem_free(mult1);   mult1   = NULL;
                }
            }
            break;

        default:
        {
            if (nb > 5) printf("Multi-thread ILU numerical decomposition for %d components has not been
implemented!!!\n", nb);
            exit(0);
            break;
        }
    }

#endif

    return status;
}

static void generate_S_theta (dCSRmat *A,
                              iCSRmat *S,
                              REAL     theta)
{
    const INT row=A->row, col=A->col;
    const INT row_plus_one = row+1;
    const INT nnz=A->IA[row]-A->IA[0];

    INT index, i, j, begin_row, end_row;
    INT *ia=A->IA, *ja=A->JA;
    REAL *aj=A->val;

    // get the diagnal entry of A
    //dvector diag; fasp_dcsr_getdiag(0, A, &diag);

    /* generate S */
    REAL row_abs_sum;

    // copy the structure of A to S
    S->row=row; S->col=col; S->nnz=nnz; S->val=NULL;

    S->IA=(INT*)fasp_mem_calloc(row_plus_one, sizeof(INT));

    S->JA=(INT*)fasp_mem_calloc(nnz, sizeof(INT));

    fasp_iarray_cp(row_plus_one, ia, S->IA);
    fasp_iarray_cp(nnz, ja, S->JA);

    for (i=0;i<row;++i) {
        /* compute scaling factor and row sum */
        row_abs_sum=0;

        begin_row=ia[i]; end_row=ia[i+1];

        for (j=begin_row;j<end_row;j++) row_abs_sum+=ABS(aj[j]);

        row_abs_sum = row_abs_sum*theta;

        /* deal with the diagonal element of S */
        //  for (j=begin_row;j<end_row;j++) {
        //      if (ja[j]==i) {S->JA[j]=-1; break;}
        //  }

        /* deal with  the element of S */
        for (j=begin_row;j<end_row;j++){
            /* if $\sum_{j=1}^n |a_{ij}|*theta>= |a_{ij}|$ */
            if ( (row_abs_sum >= ABS(aj[j])) && (ja[j] !=i) ) S->JA[j]=-1;
        }
    } // end for i

    /* Compress the strength matrix */
    index=0;
    for (i=0;i<row;++i) {
        S->IA[i]=index;
        begin_row=ia[i]; end_row=ia[i+1]-1;
        for (j=begin_row;j<=end_row;j++) {
            if (S->JA[j]>-1) {
                S->JA[index]=S->JA[j];
                index++;
            }
        }
    }

    if (index > 0) {
        S->IA[row]=index;
        S->nnz=index;
        S->JA=(INT*)fasp_mem_realloc(S->JA,index*sizeof(INT));
    }
    else {
        S->nnz = 0;
        S->JA = NULL;
    }
}

static void multicoloring (AMG_data *mgl,
                           REAL      theta,
                           INT      *rowmax,
                           INT      *groups)
{
    INT k, i, j, pre, group, iend;
    INT icount;
    INT front, rear;
    INT *IA, *JA;
    INT *cq, *newr;

    const INT n = mgl->A.row;
    dCSRmat   A = mgl->A;
    iCSRmat   S;

    S.IA = S.JA = NULL; S.val = NULL;

    theta = MAX(0.0, MIN(1.0, theta));

    if (theta > 0.0 && theta < 1.0) {
        generate_S_theta(&A, &S, theta);
        IA = S.IA;
        JA = S.JA;
    }
    else if (theta == 1.0) {

        mgl->ic = (INT*)malloc(sizeof(INT)*2);
        mgl->icmap = (INT *)malloc(sizeof(INT)*(n+1));
        mgl->ic[0] = 0;
        mgl->ic[1] = n;
        for(k=0; k<n; k++)  mgl->icmap[k]= k;

        mgl->colors = 1;
        *groups = 1;
        *rowmax = 1;

        printf("### WARNING: Theta = %lf!  [%s]\n", theta, __FUNCTION__);

        return;
    }
    else {
        IA = A.IA;
        JA = A.JA;
    }

    cq = (INT *)malloc(sizeof(INT)*(n+1));
    newr = (INT *)malloc(sizeof(INT)*(n+1));

#ifdef _OPENMP
#pragma omp parallel for private(k)
#endif
    for ( k=0; k<n; k++ ) cq[k]= k;

    group = 0;
    for ( k=0; k<n; k++ ) {
        if ((A.IA[k+1] - A.IA[k]) > group ) group = A.IA[k+1] - A.IA[k];
    }
    *rowmax = group;

    mgl->ic = (INT *)malloc(sizeof(INT)*(group+2));
    mgl->icmap = (INT *)malloc(sizeof(INT)*(n+1));

    front = n-1;
    rear = n-1;

    memset(newr, -1, sizeof(INT)*(n+1));
    memset(mgl->icmap, 0, sizeof(INT)*n);

    group=0;
    icount = 0;
    mgl->ic[0] = 0;
    pre=0;

    do {
        //front = (front+1)%n;
        front ++;
        if (front == n ) front =0; // front = front < n ?  front :  0 ;
        i = cq[front];

        if(i <= pre) {
            mgl->ic[group] = icount;
            mgl->icmap[icount] = i;
            group++;
            icount++;
#if 0
            if ((IA[i+1]-IA[i]) > igold)
                iend = MIN(IA[i+1], (IA[i] + igold));
            else
#endif
                iend = IA[i+1];

            for (j= IA[i]; j< iend; j++)  newr[JA[j]] = group;
        }
        else if (newr[i] == group) {
            //rear = (rear +1)%n;
            rear ++;
            if (rear == n) rear = 0;
            cq[rear] = i;
        }
        else {
            mgl->icmap[icount] = i;
            icount++;
#if  0
            if ((IA[i+1] - IA[i]) > igold)  iend =MIN(IA[i+1], (IA[i] + igold));
            else
#endif
                iend = IA[i+1];
            for (j = IA[i]; j< iend; j++)  newr[JA[j]] =  group;
        }
        pre=i;

    } while(rear != front);

    mgl->ic[group] = icount;
    mgl->colors = group;
    *groups = group;

    free(cq);
    free(newr);

    fasp_mem_free(S.IA); S.IA = NULL;
    fasp_mem_free(S.JA); S.JA = NULL;

    return;
}

void topologic_sort_ILU (ILU_data *iludata)
{
    INT i, j, k, l;
    INT nlevL, nlevU;

    INT n = iludata->row;
    INT *ijlu = iludata->ijlu;

    INT *level = (INT *)fasp_mem_calloc(n, sizeof(INT));
    INT *jlevL = (INT *)fasp_mem_calloc(n, sizeof(INT));
    INT *ilevL = (INT *)fasp_mem_calloc(n+1, sizeof(INT));

    INT *jlevU = (INT *)fasp_mem_calloc(n, sizeof(INT));
    INT *ilevU = (INT *)fasp_mem_calloc(n+1, sizeof(INT));

    nlevL = 0;
    ilevL[0] = 0;

    // form level for each row of lower triangular matrix.
    for (i=0; i<n; i++) {
        l = 0;
        for(j=ijlu[i]; j<ijlu[i+1]; j++) if (ijlu[j]<=i) l = MAX(l, level[ijlu[j]]);
        level[i] = l+1;
        ilevL[l+1] ++;
        nlevL = MAX(nlevL, l+1);
    }

    for (i=1; i<=nlevL; i++) ilevL[i] += ilevL[i-1];

    for (i=0; i<n; i++) {
        k = ilevL[level[i]-1];
        jlevL[k] = i;
        ilevL[level[i]-1]++;
    }

    for (i=nlevL-1; i>0; i--) ilevL[i] = ilevL[i-1];

    // form level for each row of upper triangular matrix.
    nlevU = 0;
    ilevL[0] = 0;

    for (i=0; i<n; i++) level[i] = 0;

    ilevU[0] = 0;

    for (i=n-1; i>=0; i--) {
        l = 0;
        for (j=ijlu[i]; j<ijlu[i+1]; j++) if (ijlu[j]>=i) l = MAX(l, level[ijlu[j]]);
        level[i] = l+1;
        ilevU[l+1] ++;
        nlevU = MAX(nlevU, l+1);
    }

    for (i=1; i<=nlevU; i++) ilevU[i] += ilevU[i-1];

    for (i=n-1; i>=0; i--) {
        k = ilevU[level[i]-1];
        jlevU[k] = i;
        ilevU[level[i]-1]++;
    }

    for (i=nlevU-1; i>0; i--) ilevU[i] = ilevU[i-1];

    ilevU[0] = 0;

    iludata->nlevL = nlevL+1; iludata->ilevL = ilevL;iludata->jlevL = jlevL;
    iludata->nlevU = nlevU+1; iludata->ilevU = ilevU;iludata->jlevU = jlevU;

    fasp_mem_free(level); level = NULL;
}

void mulcol_independ_set (AMG_data *mgl,
                          INT       gslvl)
{

    INT Colors, rowmax, level, prtlvl = 0;

    REAL theta = 0.00;

    INT maxlvl = MIN(gslvl, mgl->num_levels-1);

#ifdef _OPENMP
#pragma omp parallel for private(level,rowmax,Colors) schedule(static, 1)
#endif
    for ( level=0; level<maxlvl; level++ ) {

        multicoloring(&mgl[level], theta, &rowmax, &Colors);

        // print
        if ( prtlvl > PRINT_MIN )
            printf("mgl[%3d].A.row = %12d rowmax = %5d rowavg = %7.2lf colors = %5d theta = %le\n",
                   level, mgl[level].A.row, rowmax, (double)mgl[level].A.nnz/mgl[level].A.row,
                   mgl[level].colors, theta);
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.55 BlaILUSetupCSR.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, BlaILU.c, BlaSparseCSR.c, and PreDataInit.c

Definition in file BlaILUSetupCSR.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

A	Pointer to dCSRmat matrix
iludata	Pointer to ILU_data
iluparam	Pointer to ILU_param

Returns

FASP_SUCCESS if successed; 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 BlaILUSetupCSR.c.

9.56 BlaILUSetupCSR.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_functs.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;
```

```

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: m=%d, n=%d, nnz=%d\n", A->row, n, nnz);
#endif

    fasp_gettime(&setup_start);

    // Expected amount of memory for ILU needed and allocate memory
    switch (type) {
        case ILUt:
            iwk=100*nnz;     // iwk is the maxim possible nnz for ILU
            lfilt=(int)floor(n*0.5)+1;
            break;
        case ILUtp:
            iwk=100*nnz;     // iwk is the maxim possible nnz for ILU
            lfilt=(int)floor(n*0.5)+1;
            break;
        default:  // ILUk
            if (lfil == 0) iwk=nnz+500;
            else iwk=(lfil+5)*nnz;
    }

    nwork = 4*n;

#if DEBUG_MODE > 1
    printf("### DEBUG: fill-in = %d, iwk = %d, nwork = %d\n", lfil, iwk, nwork);
#endif

    // setup ILU preconditioner
    iludata->A     = A; // save a pointer to the coeff matrix for ILUtp
    iludata->row   = iludata->col   = n;
    iludata->ilevL = iludata->jlevL = NULL;
    iludata->ilevU = iludata->jlevU = NULL;
    iludata->iperm = NULL;
    iludata->type  = type;

    fasp_ilu_data_create(iwk, nwork, iludata);

#if DEBUG_MODE > 1
    printf("### DEBUG: memory usage after %s:  \n", __FUNCTION__);
    fasp_mem_usage();
#endif

    // ILU decomposition
    ijlu  = iludata->ijlu;
    luval = iludata->luval;

    switch (type) {

        case ILUt:
            fasp_ilut (n, A->val, A->JA, A->IA, lfilt, ILU_droptol, luval, ijlu,
                       iwk, &ierr, &nzlu);
            break;

        case ILUtp:
            iperm = iludata->iperm;
            fasp_ilutp (n, A->val, A->JA, A->IA, lfilt, ILU_droptol, permtol,
                        mbloc, luval, ijlu, iperm, iwk, &ierr, &nzlu);
            break;

        default:  // ILUk
            fasp_iluk (n, A->val, A->JA, A->IA, lfil, luval, ijlu, iwk,
                       &ierr, &nzlu);
            break;

    }
   if (ierr != -4)
    fasp_dcsr_shift(A, -1);

#if DEBUG_MODE > 1
    printf("### DEBUG: memory usage after ILU setup:  \n");
    fasp_mem_usage();
#endif

    iludata->nzlu  = nzlu;
    iludata->nwork = nwork;

#if DEBUG_MODE > 1
    printf("### DEBUG: iwk = %d, nzlu = %d\n", iwk, nzlu);
#endif
```

```

    if (ierr!=0) {
        printf("### ERROR: ILU setup failed (ierr=%d)!  [%s]\n", ierr, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if (iwk<nzlu) {
        printf("### ERROR: ILU needs more RAM %d!  [%s]\n", iwk-nzlu, __FUNCTION__);
        status = ERROR_SOLVER_ILUSETUP;
        goto FINISHED;
    }

    if (print_level>PRINT_NONE) {
        fasp_gettime(&setup_end);
        setup_duration = setup_end - setup_start;

        switch (type) {
            case ILUt:
                printf("ILUt setup costs %f seconds.\n", setup_duration);
                break;
            case ILUtp:
                printf("ILUtp setup costs %f seconds.\n", setup_duration);
                break;
            default:  // ILUk
                printf("ILUk setup costs %f seconds.\n", setup_duration);
                break;
        }
    }

FINISHED:

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.57 BlaILUSetupSTR.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) decoposition 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

Definition in file BlaILUSetupSTR.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

A	Pointer to dSTRmat
LU	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 BlaILUSetupSTR.c.

9.57.2.2 fasp_ilu_dstr_setup1()

```
void fasp_ilu_dstr_setup1 (
            dSTRmat * A,
            dSTRmat * LU )
```

Get ILU(1) decoposition of a structured matrix A.

Parameters

A	Pointer to oringinal structured matrix of REAL type
LU	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 BlaILUSetupSTR.c.

9.58 BlaILUSetupSTR.c

Go to the documentation of this file.

```
#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_ilu_dstr_setup0 (dSTRmat  *A,
                           dSTRmat  *LU)
{
    // local variables
    INT i,i1,ix,ixy,ii;
    INT *LUoffsets;
    INT nline, nplane;

    // information of A
    INT nc = A->nc;
    INT nc2 = nc*nc;
    INT nx = A->nx;
    INT ny = A->ny;
    INT nz = A->nz;
    INT nxy = A->nxy;
    INT ngrid = A->ngrid;
    INT nband = A->nband;

    INT  *offsets = A->offsets;
    REAL *smat=(REAL *)fasp_mem_calloc(nc2,sizeof(REAL));
    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL;
    REAL *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;

    // initialize
    if (nx == 1) {
        nline = ny;
        nplane = ngrid;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = ngrid;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = ngrid;
    }
    else {
        nline = nx;
        nplane = nxy;
    }

    // check number of bands
    if (nband == 4) {
        LUoffsets=(INT *)fasp_mem_calloc(4,sizeof(INT));
        LUoffsets[0]=-1; LUoffsets[1]=1; LUoffsets[2]=-nline; LUoffsets[3]=nline;
    }
    else if (nband == 6) {
        LUoffsets=(INT *)fasp_mem_calloc(6,sizeof(INT));
        LUoffsets[0]=-1; LUoffsets[1]=1; LUoffsets[2]=-nline;
        LUoffsets[3]=nline; LUoffsets[4]=-nplane; LUoffsets[5]=nplane;
    }
    else {
        printf("%s:  number of bands for structured ILU is illegal!\n", __FUNCTION__);
        return;
    }

    // allocate memory to store LU decomposition
    fasp_dstr_alloc(nx, ny, nz, nxy, ngrid, nband, nc, offsets, LU);

    // copy diagonal
    memcpy(LU->diag, diag, (ngrid*nc2)*sizeof(REAL));

    // check offsets and copy off-diagonals
    for (i=0; i<nband; ++i) {
        if (offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
            memcpy(LU->offdiag[0],offdiag0,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else if (offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
            memcpy(LU->offdiag[1],offdiag1,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else if (offsets[i] == -nline) {
            offdiag2 = A->offdiag[i];
            memcpy(LU->offdiag[2],offdiag2,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else if (offsets[i] == nline) {
            offdiag3 = A->offdiag[i];
            memcpy(LU->offdiag[3],offdiag3,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else if (offsets[i] == -nplane) {
            offdiag4 = A->offdiag[i];
            memcpy(LU->offdiag[4],offdiag4,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else if (offsets[i] == nplane) {
            offdiag5 = A->offdiag[i];
            memcpy(LU->offdiag[5],offdiag5,((ngrid - ABS(offsets[i]))*nc2)*sizeof(REAL));
        }
        else {
            printf("### ERROR: Illegal offset for ILU! [%s]\n", __FUNCTION__);
            return;
        }
    }

    // Setup
    if (nc == 1) {

        LU->diag[0]=1.0/(LU->diag[0]);

        for (i=1;i<ngrid;++i) {

            LU->offdiag[0][i-1]=(offdiag0[i-1])*(LU->diag[i-1]);
            if (i>=nline)
                LU->offdiag[2][i-nline]=(offdiag2[i-nline])*(LU->diag[i-nline]);
            if (i>=nplane)
                LU->offdiag[4][i-nplane]=(offdiag0[i-nplane])*(LU->diag[i-nplane]);

            LU->diag[i]=diag[i]-(LU->offdiag[0][i-1])*(LU->offdiag[1][i-1]);

            if (i>=nline)
                LU->diag[i]=LU->diag[i]-(LU->offdiag[2][i-nline])*(LU->offdiag[3][i-nline]);
            if (i>=nplane)
                LU->diag[i]=LU->diag[i]-(LU->offdiag[4][i-nplane])*(LU->offdiag[5][i-nplane]);

            LU->diag[i]=1.0/(LU->diag[i]);

        } // end for (i=1; i<ngrid; ++i)

    } // end if (nc == 1)

    else if (nc == 3) {

        fasp_smat_inv_nc3(LU->diag);

        for (i=1;i<ngrid;++i) {

            i1=(i-1)*9;
            ix=(i-nline)*9;
            ixy=(i-nplane)*9;
            ii=i*9;

            fasp_blas_smat_mul_nc3(&(offdiag0[i1]),&(LU->diag[i1]),&(LU->offdiag[0][i1]));

            if (i>=nline)
                fasp_blas_smat_mul_nc3(&(offdiag2[ix]),&(LU->diag[ix]),&(LU->offdiag[2][ix]));
            if (i>=nplane)
                fasp_blas_smat_mul_nc3(&(offdiag4[ixy]),&(LU->diag[ixy]),&(LU->offdiag[4][ixy]));

            fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1]),&(LU->offdiag[1][i1]),smat);

            fasp_blas_darray_axpyz_nc3(-1,smat,&(diag[ii]),&(LU->diag[ii]));

            if (i>=nline) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix]),&(LU->offdiag[3][ix]),smat);
                fasp_blas_darray_axpy_nc3(-1.0,smat,&(LU->diag[ii]));
            } //end if (i>=nline)

            if (i>=nplane) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixy]),&(LU->offdiag[5][ixy]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ii]));
            } // end if (i>=nplane)

            fasp_smat_inv_nc3(&(LU->diag[ii]));

        } // end for(i=1;i<A->ngrid;++i)

    }  // end if (nc == 3)

    else if (nc == 5) {

        fasp_smat_inv_nc5(LU->diag);

        for (i=1;i<ngrid;++i) {

            i1=(i-1)*25;
            ix=(i-nline)*25;
            ixy=(i-nplane)*25;
            ii=i*25;

            fasp_blas_smat_mul_nc5(&(offdiag0[i1]),&(LU->diag[i1]),&(LU->offdiag[0][i1]));

            if (i>=nline)
                fasp_blas_smat_mul_nc5(&(offdiag2[ix]),&(LU->diag[ix]),&(LU->offdiag[2][ix]));
            if (i>=nplane)
                fasp_blas_smat_mul_nc5(&(offdiag4[ixy]),&(LU->diag[ixy]),&(LU->offdiag[4][ixy]));

            fasp_blas_smat_mul_nc5(&(LU->offdiag[0][i1]),&(LU->offdiag[1][i1]),smat);

            fasp_blas_darray_axpyz_nc5(-1.0,smat,&(diag[ii]),&(LU->diag[ii]));

            if (i>=nline) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix]),&(LU->offdiag[3][ix]),smat);
                fasp_blas_darray_axpy_nc5(-1.0,smat,&(LU->diag[ii]));
            } //end if (i>=nline)

            if (i>=nplane) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixy]),&(LU->offdiag[5][ixy]),smat);
                fasp_blas_darray_axpy_nc5(-1.0,smat,&(LU->diag[ii]));
            } // end if (i>=nplane)

            fasp_smat_inv_nc5(&(LU->diag[ii]));

        } // end for(i=1;i<A->ngrid;++i)

    }  // end if (nc == 5)

    else if (nc == 7) {

        fasp_smat_inv_nc7(LU->diag);

        for (i=1;i<ngrid;++i) {

            i1=(i-1)*49;
            ix=(i-nline)*49;
            ixy=(i-nplane)*49;
            ii=i*49;

            fasp_blas_smat_mul_nc7(&(offdiag0[i1]),&(LU->diag[i1]),&(LU->offdiag[0][i1]));

            if (i>=nline)
                fasp_blas_smat_mul_nc7(&(offdiag2[ix]),&(LU->diag[ix]),&(LU->offdiag[2][ix]));
            if (i>=nplane)
                fasp_blas_smat_mul_nc7(&(offdiag4[ixy]),&(LU->diag[ixy]),&(LU->offdiag[4][ixy]));

            fasp_blas_smat_mul_nc7(&(LU->offdiag[0][i1]),&(LU->offdiag[1][i1]),smat);

            fasp_blas_darray_axpyz_nc7(-1.0,smat,&(diag[ii]),&(LU->diag[ii]));

            if (i>=nline) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ix]),&(LU->offdiag[3][ix]),smat);
                fasp_blas_darray_axpy_nc7(-1.0,smat,&(LU->diag[ii]));
            } //end if (i>=nline)

            if (i>=nplane) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixy]),&(LU->offdiag[5][ixy]),smat);
                fasp_blas_darray_axpy_nc7(-1.0,smat,&(LU->diag[ii]));
            } // end if (i>=nplane)

            fasp_smat_inv_nc7(&(LU->diag[ii]));

        } // end for(i=1;i<A->ngrid;++i)

    }  // end if (nc == 7)

    else {

        fasp_smat_inv(LU->diag,nc);

        for (i=1;i<ngrid;++i) {

            i1=(i-1)*nc2;
            ix=(i-nline)*nc2;
            ixy=(i-nplane)*nc2;
            ii=i*nc2;

            fasp_blas_smat_mul(&(offdiag0[i1]),&(LU->diag[i1]),&(LU->offdiag[0][i1]),nc);

            if (i>=nline)
                fasp_blas_smat_mul(&(offdiag2[ix]),&(LU->diag[ix]),&(LU->offdiag[2][ix]),nc);
            if (i>=nplane)
                fasp_blas_smat_mul(&(offdiag4[ixy]),&(LU->diag[ixy]),&(LU->offdiag[4][ixy]),nc);

            fasp_blas_smat_mul(&(LU->offdiag[0][i1]),&(LU->offdiag[1][i1]),smat,nc);

            fasp_blas_darray_axpyz(nc2,-1,smat,&(diag[ii]),&(LU->diag[ii]));

            if (i>=nline) {
                fasp_blas_smat_mul(&(LU->offdiag[2][ix]),&(LU->offdiag[3][ix]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ii]));
            } //end if (i>=nline)

            if (i>=nplane) {
                fasp_blas_smat_mul(&(LU->offdiag[4][ixy]),&(LU->offdiag[5][ixy]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ii]));
            } // end if (i>=nplane)

            fasp_smat_inv(&(LU->diag[ii]),nc);

        } // end for(i=1;i<A->ngrid;++i)

    }

    fasp_mem_free(smat); smat = NULL;

    return;
}

void fasp_ilu_dstr_setup1 (dSTRmat   *A,
                           dSTRmat   *LU)
{
    const INT LUnband = 12;
    INT LUoffsets[12];

    const INT nc=A->nc, nc2=nc*nc;
    const INT nx=A->nx;
    const INT ny=A->ny;
    const INT nz=A->nz;
    const INT nxy=A->nxy;
    const INT nband=A->nband;
    const INT ngrid=A->ngrid;
    INT nline, nplane;

    INT i,j,i1,ix,ixy,ixyx,ix1,ixy1,ic,i1c,ixc,ix1c,ixyc,ixy1c,ixyxc;
    register REAL *smat,t,*tc;

    if (nx == 1) {
        nline = ny;
        nplane = ngrid;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = ngrid;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = ngrid;
    }
    else {
        nline = nx;
        nplane = nxy;
    }

    smat=(REAL *)fasp_mem_calloc(nc2,sizeof(REAL));

    tc=(REAL *)fasp_mem_calloc(nc2,sizeof(REAL));

    LUoffsets[0] = -1;
    LUoffsets[1] = 1;
    LUoffsets[2] = 1-nline;
    LUoffsets[3] = nline-1;
    LUoffsets[4] = -nline;
    LUoffsets[5] = nline;
    LUoffsets[6] = nline-nplane;
    LUoffsets[7] = nplane-nline;
    LUoffsets[8] = 1-nplane;
    LUoffsets[9] = nplane-1;
    LUoffsets[10] = -nplane;
    LUoffsets[11] = nplane;

    fasp_dstr_alloc(nx,A->ny,A->nz,nxy,ngrid,LUnband,nc,LUoffsets,LU);

    if (nband == 6) memcpy(LU->offdiag[11],A->offdiag[5],((ngrid-nxy)*nc2)*sizeof(REAL));
    memcpy(LU->diag,A->diag,nc2*sizeof(REAL));

    if (nc == 1) {
        // comput the first row
        LU->diag[0]=1.0/(LU->diag[0]);
        LU->offdiag[1][0]=A->offdiag[1][0];
        LU->offdiag[5][0]=A->offdiag[3][0];
        LU->offdiag[3][0]=0;
        LU->offdiag[7][0]=0;
        LU->offdiag[9][0]=0;

        for (i=1;i<ngrid;++i) {

            i1=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;

            // comput alpha6[i-nxy]
            if (ixy>=0)
                LU->offdiag[10][ixy]=A->offdiag[4][ixy]*LU->diag[ixy];

            // comput alpha5[ixy1]
            if (ixy1>=0) {
                t=0;

                if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[1][ixy];

                LU->offdiag[8][ixy1]=t*(LU->diag[ixy1]);
            }

            // comput alpha4[ixyx]
            if (ixyx>=0) {
                t=0;

                if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[5][ixy];
                if (ixy1>=0) t-=LU->offdiag[8][ixy1]*LU->offdiag[3][ixy1];

                LU->offdiag[6][ixyx]=t*(LU->diag[ixyx]);
            }

            // comput alpha3[ix]
            if (ix>=0) {
                t=A->offdiag[2][ix];

                if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[7][ixy];

                LU->offdiag[4][ix]=t*(LU->diag[ix]);
            }

            // comput alpha2[i-nx+1]
            if (ix1>=0) {
                t=0;

                if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[1][ix];
                if (ixy1>=0) t-=LU->offdiag[8][ixy1]*LU->offdiag[7][ixy1];

                LU->offdiag[2][ix1]=t*(LU->diag[ix1]);
            }

            // comput alpha1[i-1]
            t=A->offdiag[0][i1];

            if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[3][ix];
            if (ixy>=0) t-=LU->offdiag[10][ixy]*LU->offdiag[9][ixy];

            LU->offdiag[0][i1]=t*(LU->diag[i1]);

            // comput beta1[i]
            if (i+1<ngrid) {
                t=A->offdiag[1][i];

                if (ix1>=0) t-=LU->offdiag[2][ix1]*LU->offdiag[5][ix1];
                if (ixy1>=0) t-=LU->offdiag[8][ixy1]*LU->offdiag[11][ixy1];

                LU->offdiag[1][i]=t;
            }

            // comput beta2[i]
            if (i+nline-1<ngrid) {
                t=-LU->offdiag[0][i1]*LU->offdiag[5][i1];

                if (ixyx>=0) t-=LU->offdiag[6][ixyx]*LU->offdiag[9][ixyx];

                LU->offdiag[3][i]=t;
            }

            // comput beta3[i]
            if (i+nline<ngrid) {
                t=A->offdiag[3][i];

                if (ixyx>=0) t-=LU->offdiag[6][ixyx]*LU->offdiag[11][ixyx];

                LU->offdiag[5][i]=t;
            }

            // comput beta4[i]
            if (i+nplane-nline<ngrid) {
                t=0;

                if (ix1>=0) t-=LU->offdiag[2][ix1]*LU->offdiag[9][ix1];
                if (ix>=0) t-=LU->offdiag[4][ix]*LU->offdiag[11][ix];

                LU->offdiag[7][i]=t;
            }

            // comput beta5[i]
            if (i+nplane-1<ngrid) LU->offdiag[9][i]=-LU->offdiag[0][i1]*LU->offdiag[11][i1];

            // comput d[i]
            LU->diag[i]=A->diag[i]-(LU->offdiag[0][i1])*(LU->offdiag[1][i1]);

            if (ix1>=0) LU->diag[i]-=(LU->offdiag[2][ix1])*(LU->offdiag[3][ix1]);
            if (ix>=0) LU->diag[i]-=(LU->offdiag[4][ix])*(LU->offdiag[5][ix]);
            if (ixyx>=0) LU->diag[i]-=(LU->offdiag[6][ixyx])*(LU->offdiag[7][ixyx]);
            if (ixy1>=0) LU->diag[i]-=(LU->offdiag[8][ixy1])*(LU->offdiag[9][ixy1]);
            if (ixy>=0) LU->diag[i]-=(LU->offdiag[10][ixy])*(LU->offdiag[11][ixy]);

            LU->diag[i]=1.0/(LU->diag[i]);

        } // end for (i=1; i<ngrid; ++i)

    }  // end if (nc == 1)

    else if (nc == 3) {

        // comput the first row
        fasp_smat_inv_nc3(LU->diag);
        memcpy(LU->offdiag[1],A->offdiag[1],9*sizeof(REAL));
        memcpy(LU->offdiag[5],A->offdiag[3],9*sizeof(REAL));

        for (i=1;i<ngrid;++i) {
            i1=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;
            ic=i*nc2;i1c=i1*nc2;ixc=ix*nc2;ix1c=ix1*nc2;ixyc=ixy*nc2;
            ixy1c=ixy1*nc2;ixyxc=ixyx*nc2;

            // comput alpha6[i-nxy]
            if (ixy>=0)
    fasp_blas_smat_mul_nc3(&(A->offdiag[4][ixyc]),&(LU->diag[ixyc]),&(LU->offdiag[10][ixyc]));

            // comput alpha5[ixy1]
            if (ixy1>=0) {
                for (j=0;j<9;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[1][ixyc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixy1c]),&(LU->offdiag[8][ixy1c]));
            }

            // comput alpha4[ixyx]
            if (ixyx>=0) {
                for (j=0;j<9;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[5][ixyc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[3][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixyxc]),&(LU->offdiag[6][ixyxc]));
            }

            // comput alpha3[ix]
            if (ix>=0) {

                memcpy(tc,&(A->offdiag[2][ixc]),9*sizeof(REAL));

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[7][ixyc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc3(tc,&(LU->diag[ixc]),&(LU->offdiag[4][ixc]));
            }

            // comput alpha2[i-nx+1]
            if (ix1>=0) {

                for (j=0;j<9;++j) tc[j]=0;

                if (ix>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[1][ixc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[7][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc3(tc,&(LU->diag[ix1c]),&(LU->offdiag[2][ix1c]));

            } // end if (ix1 >= 0)

            // comput alpha1[i-1]

            memcpy(tc,&(A->offdiag[0][i1c]),9*sizeof(REAL));

            if (ix>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[3][ixc]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,tc);
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[9][ixyc]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,tc);
            }

            fasp_blas_smat_mul_nc3(tc,&(LU->diag[i1c]),&(LU->offdiag[0][i1c]));

            // comput beta1[i]
            if (i+1<ngrid) {

                memcpy(&(LU->offdiag[1][ic]),&(A->offdiag[1][ic]),9*sizeof(REAL));

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]),&(LU->offdiag[5][ix1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[1][ic]));
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[11][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[1][ic]));
                }

            }

            // comput beta2[i]
            if (i+nline-1<ngrid) {

                {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]),&(LU->offdiag[5][i1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[3][ic]));
                }

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[9][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[3][ic]));
                }

            }

            // comput beta3[i]
            if (i+nline<ngrid) {

                memcpy(&(LU->offdiag[5][ic]),&(A->offdiag[3][ic]),9*sizeof(REAL));

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[11][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[5][ic]));
                }

            }

            // comput beta4[i]
            if (i+nplane-nline<ngrid) {

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]),&(LU->offdiag[9][ix1c]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[7][ic]));
                }

                if (ix>=0) {
                    fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[11][ixc]),smat);
                    fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[7][ic]));
                }

            }

            // comput beta5[i]
            if (i+nplane-1<ngrid) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]),&(LU->offdiag[11][i1c]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->offdiag[9][ic]));
            }

            // comput d[i]
            {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[0][i1c]),&(LU->offdiag[1][i1c]),smat);
                fasp_blas_darray_axpyz_nc3(-1,smat,&(A->diag[ic]),&(LU->diag[ic]));
            }

            if (ix1>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[2][ix1c]),&(LU->offdiag[3][ix1c]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ic]));
            }

            if (ix>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[4][ixc]),&(LU->offdiag[5][ixc]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ic]));
            }

            if (ixyx>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[7][ixyxc]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ic]));
            }

            if (ixy1>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[9][ixy1c]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ic]));
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc3(&(LU->offdiag[10][ixyc]),&(LU->offdiag[11][ixyc]),smat);
                fasp_blas_darray_axpy_nc3(-1,smat,&(LU->diag[ic]));
            }

            fasp_smat_inv_nc3(&(LU->diag[ic]));

        } // end for(i=1;i<ngrid;++i)

    }  // end if (nc == 3)

    else if (nc == 5) {
        // comput the first row
        // fasp_smat_inv_nc5(LU->diag);
        fasp_smat_inv(LU->diag,5);
        memcpy(LU->offdiag[1],A->offdiag[1], 25*sizeof(REAL));
        memcpy(LU->offdiag[5],A->offdiag[3], 25*sizeof(REAL));

        for(i=1;i<ngrid;++i) {
            i1=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;
            ic=i*nc2;i1c=i1*nc2;ixc=ix*nc2;ix1c=ix1*nc2;ixyc=ixy*nc2;ixy1c=ixy1*nc2;ixyxc=ixyx*nc2;

            // comput alpha6[i-nxy]
            if (ixy>=0)
      fasp_blas_smat_mul_nc5(&(A->offdiag[4][ixyc]),&(LU->diag[ixyc]),&(LU->offdiag[10][ixyc]));

            // comput alpha5[ixy1]
            if (ixy1>=0) {
                for (j=0;j<25;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]),&(LU->offdiag[1][ixyc]),smat);
                    fasp_blas_darray_axpy_nc5(-1.0,smat,tc);
                }

                fasp_blas_smat_mul_nc5(tc,&(LU->diag[ixy1c]),&(LU->offdiag[8][ixy1c]));
            }

            // comput alpha4[ixyx]
            if (ixyx>=0) {
                for (j=0;j<25;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]),&(LU->offdiag[5][ixyc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[3][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc5(tc,&(LU->diag[ixyxc]),&(LU->offdiag[6][ixyxc]));
            }

            // comput alpha3[ix]
            if (ix>=0) {

                memcpy(tc,&(A->offdiag[2][ixc]),25*sizeof(REAL));

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]),&(LU->offdiag[7][ixyc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc5(tc,&(LU->diag[ixc]),&(LU->offdiag[4][ixc]));
            }

            // comput alpha2[i-nx+1]
            if (ix1>=0) {

                for (j=0;j<25;++j) tc[j]=0;

                if (ix>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]),&(LU->offdiag[1][ixc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[7][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc5(tc,&(LU->diag[ix1c]),&(LU->offdiag[2][ix1c]));

            } // end if (ix1 >= 0)

            // comput alpha1[i-1]

            memcpy(tc,&(A->offdiag[0][i1c]), 25*sizeof(REAL));

            if (ix>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]),&(LU->offdiag[3][ixc]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,tc);
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]),&(LU->offdiag[9][ixyc]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,tc);
            }

            fasp_blas_smat_mul_nc5(tc,&(LU->diag[i1c]),&(LU->offdiag[0][i1c]));

            // comput beta1[i]
            if (i+1<ngrid) {

                memcpy(&(LU->offdiag[1][ic]),&(A->offdiag[1][ic]), 25*sizeof(REAL));

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]),&(LU->offdiag[5][ix1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[1][ic]));
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[11][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[1][ic]));
                }

            }

            // comput beta2[i]
            if (i+nline-1<ngrid) {

                {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[0][i1c]),&(LU->offdiag[5][i1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[3][ic]));
                }

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[9][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[3][ic]));
                }

            }

            // comput beta3[i]
            if (i+nline<ngrid) {

                memcpy(&(LU->offdiag[5][ic]),&(A->offdiag[3][ic]), 25*sizeof(REAL));

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[11][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[5][ic]));
                }

            }

            // comput beta4[i]
            if (i+nplane-nline<ngrid) {

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]),&(LU->offdiag[9][ix1c]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[7][ic]));
                }

                if (ix>=0) {
                    fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]),&(LU->offdiag[11][ixc]),smat);
                    fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[7][ic]));
                }

            }

            // comput beta5[i]
            if (i+nplane-1<ngrid) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[0][i1c]),&(LU->offdiag[11][i1c]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->offdiag[9][ic]));
            }

            // comput d[i]
            {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[0][i1c]),&(LU->offdiag[1][i1c]),smat);
                fasp_blas_darray_axpyz_nc5(-1,smat,&(A->diag[ic]),&(LU->diag[ic]));
            }

            if (ix1>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[2][ix1c]),&(LU->offdiag[3][ix1c]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->diag[ic]));
            }

            if (ix>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[4][ixc]),&(LU->offdiag[5][ixc]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->diag[ic]));
            }

            if (ixyx>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[7][ixyxc]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->diag[ic]));
            }

            if (ixy1>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[9][ixy1c]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->diag[ic]));
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc5(&(LU->offdiag[10][ixyc]),&(LU->offdiag[11][ixyc]),smat);
                fasp_blas_darray_axpy_nc5(-1,smat,&(LU->diag[ic]));
            }

            //fasp_smat_inv_nc5(&(LU->diag[ic]));
            fasp_smat_inv(&(LU->diag[ic]), 5);

        } // end for(i=1;i<ngrid;++i)

    }  // end if (nc == 5)

    else if (nc == 7) {
        // comput the first row
        //fasp_smat_inv_nc5(LU->diag);
        fasp_smat_inv(LU->diag,7);
        memcpy(LU->offdiag[1],A->offdiag[1], 49*sizeof(REAL));
        memcpy(LU->offdiag[5],A->offdiag[3], 49*sizeof(REAL));

        for(i=1;i<ngrid;++i) {
            i1=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;
            ic=i*nc2;i1c=i1*nc2;ixc=ix*nc2;ix1c=ix1*nc2;ixyc=ixy*nc2;ixy1c=ixy1*nc2;ixyxc=ixyx*nc2;

            // comput alpha6[i-nxy]
            if (ixy>=0)
fasp_blas_smat_mul_nc7(&(A->offdiag[4][ixyc]),&(LU->diag[ixyc]),&(LU->offdiag[10][ixyc]));

            // comput alpha5[ixy1]
            if (ixy1>=0) {
                for (j=0;j<49;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]),&(LU->offdiag[1][ixyc]),smat);
                    fasp_blas_darray_axpy_nc7(-1.0,smat,tc);
                }

                fasp_blas_smat_mul_nc7(tc,&(LU->diag[ixy1c]),&(LU->offdiag[8][ixy1c]));
            }

            // comput alpha4[ixyx]
            if (ixyx>=0) {
                for (j=0;j<49;++j) tc[j]=0;

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]),&(LU->offdiag[5][ixyc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[3][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc7(tc,&(LU->diag[ixyxc]),&(LU->offdiag[6][ixyxc]));
            }

            // comput alpha3[ix]
            if (ix>=0) {

                memcpy(tc,&(A->offdiag[2][ixc]),49*sizeof(REAL));

                if (ixy>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]),&(LU->offdiag[7][ixyc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc7(tc,&(LU->diag[ixc]),&(LU->offdiag[4][ixc]));
            }

            // comput alpha2[i-nx+1]
            if (ix1>=0) {

                for (j=0;j<49;++j) tc[j]=0;

                if (ix>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]),&(LU->offdiag[1][ixc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[7][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,tc);
                }

                fasp_blas_smat_mul_nc7(tc,&(LU->diag[ix1c]),&(LU->offdiag[2][ix1c]));

            } // end if (ix1 >= 0)

            // comput alpha1[i-1]

            memcpy(tc,&(A->offdiag[0][i1c]), 49*sizeof(REAL));

            if (ix>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]),&(LU->offdiag[3][ixc]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,tc);
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]),&(LU->offdiag[9][ixyc]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,tc);
            }

            fasp_blas_smat_mul_nc7(tc,&(LU->diag[i1c]),&(LU->offdiag[0][i1c]));

            // comput beta1[i]
            if (i+1<ngrid) {

                memcpy(&(LU->offdiag[1][ic]),&(A->offdiag[1][ic]), 49*sizeof(REAL));

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ix1c]),&(LU->offdiag[5][ix1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[1][ic]));
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[11][ixy1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[1][ic]));
                }

            }

            // comput beta2[i]
            if (i+nline-1<ngrid) {

                {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[0][i1c]),&(LU->offdiag[5][i1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[3][ic]));
                }

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[9][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[3][ic]));
                }

            }

            // comput beta3[i]
            if (i+nline<ngrid) {

                memcpy(&(LU->offdiag[5][ic]),&(A->offdiag[3][ic]), 49*sizeof(REAL));

                if (ixyx>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[11][ixyxc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[5][ic]));
                }

            }

            // comput beta4[i]
            if (i+nplane-nline<ngrid) {

                if (ix1>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ix1c]),&(LU->offdiag[9][ix1c]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[7][ic]));
                }

                if (ix>=0) {
                    fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]),&(LU->offdiag[11][ixc]),smat);
                    fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[7][ic]));
                }

            }

            // comput beta5[i]
            if (i+nplane-1<ngrid) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[0][i1c]),&(LU->offdiag[11][i1c]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->offdiag[9][ic]));
            }

            // comput d[i]
            {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[0][i1c]),&(LU->offdiag[1][i1c]),smat);
                fasp_blas_darray_axpyz_nc7(-1,smat,&(A->diag[ic]),&(LU->diag[ic]));
            }

            if (ix1>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[2][ix1c]),&(LU->offdiag[3][ix1c]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->diag[ic]));
            }

            if (ix>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[4][ixc]),&(LU->offdiag[5][ixc]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->diag[ic]));
            }

            if (ixyx>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[7][ixyxc]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->diag[ic]));
            }

            if (ixy1>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[9][ixy1c]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->diag[ic]));
            }

            if (ixy>=0) {
                fasp_blas_smat_mul_nc7(&(LU->offdiag[10][ixyc]),&(LU->offdiag[11][ixyc]),smat);
                fasp_blas_darray_axpy_nc7(-1,smat,&(LU->diag[ic]));
            }

            //fasp_smat_inv_nc5(&(LU->diag[ic]));
            fasp_smat_inv(&(LU->diag[ic]), 7);

        } // end for(i=1;i<ngrid;++i)

    }  // end if (nc == 7)

    else {
        // comput the first row
        fasp_smat_inv(LU->diag,nc);
        memcpy(LU->offdiag[1],A->offdiag[1],nc2*sizeof(REAL));
        memcpy(LU->offdiag[5],A->offdiag[3],nc2*sizeof(REAL));

        for(i=1;i<ngrid;++i) {

            i1=i-1;ix=i-nline;ixy=i-nplane;ix1=ix+1;ixyx=ixy+nline;ixy1=ixy+1;
            ic=i*nc2;i1c=i1*nc2;ixc=ix*nc2;ix1c=ix1*nc2;ixyc=ixy*nc2;ixy1c=ixy1*nc2;ixyxc=ixyx*nc2;
            // comput alpha6[i-nxy]
            if (ixy>=0)
                fasp_blas_smat_mul(&(A->offdiag[4][ixyc]),&(LU->diag[ixyc]),&(LU->offdiag[10][ixyc]),nc);

            // comput alpha5[ixy1]
            if (ixy1>=0) {
                for (j=0;j<nc2;++j) tc[j]=0;
                if (ixy>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]),&(LU->offdiag[1][ixyc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }

                fasp_blas_smat_mul(tc,&(LU->diag[ixy1c]),&(LU->offdiag[8][ixy1c]),nc);
            }

            // comput alpha4[ixyx]
            if (ixyx>=0) {
                for (j=0;j<nc2;++j) tc[j]=0;
                if (ixy>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]),&(LU->offdiag[5][ixyc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }
                if (ixy1>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[3][ixy1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }

                fasp_blas_smat_mul(tc,&(LU->diag[ixyxc]),&(LU->offdiag[6][ixyxc]),nc);
            }

            // comput alpha3[ix]
            if (ix>=0) {

                memcpy(tc,&(A->offdiag[2][ixc]),nc2*sizeof(REAL));
                if (ixy>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]),&(LU->offdiag[7][ixyc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }

                fasp_blas_smat_mul(tc,&(LU->diag[ixc]),&(LU->offdiag[4][ixc]),nc);
            }

            // comput alpha2[i-nx+1]
            if (ix1>=0) {

                for (j=0;j<nc2;++j) tc[j]=0;

                if (ix>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[4][ixc]),&(LU->offdiag[1][ixc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }

                if (ixy1>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[7][ixy1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,tc);
                }

                fasp_blas_smat_mul(tc,&(LU->diag[ix1c]),&(LU->offdiag[2][ix1c]),nc);
            }

            // comput alpha1[i-1]

            memcpy(tc,&(A->offdiag[0][i1c]),nc2*sizeof(REAL));
            if (ix>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[4][ixc]),&(LU->offdiag[3][ixc]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,tc);
            }
            if (ixy>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]),&(LU->offdiag[9][ixyc]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,tc);
            }

            fasp_blas_smat_mul(tc,&(LU->diag[i1c]),&(LU->offdiag[0][i1c]),nc);

            // comput beta1[i]
            if (i+1<ngrid) {

                memcpy(&(LU->offdiag[1][ic]),&(A->offdiag[1][ic]),nc2*sizeof(REAL));
                if (ix1>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[2][ix1c]),&(LU->offdiag[5][ix1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[1][ic]));
                }
                if (ixy1>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[11][ixy1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[1][ic]));
                }

            }

            // comput beta2[i]
            if (i+nline-1<ngrid) {

                {
                    fasp_blas_smat_mul(&(LU->offdiag[0][i1c]),&(LU->offdiag[5][i1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[3][ic]));
                }

                if (ixyx>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[9][ixyxc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[3][ic]));
                }

            }

            // comput beta3[i]
            if (i+nline<ngrid) {

                memcpy(&(LU->offdiag[5][ic]),&(A->offdiag[3][ic]),nc2*sizeof(REAL));
                if (ixyx>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[11][ixyxc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[5][ic]));
                }

            }

            // comput beta4[i]
            if (i+nplane-nline<ngrid) {

                if (ix1>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[2][ix1c]),&(LU->offdiag[9][ix1c]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[7][ic]));
                }

                if (ix>=0) {
                    fasp_blas_smat_mul(&(LU->offdiag[4][ixc]),&(LU->offdiag[11][ixc]),smat,nc);
                    fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[7][ic]));
                }
            }

            // comput beta5[i]
            if (i+nplane-1<ngrid) {
                fasp_blas_smat_mul(&(LU->offdiag[0][i1c]),&(LU->offdiag[11][i1c]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->offdiag[9][ic]));
            }

            // comput d[i]
            {
                fasp_blas_smat_mul(&(LU->offdiag[0][i1c]),&(LU->offdiag[1][i1c]),smat,nc);
                fasp_blas_darray_axpyz(nc2,-1,smat,&(A->diag[ic]),&(LU->diag[ic]));
            }
            if (ix1>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[2][ix1c]),&(LU->offdiag[3][ix1c]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ic]));
            }

            if (ix>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[4][ixc]),&(LU->offdiag[5][ixc]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ic]));
            }

            if (ixyx>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[6][ixyxc]),&(LU->offdiag[7][ixyxc]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ic]));
            }


            if (ixy1>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[8][ixy1c]),&(LU->offdiag[9][ixy1c]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ic]));
            }

            if (ixy>=0) {
                fasp_blas_smat_mul(&(LU->offdiag[10][ixyc]),&(LU->offdiag[11][ixyc]),smat,nc);
                fasp_blas_darray_axpy(nc2,-1,smat,&(LU->diag[ic]));
            }

            fasp_smat_inv(&(LU->diag[ic]),nc);

        }

    } // end else

    fasp_mem_free(smat); smat = NULL;
    fasp_mem_free(tc);   tc   = NULL;

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 ∗filemat, const char ∗filerhs, 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 ∗filemat, 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

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

filename	File name for A
A	Pointer to the dBSRmat matrix A

Author

Chensong Zhang

Date

01/07/2021

Definition at line 1292 of file BlaIO.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

filename	File name for matrix A
A	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 BlaIO.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

filename	File name for A
A	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

filename	Name of file to write to
A	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

A	Pointer to the dCOOmat matrix A

Author

Ziteng Wang

Date

12/24/2012

Definition at line 1545 of file BlaIO.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

filename	File name for matrix
A	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 BlaIO.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

filename	File name for matrix
A	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

filename	File name for matrix
A	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

A	pointer to the dCSRmat matrix
filename	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 BlaIO.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

fni	File name for matrix i-index
fnj	File name for matrix j-index
fna	File name for matrix values
fnb	File name for vector values
A	Pointer to the CSR matrix
b	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 BlaIO.c.

9.59.2.11 fasp_dcsr_print()

```
void fasp_dcsr_print (
             const dCSRmat * A )
```

Print out a dCSRmat matrix in coordinate format.

Parameters

A	Pointer to the dCSRmat matrix A

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1523 of file BlaIO.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

filename	Char for matrix file name
A	Pointer to the CSR matrix

Author

Ziteng Wang

Date

12/25/2012

Definition at line 252 of file BlaIO.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

filename	Name of file to write to
A	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 BlaIO.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

filename	Name of file to write to
A	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

filename	File name
A	Pointer to the CSR matrix
b	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

filemat	File name for matrix
filerhs	File name for right-hand side
A	Pointer to the dCSR matrix
b	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 BlaIO.c.

9.59.2.17 fasp_dcsrvec_write1()

```
void fasp_dcsrvec_write1 (
             const char * filename,
             dCSRmat * A,
             dvector * b )
```

Write A and b to a SINGLE disk file.

Parameters

filename	File name
A	Pointer to the CSR matrix
b	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

filemat	File name for matrix
filerhs	File name for right-hand side
A	Pointer to the dCSR matrix
b	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 BlaIO.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

filename	File name for matrix
A	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 BlaIO.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

filename	File name for matrix
A	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 BlaIO.c.

9.59.2.21 fasp_dstr_print()

```
void fasp_dstr_print (
            const dSTRmat * A )
```

Print out a dSTRmat matrix in coordinate format.

Parameters

A	Pointer to the dSTRmat matrix A

Author

Ziteng Wang

Date

12/24/2012

Definition at line 1701 of file BlaIO.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

filename	File name for the matrix
A	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 BlaIO.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

filename	File name for A
A	Pointer to the dSTRmat matrix A

Note

The 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 BlaIO.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

n	An interger (if n=0, then print all entries)
u	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

filename	File name for vector b
b	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

vec	Pointer to the dvector
filename	File name

Author

Xuehai Huang

Date

03/29/2009

Definition at line 1388 of file BlaIO.c.

9.59.2.27 fasp_dvecind_read()

```
void fasp_dvecind_read (
            const char * filename,
            dvector * b )
```

Read b from matrix disk file.

Parameters

filename	File name for vector b
b	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 BlaIO.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

vec	Pointer to the dvector
filename	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

input_file	File name of vector file
A	Pointer to the matrix
b	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,
            ivector * u )
```

Print first n entries of a vector of INT type.

Parameters

n	An interger (if n=0, then print all entries)
u	Pointer to an ivector

Author

Chensong Zhang

Date

03/29/2009

Definition at line 1503 of file BlaIO.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

filename	File name for vector b
b	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 BlaIO.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

vec	Pointer to the dvector
filename	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

filename	File name for vector b
b	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

filename	File name of matrix file
A	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

filename	File name of matrix file
A	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 BlaIO.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

filename	File name of matrix file
A	Pointer to the matrix
flag	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 BlaIO.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

filerhs	File name of vector file
b	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

filerhs	File name of vector file
b	Pointer to the vector
flag	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.
```

#include "fasp.h"
#include "fasp_functs.h"
#include "hb_io.h"

// Flags which indicates lengths of INT and REAL numbers
int ilength;
int dlength;
/*----------------------------------*/
/*--  Declare Private Functions  --*/
/*----------------------------------*/

#include "BlaIOUtil.inl"

/*----------------------------------*/
/*--      Public Functions       --*/
```

```
/*----------------------------------*/

void fasp_dcsrvec_read1(const char* filename, dCSRmat* A, dvector* b)
{
    int  i, m, n, idata;
    REAL ddata;

    // Open input disk file
    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    // Read CSR matrix
    if (fscanf(fp, "%d %d", &m, &n) > 0) {
        A->row = m;
        A->col = n;
    } else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    A->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    for (i = 0; i <= m; ++i) {
        if (fscanf(fp, "%d", &idata) > 0)
            A->IA[i] = idata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    INT nnz = A->IA[m] - A->IA[0];

    A->nnz = nnz;
    A->JA  = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    A->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));

    for (i = 0; i < nnz; ++i) {
        if (fscanf(fp, "%d", &idata) > 0)
            A->JA[i] = idata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    for (i = 0; i < nnz; ++i) {
        if (fscanf(fp, "%lf", &ddata) > 0)
            A->val[i] = ddata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    // Read RHS vector
    if (fscanf(fp, "%d", &m) > 0) b->row = m;

    b->val = (REAL*)fasp_mem_calloc(m, sizeof(REAL));

    for (i = 0; i < m; ++i) {
        if (fscanf(fp, "%lf", &ddata) > 0)
            b->val[i] = ddata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);
}

void fasp_dcsrvec_read2(const char* filemat, const char* filerhs, dCSRmat* A,
                        dvector* b)
{
    int i, n, tempi;

    /* read the matrix from file */
    FILE* fp = fopen(filemat, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filemat);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filemat);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d\n", &n) > 0) {
        A->row = n;
        A->col = n;
        A->IA  = (INT*)fasp_mem_calloc(n + 1, sizeof(INT));
    } else {
        fasp_chkerr(ERROR_WRONG_FILE, filemat);
    }

    for (i = 0; i <= n; ++i) {
        if (fscanf(fp, "%d\n", &tempi) > 0)
            A->IA[i] = tempi - 1;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filemat);
        }
    }

    INT nz = A->IA[n];
    A->nnz = nz;
    A->JA  = (INT*)fasp_mem_calloc(nz, sizeof(INT));
    A->val = (REAL*)fasp_mem_calloc(nz, sizeof(REAL));

    for (i = 0; i < nz; ++i) {
        if (fscanf(fp, "%d\n", &tempi) > 0)
            A->JA[i] = tempi - 1;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filemat);
        }
    }

    for (i = 0; i < nz; ++i) {
        if (fscanf(fp, "%le\n", &(A->val[i])) <= 0) {
            fasp_chkerr(ERROR_WRONG_FILE, filemat);
        }
    }

    fclose(fp);

    /* Read the rhs from file */
    b->row = n;
    b->val = (REAL*)fasp_mem_calloc(n, sizeof(REAL));

    fp = fopen(filerhs, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filerhs);

    if (fscanf(fp, "%d\n", &n) < 0) fasp_chkerr(ERROR_WRONG_FILE, filerhs);

    if (n != b->row) {
        printf("### WARNING: rhs size = %d, matrix size = %d!\n", n, b->row);
        fasp_chkerr(ERROR_MAT_SIZE, filemat);
    }

    for (i = 0; i < n; ++i) {
        if (fscanf(fp, "%le\n", &(b->val[i])) <= 0) {
            fasp_chkerr(ERROR_WRONG_FILE, filerhs);
        }
    }

    fclose(fp);
}

void fasp_dcsr_read(const char* filename, dCSRmat* A)
{
    int  i, m, idata;
    REAL ddata;

    // Open input disk file
    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    // Read CSR matrix
    if (fscanf(fp, "%d", &m) > 0)
        A->row = A->col = m;
    else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    A->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    for (i = 0; i <= m; ++i) {
        if (fscanf(fp, "%d", &idata) > 0)
            A->IA[i] = idata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    // If IA starts from 1, shift by -1
    if (A->IA[0] == 1)
        for (i = 0; i <= m; ++i) A->IA[i]--;

    INT nnz = A->IA[m] - A->IA[0];

    A->nnz = nnz;
    A->JA  = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    A->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));

    for (i = 0; i < nnz; ++i) {
        if (fscanf(fp, "%d", &idata) > 0)
            A->JA[i] = idata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    // If JA starts from 1, shift by -1
    if (A->JA[0] == 1)
        for (i = 0; i < nnz; ++i) A->JA[i]--;

    for (i = 0; i < nnz; ++i) {
        if (fscanf(fp, "%lf", &ddata) > 0)
            A->val[i] = ddata;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);
}

void fasp_dcoo_read(const char* filename, dCSRmat* A)
{
    int  i, j, k, m, n, nnz;
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);

    for (k = 0; k < nnz; k++) {
        if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
            Atmp.rowind[k] = i;
            Atmp.colind[k] = j;
            Atmp.val[k]    = value;
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dcoo_read1(const char* filename, dCSRmat* A)
{
    int  i, j, k, m, n, nnz;
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);

    for (k = 0; k < nnz; k++) {
        if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
            Atmp.rowind[k] = i - 1;
            Atmp.colind[k] = j - 1;
            Atmp.val[k]    = value;
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dcoovec_bin_read(const char* fni, const char* fnj, const char* fna,
                           const char* fnb, dCSRmat* A, dvector* b)
{
    size_t n, type, nnz, i;
    FILE*  fp;

    fp = fopen(fnb, "rb");
    if (fp == NULL) {
        fasp_chkerr(ERROR_WRONG_FILE, fnb);
    }
    printf("%s:  reading file %s ...\n", __FUNCTION__, fnb);
    fread(&n, sizeof(size_t), 1, fp);
    b->row = n;
    b->val = (double*)fasp_mem_calloc(n, sizeof(double));
    fread(b->val, sizeof(double), n, fp);
    fclose(fp);

    fp = fopen(fni, "rb");
    if (fp == NULL) {
        fasp_chkerr(ERROR_WRONG_FILE, fni);
    }
    printf("%s:  reading file %s ...\n", __FUNCTION__, fni);
    fread(&type, sizeof(size_t), 1, fp);
    fread(&nnz, sizeof(size_t), 1, fp);
    dCOOmat Atmp = fasp_dcoo_create(n, n, nnz);
    Atmp.rowind  = (int*)fasp_mem_calloc(nnz, sizeof(int));
    fread(Atmp.rowind, sizeof(int), nnz, fp);
    for (i = 0; i < nnz; i++) Atmp.rowind[i] = Atmp.rowind[i] - 1;
    fclose(fp);

    fp = fopen(fnj, "rb");
    if (fp == NULL) {
        fasp_chkerr(ERROR_WRONG_FILE, fnj);
    }
    printf("%s:  reading file %s ...\n", __FUNCTION__, fnj);
    fread(&type, sizeof(size_t), 1, fp);
    fread(&nnz, sizeof(size_t), 1, fp);
    Atmp.colind = (int*)fasp_mem_calloc(nnz, sizeof(int));
    fread(Atmp.colind, sizeof(int), nnz, fp);
    for (i = 0; i < nnz; i++) Atmp.colind[i] = Atmp.colind[i] - 1;
    fclose(fp);

    fp = fopen(fna, "rb");
    if (fp == NULL) {
        fasp_chkerr(ERROR_WRONG_FILE, fna);
    }
    printf("%s:  reading file %s ...\n", __FUNCTION__, fna);
    fread(&type, sizeof(size_t), 1, fp);
    fread(&nnz, sizeof(size_t), 1, fp);
    Atmp.val = (double*)fasp_mem_calloc(nnz, sizeof(double));
    fread(Atmp.val, sizeof(double), nnz, fp);
    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dcoo_shift_read(const char* filename, dCSRmat* A)
{
    int  i, j, k, m, n, nnz;
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);

    for (k = 0; k < nnz; k++) {
        if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
            Atmp.rowind[k] = i - 1;
            Atmp.colind[k] = j - 1;
            Atmp.val[k]    = value;
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dmtx_read(const char* filename, dCSRmat* A)
{
    int  i, j, m, n, nnz;
    INT  innz; // index of nonzeros
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);

    innz = 0;

    while (innz < nnz) {
        if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {
            Atmp.rowind[innz] = i - 1;
            Atmp.colind[innz] = j - 1;
            Atmp.val[innz]    = value;
            innz              = innz + 1;
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dmtxsym_read(const char* filename, dCSRmat* A)
{
    int  i, j, m, n, nnz;
    int  innz; // index of nonzeros
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    if (fscanf(fp, "%d %d %d", &m, &n, &nnz) <= 0) {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    nnz          = 2 * (nnz - m) + m; // adjust for sym problem
    dCOOmat Atmp = fasp_dcoo_create(m, n, nnz);

    innz = 0;

    while (innz < nnz) {
        if (fscanf(fp, "%d %d %le", &i, &j, &value) != EOF) {

            if (i == j) {
                Atmp.rowind[innz] = i - 1;
                Atmp.colind[innz] = j - 1;
                Atmp.val[innz]    = value;
                innz              = innz + 1;
            } else {
                Atmp.rowind[innz]     = i - 1;
                Atmp.rowind[innz + 1] = j - 1;
                Atmp.colind[innz]     = j - 1;
                Atmp.colind[innz + 1] = i - 1;
                Atmp.val[innz]        = value;
                Atmp.val[innz + 1]    = value;
                innz                  = innz + 2;
            }
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    fclose(fp);

    fasp_format_dcoo_dcsr(&Atmp, A);
    fasp_dcoo_free(&Atmp);
}

void fasp_dstr_read(const char* filename, dSTRmat* A)
{
    int  nx, ny, nz, nxy, ngrid, nband, nc, offset;
    int  i, k, n;
    REAL value;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    // read dimension of the problem
    if (fscanf(fp, "%d %d %d", &nx, &ny, &nz) > 0) {
        A->nx = nx;
        A->ny = ny;
        A->nz = nz;
    } else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    nxy      = nx * ny;
    ngrid    = nxy * nz;
    A->nxy   = nxy;
    A->ngrid = ngrid;

    // read number of components
    if (fscanf(fp, "%d", &nc) > 0)
        A->nc = nc;
    else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    // read number of bands
    if (fscanf(fp, "%d", &nband) > 0)
        A->nband = nband;
    else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    A->offsets = (INT*)fasp_mem_calloc(nband, sizeof(INT));

    // read diagonal
    if (fscanf(fp, "%d", &n) > 0) {
        A->diag = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
    } else {
        fasp_chkerr(ERROR_WRONG_FILE, filename);
    }

    for (i = 0; i < n; ++i) {
        if (fscanf(fp, "%le", &value) > 0)
            A->diag[i] = value;
        else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }
    }

    // read offdiags
    k          = nband;
    A->offdiag = (REAL**)fasp_mem_calloc(nband, sizeof(REAL*));
    while (k--) {
        // read number band k
        if (fscanf(fp, "%d %d", &offset, &n) > 0) {
            A->offsets[nband - k - 1] = offset;
        } else {
            fasp_chkerr(ERROR_WRONG_FILE, filename);
        }

        A->offdiag[nband - k - 1] = (REAL*)fasp_mem_calloc(n, sizeof(REAL));
        for (i = 0; i < n; ++i) {
            if (fscanf(fp, "%le", &value) > 0) {
                A->offdiag[nband - k - 1][i] = value;
            } else {
                fasp_chkerr(ERROR_WRONG_FILE, filename);
            }
        }
    }

    fclose(fp);
}

void fasp_dbsr_read(const char* filename, dBSRmat* A)
{
    int    ROW, COL, NNZ, nb, storage_manner;
    int    i, n;
    int    index;
    REAL   value;
    size_t status;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    status = fscanf(fp, "%d %d %d", &ROW, &COL, &NNZ); // dimensions of the problem
    fasp_chkerr(status, filename);
    A->ROW = ROW;
    A->COL = COL;
    A->NNZ = NNZ;

    status = fscanf(fp, "%d", &nb); // read the size of each block
    fasp_chkerr(status, filename);
    A->nb = nb;

    status = fscanf(fp, "%d", &storage_manner); // read the storage_manner
    fasp_chkerr(status, filename);
    A->storage_manner = storage_manner;

    // allocate memory space
    fasp_dbsr_alloc(ROW, COL, NNZ, nb, storage_manner, A);

    // read IA
    status = fscanf(fp, "%d", &n);
    fasp_chkerr(status, filename);
    for (i = 0; i < n; ++i) {
        status = fscanf(fp, "%d", &index);
        fasp_chkerr(status, filename);
        A->IA[i] = index;
    }

    // read JA
    status = fscanf(fp, "%d", &n);
    fasp_chkerr(status, filename);
    for (i = 0; i < n; ++i) {
        status = fscanf(fp, "%d", &index);
        fasp_chkerr(status, filename);
        A->JA[i] = index;
    }

    // read val
    status = fscanf(fp, "%d", &n);
    fasp_chkerr(status, filename);
    for (i = 0; i < n; ++i) {
        status = fscanf(fp, "%le", &value);
        fasp_chkerr(status, filename);
        A->val[i] = value;
    }

    fclose(fp);
}

void fasp_dvecind_read(const char* filename, dvector* b)
{
    int    i, n, index;
    REAL   value;
    size_t status;

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    status = fscanf(fp, "%d", &n);
    fasp_dvec_alloc(n, b);

    for (i = 0; i < n; ++i) {

        status = fscanf(fp, "%d %le", &index, &value);

        if (value > BIGREAL || index >= n) {
            fasp_dvec_free(b);
            fclose(fp);

            printf("### ERROR: Wrong index = %d or value = %lf\n", index, value);
            fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
        }

        b->val[index] = value;
    }

    fclose(fp);
    fasp_chkerr(status, filename);
}

void fasp_dvec_read(const char* filename, dvector* b)
{
    int    i, n;
    REAL   value;
    size_t status;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    status = fscanf(fp, "%d", &n);

    fasp_dvec_alloc(n, b);

    for (i = 0; i < n; ++i) {

        status    = fscanf(fp, "%le", &value);
        b->val[i] = value;

        if (value > BIGREAL) {
            fasp_dvec_free(b);
            fclose(fp);

            printf("### ERROR: Wrong value = %lf!\n", value);
            fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
        }
    }

    fclose(fp);
    fasp_chkerr(status, filename);
}

void fasp_ivecind_read(const char* filename, ivector* b)
{
    int    i, n, index, value;
    size_t status;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    status = fscanf(fp, "%d", &n);
    fasp_ivec_alloc(n, b);

    for (i = 0; i < n; ++i) {
        status        = fscanf(fp, "%d %d", &index, &value);
        b->val[index] = value;
    }

    fclose(fp);
    fasp_chkerr(status, filename);
}

void fasp_ivec_read(const char* filename, ivector* b)
{
    int    i, n, value;
    size_t status;

    FILE* fp = fopen(filename, "r");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    skip_comments(fp); // skip the comments in the beginning --zcs 06/30/2020

    status = fscanf(fp, "%d", &n);
    fasp_ivec_alloc(n, b);

    for (i = 0; i < n; ++i) {
        status    = fscanf(fp, "%d", &value);
        b->val[i] = value;
    }

    fclose(fp);
    fasp_chkerr(status, filename);
```

```
}

void fasp_dcsrvec_write1(const char* filename, dCSRmat* A, dvector* b)
{
    INT m = A->row, n = A->col, nnz = A->nnz;
    INT i;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    /* write the matrix to file */
    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d %d\n", m, n);
    for (i = 0; i < m + 1; ++i) {
        fprintf(fp, "%d\n", A->IA[i]);
    }
    for (i = 0; i < nnz; ++i) {
        fprintf(fp, "%d\n", A->JA[i]);
    }
    for (i = 0; i < nnz; ++i) {
        fprintf(fp, "%le\n", A->val[i]);
    }

    m = b->row;

    /* write the rhs to file */
    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d\n", m);

    for (i = 0; i < m; ++i) fprintf(fp, "%le\n", b->val[i]);

    fclose(fp);
}

void fasp_dcsrvec_write2(const char* filemat, const char* filerhs, dCSRmat* A,
                         dvector* b)
{
    INT m = A->row, nnz = A->nnz;
    INT i;

    FILE* fp = fopen(filemat, "w");

    /* write the matrix to file */
    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filemat);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filemat);

    fprintf(fp, "%d\n", m);
    for (i = 0; i < m + 1; ++i) {
        fprintf(fp, "%d\n", A->IA[i] + 1);
    }
    for (i = 0; i < nnz; ++i) {
        fprintf(fp, "%d\n", A->JA[i] + 1);
    }
    for (i = 0; i < nnz; ++i) {
        fprintf(fp, "%le\n", A->val[i]);
    }

    fclose(fp);

    m = b->row;

    fp = fopen(filerhs, "w");

    /* write the rhs to file */
    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filerhs);

    fprintf(fp, "%d\n", m);

    for (i = 0; i < m; ++i) fprintf(fp, "%le\n", b->val[i]);

    fclose(fp);
}

void fasp_dcoo_write(const char* filename, dCSRmat* A)
{
    const INT m = A->row, n = A->col;
    INT       i, j;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d  %d  %d\n", m, n, A->nnz);
    for (i = 0; i < m; ++i) {
        for (j = A->IA[i]; j < A->IA[i + 1]; j++)
            fprintf(fp, "%d  %d  %0.15e\n", i, A->JA[j], A->val[j]);
    }

    fclose(fp);
}

void fasp_dstr_write(const char* filename, dSTRmat* A)
{
    const INT nx = A->nx, ny = A->ny, nz = A->nz;
    const INT ngrid = A->ngrid, nband = A->nband, nc = A->nc;

    INT* offsets = A->offsets;

    INT i, k, n;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d  %d  %d\n", nx, ny, nz); // write dimension of the problem

    fprintf(fp, "%d\n", nc); // read number of components

    fprintf(fp, "%d\n", nband); // write number of bands

    // write diagonal
    n = ngrid * nc * nc;    // number of nonzeros in each band
    fprintf(fp, "%d\n", n); // number of diagonal entries
    for (i = 0; i < n; ++i) fprintf(fp, "%le\n", A->diag[i]);

    // write offdiags
    k = nband;
    while (k--) {
        INT offset = offsets[nband - k - 1];
        n          = (ngrid - ABS(offset)) * nc * nc; // number of nonzeros in each band
        fprintf(fp, "%d  %d\n", offset, n);           // read number band k
        for (i = 0; i < n; ++i) {
            fprintf(fp, "%le\n", A->offdiag[nband - k - 1][i]);
        }
    }

    fclose(fp);
}

void fasp_dbsr_print(const char* filename, dBSRmat* A)
{
    const INT ROW = A->ROW;
    const INT nb  = A->nb;
    const INT nb2 = nb * nb;

    INT*  ia  = A->IA;
    INT*  ja  = A->JA;
    REAL* val = A->val;

    INT i, j, k, ind;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  printing to file %s ...\n", __FUNCTION__, filename);

    for (i = 0; i < ROW; i++) {
        for (k = ia[i]; k < ia[i + 1]; k++) {
            j = ja[k];
            fprintf(fp, "A[%d,%d]=\n", i, j);
            for (ind = 0; ind < nb2; ind++) {
                fprintf(fp, "%+.10E  ", val[k * nb2 + ind]);
            }
            fprintf(fp, "\n");
        }
    }
}

void fasp_dbsr_write(const char* filename, dBSRmat* A)
{
    const INT ROW = A->ROW, COL = A->COL, NNZ = A->NNZ;
    const INT nb = A->nb, storage_manner = A->storage_manner;

    INT*  ia  = A->IA;
    INT*  ja  = A->JA;
    REAL* val = A->val;

    INT i, n;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d  %d  %d\n", ROW, COL, NNZ); // write dimension of the block matrix

    fprintf(fp, "%d\n", nb); // write the size of each block

    fprintf(fp, "%d\n", storage_manner); // write storage manner of each block

    // write A->IA
    n = ROW + 1;            // length of A->IA
    fprintf(fp, "%d\n", n); // length of A->IA
    for (i = 0; i < n; ++i) fprintf(fp, "%d\n", ia[i]);

    // write A->JA
    n = NNZ;                // length of A->JA
    fprintf(fp, "%d\n", n); // length of A->JA
    for (i = 0; i < n; ++i) fprintf(fp, "%d\n", ja[i]);

    // write A->val
    n = NNZ * nb * nb;      // length of A->val
    fprintf(fp, "%d\n", n); // length of A->val
    for (i = 0; i < n; ++i) fprintf(fp, "%le\n", val[i]);

    fclose(fp);
}

void fasp_dvec_write(const char* filename, dvector* vec)
{
    INT m = vec->row, i;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d\n", m);

    for (i = 0; i < m; ++i) fprintf(fp, "%0.15e\n", vec->val[i]);

    fclose(fp);
}

void fasp_dvecind_write(const char* filename, dvector* vec)
{
    INT m = vec->row, i;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    fprintf(fp, "%d\n", m);

    for (i = 0; i < m; ++i) fprintf(fp, "%d %le\n", i, vec->val[i]);

    fclose(fp);
}
```

```

void fasp_ivec_write(const char* filename, ivector* vec)
{
    INT m = vec->row, i;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    // write number of nonzeros
    fprintf(fp, "%d\n", m);

    // write index and value each line
    for (i = 0; i < m; ++i) fprintf(fp, "%d %d\n", i, vec->val[i] + 1);

    fclose(fp);
}

void fasp_dvec_print(const INT n, dvector* u)
{
    INT i;
    INT NumPrint = n;

    if (n <= 0) NumPrint = u->row; // print all

    for (i = 0; i < NumPrint; ++i) printf("vec_%d = %15.10E\n", i, u->val[i]);
}

void fasp_ivec_print(const INT n, ivector* u)
{
    INT i;
    INT NumPrint = n;

    if (n <= 0) NumPrint = u->row; // print all

    for (i = 0; i < NumPrint; ++i) printf("vec_%d = %d\n", i, u->val[i]);
}

void fasp_dcsr_print(const dCSRmat* A)
{
    const INT m = A->row, n = A->col;
    INT       i, j;

    printf("nrow = %d, ncol = %d, nnz = %d\n", m, n, A->nnz);
    for (i = 0; i < m; ++i) {
        for (j = A->IA[i]; j < A->IA[i + 1]; j++)
            printf("A_(%d,%d) = %+.10E\n", i, A->JA[j], A->val[j]);
    }
}

void fasp_dcoo_print(const dCOOmat* A)
{
    INT k;

    printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
    for (k = 0; k < A->nnz; k++) {
        printf("A_(%d,%d) = %+.10E\n", A->rowind[k], A->colind[k], A->val[k]);
    }
}

void fasp_dbsr_write_coo(const char* filename, const dBSRmat* A)
{

    INT i, j, k, l;
    INT nb, nb2;
    nb  = A->nb;
    nb2 = nb * nb;

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

#if DEBUG_MODE > PRINT_MIN
    printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d, nb = %d\n", A->ROW, A->COL,
           A->NNZ, A->nb);
    printf("### DEBUG: storage_manner = %d\n", A->storage_manner);
#endif

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    // write dimension of the block matrix
    fprintf(fp, "%% dimension of the block matrix and nonzeros %d  %d  %d\n", A->ROW,
            A->COL, A->NNZ);
    // write the size of each block
    fprintf(fp, "%% the size of each block %d\n", A->nb);
    // write storage manner of each block
    fprintf(fp, "%% storage manner of each block %d\n", A->storage_manner);

    for (i = 0; i < A->ROW; i++) {
        for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
            for (k = 0; k < A->nb; k++) {
                for (l = 0; l < A->nb; l++) {
                    fprintf(fp, "%d %d %+.10E\n", i * nb + k + 1, A->JA[j] * nb + l + 1,
                            A->val[j * nb2 + k * nb + l]);
                }
            }
        }
    }

    fclose(fp);
}

void fasp_dcsr_write_coo(const char* filename, const dCSRmat* A)
{

    INT i, j;

#if DEBUG_MODE > PRINT_MIN
    printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
#endif

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    // write dimension of the matrix
    fprintf(fp, "%% dimension of the matrix and nonzeros %d  %d  %d\n", A->row, A->col,
            A->nnz);

    for (i = 0; i < A->row; i++) {
        for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
            fprintf(fp, "%d %d %+.15E\n", i + 1, A->JA[j] + 1, A->val[j]);
        }
    }

    fclose(fp);
}

void fasp_dcsr_write_mtx(const char* filename, const dCSRmat* A)
{
    INT i, j;

#if DEBUG_MODE > PRINT_MIN
    printf("nrow = %d, ncol = %d, nnz = %d\n", A->row, A->col, A->nnz);
#endif

    FILE* fp = fopen(filename, "w");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    // write dimension of the matrix
    fprintf(fp, "%% MatrixMarket matrix coordinate general\n");
    fprintf(fp, "%d  %d  %d\n", A->row, A->col, A->nnz);

    for (i = 0; i < A->row; i++) {
        for (j = A->IA[i]; j < A->IA[i + 1]; j++) {
            fprintf(fp, "%d %d %+.15E\n", i + 1, A->JA[j] + 1, A->val[j]);
        }
    }

    fclose(fp);
}

void fasp_dstr_print(const dSTRmat* A)
{
    // TODO: To be added later!  --Chensong
```

```
}

void fasp_matrix_read(const char* filename, void* A)
{

    int    index, flag;
    SHORT  EndianFlag;
    size_t status;

    FILE* fp = fopen(filename, "rb");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    status = fread(&index, sizeof(INT), 1, fp);
    fasp_chkerr(status, filename);

    // matrix stored in ASCII format
    if (index == 808464432) {

        fclose(fp);
        fp = fopen(filename, "r"); // reopen file of reading file in ASCII

        status = fscanf(fp, "%d\n", &flag); // jump over the first line
        fasp_chkerr(status, __FUNCTION__);

        status = fscanf(fp, "%d\n", &flag); // reading the format information
        fasp_chkerr(status, __FUNCTION__);

        flag = (INT)flag / 100;

        switch (flag) {
            case 0:
                fasp_dcsr_read_s(fp, (dCSRmat*)A);
                break;
            case 1:
                fasp_dcoo_read_s(fp, (dCSRmat*)A);
                break;
            case 2:
                fasp_dbsr_read_s(fp, (dBSRmat*)A);
                break;
            case 3:
                fasp_dstr_read_s(fp, (dSTRmat*)A);
                break;
            case 4:
                fasp_dcoo_read_s(fp, (dCSRmat*)A);
                break;
            case 5:
                fasp_dmtx_read_s(fp, (dCSRmat*)A);
                break;
            case 6:
                fasp_dmtxsym_read_s(fp, (dCSRmat*)A);
                break;
            default:
                printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
                fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
        }

        fclose(fp);
        return;
    }

    // matrix stored in binary format

    // test Endian consistence of machine and file
    EndianFlag = index;

    status = fread(&index, sizeof(INT), 1, fp);
    fasp_chkerr(status, filename);

    index   = endian_convert_int(index, sizeof(INT), EndianFlag);
    flag    = (INT)index / 100;
    ilength = (INT)(index - flag * 100) / 10;
    dlength = index % 10;

    switch (flag) {
        case 1:
            fasp_dcsr_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 2:
            fasp_dbsr_read_b(fp, (dBSRmat*)A, EndianFlag);
            break;
        case 3:
            fasp_dstr_read_b(fp, (dSTRmat*)A, EndianFlag);
            break;
        case 4:
            fasp_dcoo_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 5:
            fasp_dmtx_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 6:
            fasp_dmtxsym_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        default:
            printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
            fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
    }

    fclose(fp);
}

void fasp_matrix_read_bin(const char* filename, void* A)
{
    int    index, flag;
    SHORT  EndianFlag = 1;
    size_t status;

    FILE* fp = fopen(filename, "rb");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filename);

    status = fread(&index, sizeof(INT), 1, fp);
    fasp_chkerr(status, filename);

    index = endian_convert_int(index, sizeof(INT), EndianFlag);

    flag    = (INT)index / 100;
    ilength = (int)(index - flag * 100) / 10;
    dlength = index % 10;

    switch (flag) {
        case 1:
            fasp_dcoo_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 2:
            fasp_dbsr_read_b(fp, (dBSRmat*)A, EndianFlag);
            break;
        case 3:
            fasp_dstr_read_b(fp, (dSTRmat*)A, EndianFlag);
            break;
        case 4:
            fasp_dcsr_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 5:
            fasp_dmtx_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        case 6:
            fasp_dmtxsym_read_b(fp, (dCSRmat*)A, EndianFlag);
            break;
        default:
            printf("### ERROR: Unknown flag %d in %s!\n", flag, filename);
            fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
    }

    fclose(fp);
}

void fasp_matrix_write(const char* filename, void* A, const INT flag)
{
    INT   fileflag, matrixflag;
    FILE* fp;

    matrixflag = flag % 100;
    fileflag   = (INT)flag / 100;

    // write matrix in ASCII file
    if (!fileflag) {

        fp = fopen(filename, "w");

        if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

        printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

        fprintf(fp, "%d%d%d%d\n", fileflag, fileflag, fileflag, fileflag);

        fprintf(fp, "%d%d%d\n", matrixflag, (int)sizeof(INT), (int)sizeof(REAL));

        switch (matrixflag) {
            case 1:
                fasp_dcsr_write_s(fp, (dCSRmat*)A);
                break;
            case 2:
                fasp_dbsr_write_s(fp, (dBSRmat*)A);
                break;
            case 3:
                fasp_dstr_write_s(fp, (dSTRmat*)A);
                break;
            default:
                printf("### WARNING: Unknown matrix flag %d\n", matrixflag);
        }
        fclose(fp);
        return;
    }

    // write matrix in binary file
    fp = fopen(filename, "wb");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filename);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filename);

    INT putflag = fileflag * 100 + sizeof(INT) * 10 + sizeof(REAL);
    fwrite(&putflag, sizeof(INT), 1, fp);

    switch (matrixflag) {
        case 1:
            fasp_dcsr_write_b(fp, (dCSRmat*)A);
            break;
        case 2:
            fasp_dbsr_write_b(fp, (dBSRmat*)A);
            break;
        case 3:
            fasp_dstr_write_b(fp, (dSTRmat*)A);
            break;
        default:
            printf("### WARNING: Unknown matrix flag %d\n", matrixflag);
    }

    fclose(fp);
}

void fasp_vector_read(const char* filerhs, void* b)
{
    int    index, flag;
    SHORT  EndianFlag;
    size_t status;

    FILE* fp = fopen(filerhs, "rb");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);

    printf("%s:  reading file %s ...\n", __FUNCTION__, filerhs);

    status = fread(&index, sizeof(INT), 1, fp);
    fasp_chkerr(status, filerhs);

    // vector stored in ASCII
    if (index == 808464432) {

        fclose(fp);
        fp = fopen(filerhs, "r");

        if (!fscanf(fp, "%d\n", &flag))
            printf("### ERROR: File format problem in %s!\n", __FUNCTION__);
        // TODO: Check why skip this flag ???  --Chensong

        if (!fscanf(fp, "%d\n", &flag))
            printf("### ERROR: File format problem in %s!\n", __FUNCTION__);
        flag = (int)flag / 100;

        switch (flag) {
            case 1:
                fasp_dvec_read_s(fp, (dvector*)b);
                break;
            case 2:
                fasp_ivec_read_s(fp, (ivector*)b);
                break;
            case 3:
                fasp_dvecind_read_s(fp, (dvector*)b);
                break;
            case 4:
                fasp_ivecind_read_s(fp, (ivector*)b);
                break;
        }
        fclose(fp);
        return;
    }

    // vector stored in binary
    EndianFlag = index;
    status     = fread(&index, sizeof(INT), 1, fp);
    fasp_chkerr(status, filerhs);

    index   = endian_convert_int(index, sizeof(INT), EndianFlag);
    flag    = (int)index / 100;
    ilength = (int)(index - 100 * flag) / 10;
    dlength = index % 10;

    switch (flag) {
        case 1:
            fasp_dvec_read_b(fp, (dvector*)b, EndianFlag);
            break;
        case 2:
            fasp_ivec_read_b(fp, (ivector*)b, EndianFlag);
            break;
        case 3:
            fasp_dvecind_read_b(fp, (dvector*)b, EndianFlag);
            break;
        case 4:
            fasp_ivecind_read_b(fp, (ivector*)b, EndianFlag);
            break;
        default:
            printf("### ERROR: Unknown flag %d in %s!\n", flag, filerhs);
            fasp_chkerr(ERROR_WRONG_FILE, __FUNCTION__);
    }

    fclose(fp);
}

void fasp_vector_write(const char* filerhs, void* b, const INT flag)
{

    INT   fileflag, vectorflag;
    FILE* fp;

    fileflag   = (int)flag / 10;
    vectorflag = (int)flag % 10;

    // write vector in ASCII
    if (!fileflag) {
        fp = fopen(filerhs, "w");

        if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);

        printf("%s:  writing to file %s ...\n", __FUNCTION__, filerhs);

        fprintf(fp, "%d%d%d%d\n", fileflag, fileflag, fileflag, fileflag);

        fprintf(fp, "%d%d%d\n", vectorflag, (int)sizeof(INT), (int)sizeof(REAL));

        switch (vectorflag) {
            case 1:
                fasp_dvec_write_s(fp, (dvector*)b);
                break;
            case 2:
                fasp_ivec_write_s(fp, (ivector*)b);
                break;
            case 3:
                fasp_dvecind_write_s(fp, (dvector*)b);
                break;
            case 4:
                fasp_ivecind_write_s(fp, (ivector*)b);
                break;
            default:
                printf("### WARNING: Unknown vector flag %d\n", vectorflag);
        }

        fclose(fp);
        return;
    }

    // write vector in binary
    fp = fopen(filerhs, "wb");

    if (fp == NULL) fasp_chkerr(ERROR_OPEN_FILE, filerhs);

    printf("%s:  writing to file %s ...\n", __FUNCTION__, filerhs);

    INT putflag = vectorflag * 100 + sizeof(INT) * 10 + sizeof(REAL);
    fwrite(&putflag, sizeof(INT), 1, fp);

    switch (vectorflag) {
        case 1:
            fasp_dvec_write_b(fp, (dvector*)b);
            break;
        case 2:
            fasp_ivec_write_b(fp, (ivector*)b);
            break;
        case 3:
            fasp_dvecind_write_b(fp, (dvector*)b);
            break;
        case 4:
            fasp_ivecind_write_b(fp, (ivector*)b);
            break;
        default:
            printf("### WARNING: Unknown vector flag %d\n", vectorflag);
    }

    fclose(fp);
}

void fasp_hb_read(const char* input_file, dCSRmat* A, dvector* b)
{
    //-------------------------
    // Setup local variables
    //-------------------------
    // variables for FASP
    dCSRmat tempA;

    // variables for hb_io

    int*    colptr = NULL;
    double* exact  = NULL;
    double* guess  = NULL;
    int     i;
    int     indcrd;
    char*   indfmt = NULL;
    FILE*   input;
    int     j;
    char*   key    = NULL;
    char*   mxtype = NULL;
    int     ncol;
    int     neltvl;
    int     nnzero;
    int     nrhs;
    int     nrhsix;
    int     nrow;
    int     ptrcrd;
    char*   ptrfmt = NULL;
    int     rhscrd;
    char*   rhsfmt = NULL;
    int*    rhsind = NULL;
    int*    rhsptr = NULL;
    char*   rhstyp = NULL;
    double* rhsval = NULL;
    double* rhsvec = NULL;
    int*    rowind = NULL;
    char*   title  = NULL;
    int     totcrd;
    int     valcrd;
    char*   valfmt = NULL;
    double* values = NULL;

    printf("\n");
    printf("HB_FILE_READ reads all the data in an HB file.\n");

    printf("\n");
    printf("Reading the file '%s'\n", input_file);

    input = fopen(input_file, "rt");

    if (!input) {
        printf("### ERROR: Fail to open the file [%s]\n", input_file);
        fasp_chkerr(ERROR_OPEN_FILE, __FUNCTION__);
    }

    //-------------------------
    // Reading...
    //-------------------------
    hb_file_read(input, &title, &key, &totcrd, &ptrcrd, &indcrd, &valcrd, &rhscrd,
                 &mxtype, &nrow, &ncol, &nnzero, &neltvl, &ptrfmt, &indfmt, &valfmt,
                 &rhsfmt, &rhstyp, &nrhs, &nrhsix, &colptr, &rowind, &values, &rhsval,
                 &rhsptr, &rhsind, &rhsvec, &guess, &exact);

    //-------------------------
    // Printing if needed
    //-------------------------
#if DEBUG_MODE > PRINT_MIN
    /*
Print out the header information.
*/
    hb_header_print(title, key, totcrd, ptrcrd, indcrd, valcrd, rhscrd, mxtype, nrow,
                    ncol, nnzero, neltvl, ptrfmt, indfmt, valfmt, rhsfmt, rhstyp, nrhs,
                    nrhsix);
    /*
Print the structure information.
*/
    hb_structure_print(ncol, mxtype, nnzero, neltvl, colptr, rowind);

    /*
Print the values.
*/
    hb_values_print(ncol, colptr, mxtype, nnzero, neltvl, values);

    if (0 < rhscrd) {
        /*
Print a bit of the right hand sides.
*/
        if (rhstyp[0] == 'F') {
            r8mat_print_some(nrow, nrhs, rhsval, 1, 1, 5, 5, "  Part of RHS");
        } else if (rhstyp[0] == 'M' && mxtype[2] == 'A') {
            i4vec_print_part(nrhs + 1, rhsptr, 10, "  Part of RHSPTR");
            i4vec_print_part(nrhsix, rhsind, 10, "  Part of RHSIND");
            r8vec_print_part(nrhsix, rhsvec, 10, "  Part of RHSVEC");
        } else if (rhstyp[0] == 'M' && mxtype[2] == 'E') {
            r8mat_print_some(nnzero, nrhs, rhsval, 1, 1, 5, 5, "  Part of RHS");
        }
        /*
Print a bit of the starting guesses.
*/
        if (rhstyp[1] == 'G') {
            r8mat_print_some(nrow, nrhs, guess, 1, 1, 5, 5, "  Part of GUESS");
        }
        /*
Print a bit of the exact solutions.
*/
        if (rhstyp[2] == 'X') {
            r8mat_print_some(nrow, nrhs, exact, 1, 1, 5, 5, "  Part of EXACT");
        }
    }
#endif

    //-------------------------
    // Closing
    //-------------------------
    fclose(input);

    //-------------------------
    // Convert to FASP format
    //-------------------------

    // convert matrix
    if (ncol != nrow) {
        printf("### ERROR: The matrix is not square!  [%s]\n", __FUNCTION__);
        goto FINISHED;
    }

    tempA = fasp_dcsr_create(nrow, ncol, nnzero);

    for (i = 0; i <= ncol; i++) tempA.IA[i] = colptr[i] - 1;
    for (i = 0; i < nnzero; i++) tempA.JA[i] = rowind[i] - 1;
    fasp_darray_cp(nnzero, values, tempA.val);

    // if the matrix is symmeric
    if (mxtype[1] == 'S') {

        // A = A'+ A
        dCSRmat tempA_tran;
        fasp_dcsr_trans(&tempA, &tempA_tran);
        fasp_blas_dcsr_add(&tempA, 1.0, &tempA_tran, 1.0, A);
        fasp_dcsr_free(&tempA);
        fasp_dcsr_free(&tempA_tran);

        // modify diagonal entries
        for (i = 0; i < A->row; i++) {

            for (j = A->IA[i]; j < A->IA[i + 1]; j++) {

                if (A->JA[j] == i) {
                    A->val[j] = A->val[j] / 2;
                    break;
                }
            }
        }
    }
    // if the matrix is not symmetric
    else {
        fasp_dcsr_trans(&tempA, A);
        fasp_dcsr_free(&tempA);
    }

    // convert right hand side

    if (nrhs == 0) {

        printf("### ERROR: No right hand side!  [%s]\n", __FUNCTION__);
        goto FINISHED;
    } else if (nrhs > 1) {

        printf("### ERROR: More than one right hand side!  [%s]\n", __FUNCTION__);
        goto FINISHED;
    } else {

        fasp_dvec_alloc(nrow, b);
        fasp_darray_cp(nrow, rhsval, b->val);
    }

    //-------------------------
    // Cleanning
    //-------------------------
FINISHED:
    if (colptr) free(colptr);
    if (exact) free(exact);
    if (guess) free(guess);
    if (rhsind) free(rhsind);
    if (rhsptr) free(rhsptr);
    if (rhsval) free(rhsval);
    if (rhsvec) free(rhsvec);
    if (rowind) free(rowind);
    if (values) free(values);

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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)

    Resverse CMK ordering.

9.61.1 Detailed Description

Generating ordering using algebraic information.

Note

This file contains Level-1 (Bla) functions.

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

A	Pointer to matrix
oindex	Pointer to index of vertices in order
order	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 )
```

Resverse CMK ordering.

Parameters

A	Pointer to matrix
order	Pointer to vertices with increasing degree
oindex	Pointer to index of vertices in order
rorder	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.

```
#include "fasp.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static void CMK_ordering (const dCSRmat *, INT, INT, INT, INT, INT *, INT *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_dcsr_CMK_order (const dCSRmat *A,
                          INT           *order,
                          INT           *oindex)
{
    const INT *ia = A->IA;
    const INT row = A->row;

    INT i, loc, s, vt, mindg, innz;

    s = 0;
    vt = 0;
    mindg = row+1;

    // select node with minimal degree
    for (i=0; i<row; ++i) {
        innz = ia[i+1] - ia[i];
        if (innz > 1) {
            oindex[i] = -innz;
            if (innz < mindg) {
                mindg = innz;
                vt = i;
            }
        }
        else { // order those diagonal rows first
```

```
            oindex[i] = s;
            order[s] = i;
            s ++;
        }
    }

    loc = s;

    // start to order
    CMK_ordering (A, loc, s, vt, mindg, oindex, order);
}

void fasp_dcsr_RCMK_order (const dCSRmat *A,
                           INT           *order,
                           INT           *oindex,
                           INT           *rorder)
{
    INT i;
    INT row = A->row;

    // Form CMK order
    fasp_dcsr_CMK_order(A, order, oindex);

    // Reverse CMK order
    for (i=0; i<row; ++i) rorder[i] = order[row-1-i];
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void CMK_ordering (const dCSRmat *A,
                          INT            loc,
                          INT            s,
                          INT            jj,
                          INT            mindg,
                          INT           *oindex,
                          INT           *order)
{
    const INT  row = A->row;
    const INT *ia  = A->IA;
    const INT *ja  = A->JA;

    INT       i, j, sp1, k;
    SHORT     flag = 1;

    if (s < row) {
        order[s] = jj;
        oindex[jj] = s;
    }

    while (loc <= s && s < row) {
        i = order[loc];
        sp1 = s+1;
        // neighbor nodes are priority.
        for (j=ia[i]+1; j<ia[i+1]; ++j) {
            k = ja[j];
            if (oindex[k] < 0){
                s++;
                order[s] = k;
            }
        }
        // ordering neighbor nodes by increasing degree
        if (s > sp1) {
            while (flag) {
                flag = 0;
                for (i=sp1+1; i<=s; ++i) {
                    if (oindex[order[i]] > oindex[order[i-1]]) {
                        j = order[i];
                        order[i] = order[i-1];
                        order[i-1] = j;
                        flag = 1;
                    }
                }
            }
        }

        for (i=sp1; i<=s; ++i) oindex[order[i]] = i;

        loc ++;
    }

    // deal with remainder
    if (s < row) {
        jj = 0;
        i  = 0;
        while (jj == 0) {
            i ++;
            if (i >= row) {
                mindg++;
                i = 0;
            }
            if (oindex[i] < 0 && (ia[i+1]-ia[i] == mindg)) {
                jj = i;
            }
        }

        s ++;

        CMK_ordering (A, loc, s, jj, mindg, oindex, order);
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

swzdata	Pointer to the Schwarz data
swzparam	Pointer to the Schwarz parameter
x	Pointer to solution vector
b	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

swzdata	Pointer to the Schwarz data
swzparam	Pointer to the Schwarz parameter
x	Pointer to solution vector
b	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

swzdata	Pointer to the Schwarz data
swzparam	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_functs.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;
```

```
    INT *iblock = NULL, *jblock = NULL, *mask = NULL, *maxa = NULL;

    // return
    INT flag = FASP_SUCCESS;

    swzdata->swzparam = swzparam;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // allocate memory
    maxa    = (INT *)fasp_mem_calloc(n,sizeof(INT));
    mask    = (INT *)fasp_mem_calloc(n,sizeof(INT));
    iblock  = (INT *)fasp_mem_calloc(n,sizeof(INT));
    jblock  = (INT *)fasp_mem_calloc(n,sizeof(INT));

    nsizeall=0;
    memset(mask,   0, sizeof(INT)*n);
    memset(iblock, 0, sizeof(INT)*n);
    memset(maxa,   0, sizeof(INT)*n);

    maxa[0]=0;

    // select root nodes
    MIS = fasp_sparse_mis(&A);

    /*-------------------------------------------*/
    // find the blocks
    /*-------------------------------------------*/

    // first pass:  do a maxlev level sets out for each node
    for ( i = 0; i < MIS.row; i++ ) {
        inroot = MIS.val[i];
        SWZ_level(inroot,&A,mask,&nlvl,maxa,jblock,maxlev);
        nsizei=maxa[nlvl];
        nsizeall+=nsizei;
    }

#if DEBUG_MODE > 1
    printf("### DEBUG: nsizeall = %d\n", nsizeall);
#endif

    // calculated the size of jblock up to here
    jblock = (INT *)fasp_mem_realloc(jblock,(nsizeall+n)*sizeof(INT));

    // second pass:  redo the same again, but this time we store in jblock
    maxa[0]=0;
    iblock[0]=0;
    nsizeall=0;
    jb=jblock;
    for (i=0;i<MIS.row;i++) {
        inroot = MIS.val[i];
        SWZ_level(inroot,&A,mask,&nlvl,maxa,jb,maxlev);
        nsizei=maxa[nlvl];
        iblock[i+1]=iblock[i]+nsizei;
        nsizeall+=nsizei;
        jb+=nsizei;
    }
    nblk = MIS.row;

#if DEBUG_MODE > 1
    printf("### DEBUG: nsizeall = %d, %d\n", nsizeall, iblock[nblk]);
#endif

    /*-------------------------------------------*/
    //  LU decomposition of blocks
    /*-------------------------------------------*/

    memset(mask, 0, sizeof(INT)*n);

    swzdata->blk_data = (dCSRmat*)fasp_mem_calloc(nblk, sizeof(dCSRmat));

    SWZ_block(swzdata, nblk, iblock, jblock, mask);

    // Setup for each block solver
    switch (blksolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            /* use MUMPS direct solver on each block */
            dCSRmat *blk = swzdata->blk_data;
            Mumps_data *mumps = (Mumps_data*)fasp_mem_calloc(nblk, sizeof(Mumps_data));
            for (i=0; i<nblk; ++i)
                mumps[i] = fasp_mumps_factorize(&blk[i], NULL, NULL, PRINT_NONE);
            swzdata->mumps = mumps;

            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            /* use UMFPACK direct solver on each block */
            dCSRmat *blk = swzdata->blk_data;
            void **numeric  = (void**)fasp_mem_calloc(nblk, sizeof(void*));
            dCSRmat Ac_tran;
            for (i=0; i<nblk; ++i) {
                Ac_tran = fasp_dcsr_create(blk[i].row, blk[i].col, blk[i].nnz);
                fasp_dcsr_transz(&blk[i], NULL, &Ac_tran);
                fasp_dcsr_cp(&Ac_tran, &blk[i]);
                numeric[i] = fasp_umfpack_factorize(&blk[i], 0);
            }
            swzdata->numeric = numeric;
            fasp_dcsr_free(&Ac_tran);

            break;
        }
#endif

        default:  {
            /* do nothing for iterative methods */
        }
    }

#if DEBUG_MODE > 1
    printf("### DEBUG: n = %d, #blocks = %d, max block size = %d\n",
           n, nblk, swzdata->maxbs);
#endif

    /*-------------------------------------------*/
    //  return
    /*-------------------------------------------*/
    swzdata->nblk   = nblk;
    swzdata->iblock = iblock;
    swzdata->jblock = jblock;
    swzdata->mask   = mask;
    swzdata->maxa   = maxa;
    swzdata->SWZ_type = swzparam->SWZ_type;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return flag;
}

void fasp_dcsr_swz_forward (SWZ_data   *swzdata,
                            SWZ_param  *swzparam,
                            dvector    *x,
                            dvector    *b)
{
    INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;

    // Schwarz partition
    INT      nblk      = swzdata->nblk;
    dCSRmat *blk       = swzdata->blk_data;
    INT     *iblock    = swzdata->iblock;
    INT     *jblock    = swzdata->jblock;
    INT     *mask      = swzdata->mask;
    INT      blksolver = swzparam->SWZ_blksolver;

    // Schwarz data
    dCSRmat    A    = swzdata->A;
    INT       *ia   = A.IA;
    INT       *ja   = A.JA;
    REAL      *val = A.val;

    // Local solution and right hand vectors
    dvector rhs    = swzdata->rhsloc1;
    dvector u      = swzdata->xloc1;

```

```
#if WITH_UMFPACK
    void **numeric = swzdata->numeric;
#endif

#if WITH_MUMPS
    Mumps_data *mumps = swzdata->mumps;
#endif

    for (is=0; is<nblk; ++is) {
        // Form the right hand of eack block
        ibl0 = iblock[is];
        ibl1 = iblock[is+1];
        nloc = ibl1-ibl0;
        for (i=0; i<nloc; ++i ) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            mask[ki] = i+1;
        }

        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki = jblock[iblk];
            rhs.val[i] = b->val[ki];
            iaa = ia[ki]-1;
            iab = ia[ki+1]-1;
            for (kij = iaa; kij<iab; ++kij) {
                kj = ja[kij]-1;
                j  = mask[kj];
                if(j == 0) {
                    rhs.val[i] -= val[kij]*x->val[kj];
                }
            }
        }

        // Solve each block
        switch (blksolver) {

#if WITH_MUMPS
            case SOLVER_MUMPS:  {
                /* use MUMPS direct solver on each block */
                fasp_mumps_solve(&blk[is], &rhs, &u, mumps[is], 0);
                break;
            }
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:  {
                /* use UMFPACK direct solver on each block */
                fasp_umfpack_solve(&blk[is], &rhs, &u, numeric[is], 0);
                break;
            }
#endif
            default:
                /* use iterative solver on each block */
                u.row = blk[is].row;
                rhs.row = blk[is].row;
                fasp_dvec_set(u.row, &u, 0);
                fasp_solver_dcsr_pvgmres(&blk[is], &rhs, &u, NULL, 1e-8, 100, 20, 1, 0);
        }

        //zero the mask so that everyting is as it was
        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            mask[ki] = 0;
            x->val[ki] = u.val[i];
        }
    }
}

void fasp_dcsr_swz_backward (SWZ_data   *swzdata,
                             SWZ_param  *swzparam,
                             dvector    *x,
                             dvector    *b)
{
    INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;

    // Schwarz partition
    INT      nblk      = swzdata->nblk;
    dCSRmat *blk       = swzdata->blk_data;
    INT     *iblock    = swzdata->iblock;
    INT      *jblock    = swzdata->jblock;
    INT      *mask      = swzdata->mask;
    INT       blksolver = swzparam->SWZ_blksolver;

    // Schwarz data
    dCSRmat  A    = swzdata->A;
    INT      *ia  = A.IA;
    INT      *ja  = A.JA;
    REAL     *val = A.val;

    // Local solution and right hand vectors
    dvector rhs   = swzdata->rhsloc1;
    dvector u     = swzdata->xloc1;

#if WITH_UMFPACK
    void **numeric = swzdata->numeric;
#endif

#if WITH_MUMPS
    Mumps_data *mumps = swzdata->mumps;
#endif

    for (is=nblk-1; is>=0; --is) {
        // Form the right hand of eack block
        ibl0 = iblock[is];
        ibl1 = iblock[is+1];
        nloc = ibl1-ibl0;
        for (i=0; i<nloc; ++i ) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            mask[ki] = i+1;
        }

        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki = jblock[iblk];
            rhs.val[i] = b->val[ki];
            iaa = ia[ki]-1;
            iab = ia[ki+1]-1;
            for (kij = iaa; kij<iab; ++kij) {
                kj = ja[kij]-1;
                j  = mask[kj];
                if(j == 0) {
                    rhs.val[i] -= val[kij]*x->val[kj];
                }
            }
        }

        // Solve each block
        switch (blksolver) {

#if WITH_MUMPS
            case SOLVER_MUMPS:  {
                /* use MUMPS direct solver on each block */
                fasp_mumps_solve(&blk[is], &rhs, &u, mumps[is], 0);
                break;
            }
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:  {
                /* use UMFPACK direct solver on each block */
                fasp_umfpack_solve(&blk[is], &rhs, &u, numeric[is], 0);
                break;
            }
#endif
            default:
                /* use iterative solver on each block */
                rhs.row = blk[is].row;
                u.row   = blk[is].row;
                fasp_dvec_set(u.row, &u, 0);
                fasp_solver_dcsr_pvgmres (&blk[is], &rhs, &u, NULL, 1e-8, 100, 20, 1, 0);
        }

        //zero the mask so that everyting is as it was
        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            mask[ki] = 0;
            x->val[ki] = u.val[i];
        }
    }
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void SWZ_level (const INT   inroot,
                       dCSRmat    *A,
                       INT        *mask,
                       INT        *nlvl,
                       INT        *iblock,
                       INT        *jblock,
                       const INT   maxlev)
{
    INT *ia = A->IA;
    INT *ja = A->JA;
    INT nnz = A->nnz;
    INT i, j, lvl, lbegin, lvlend, nsize, node;
    INT jstrt, jstop, nbr, lvsize;

    // This is diagonal
    if (ia[inroot+1]-ia[inroot] <= 1) {
        lvl = 0;
        iblock[lvl] = 0;
        jblock[iblock[lvl]] = inroot;
        lvl ++;
        iblock[lvl] = 1;
    }
    else {
        // input node as root node (level 0)
        lvl = 0;
        jblock[0] = inroot;
        lvlend = 0;
        nsize  = 1;
        // mark root node
        mask[inroot] = 1;

        lvsize = nnz;

        // form the level hierarchy for root node(level1, level2, ...  maxlev)
        while (lvsize > 0 && lvl < maxlev) {
            lbegin = lvlend;
            lvlend = nsize;
            iblock[lvl] = lbegin;
            lvl ++;
            for(i=lbegin; i<lvlend; ++i) {
                node = jblock[i];
                jstrt = ia[node]-1;
                jstop = ia[node+1]-1;
                for (j = jstrt; j<jstop; ++j) {
                    nbr = ja[j]-1;
                    if (mask[nbr] == 0) {
                        jblock[nsize] = nbr;
                        mask[nbr] = lvl;
                        nsize ++;
                    }
                }
            }
            lvsize = nsize - lvlend;
        }

        iblock[lvl] = nsize;

        // reset mask array
        for (i = 0; i< nsize; ++i) {
            node = jblock[i];
            mask[node] = 0;
        }
    }

    *nlvl = lvl;
}

static void SWZ_block (SWZ_data   *swzdata,
                       const INT   nblk,
                       const INT  *iblock,
                       const INT  *jblock,
                       INT        *mask)
{
    INT i, j, iblk, ki, kj, kij, is, ibl0, ibl1, nloc, iaa, iab;
    INT maxbs = 0, count, nnz;

    dCSRmat A = swzdata->A;
    dCSRmat *blk = swzdata->blk_data;

    INT  *ia  = A.IA;
    INT  *ja  = A.JA;
    REAL *val = A.val;

    // get maximal block size
    for (is=0; is<nblk; ++is) {
        ibl0 = iblock[is];
        ibl1 = iblock[is+1];
        nloc = ibl1-ibl0;
        maxbs = MAX(maxbs, nloc);
    }

    swzdata->maxbs = maxbs;

    // allocate memory for each sub_block's right hand
    swzdata->xloc1   = fasp_dvec_create(maxbs);
    swzdata->rhsloc1 = fasp_dvec_create(maxbs);

    for (is=0; is<nblk; ++is) {
        ibl0 = iblock[is];
        ibl1 = iblock[is+1];
        nloc = ibl1-ibl0;
        count = 0;
        for (i=0; i<nloc; ++i ) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            iaa  = ia[ki]-1;
            iab  = ia[ki+1]-1;
            count += iab - iaa;
            mask[ki] = i+1;
        }

        blk[is] = fasp_dcsr_create(nloc, nloc, count);
        blk[is].IA[0] = 0;
        nnz = 0;

        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki = jblock[iblk];
            iaa = ia[ki]-1;
            iab = ia[ki+1]-1;
            for (kij = iaa; kij<iab; ++kij) {
                kj = ja[kij]-1;
                j  = mask[kj];
                if(j != 0) {
                    blk[is].JA[nnz] = j-1;
                    blk[is].val[nnz] = val[kij];
                    nnz ++;
                }
            }
            blk[is].IA[i+1] = nnz;
        }

        blk[is].nnz = nnz;

        // zero the mask so that everyting is as it was
        for (i=0; i<nloc; ++i) {
            iblk = ibl0 + i;
            ki   = jblock[iblk];
            mask[ki] = 0;
        }
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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_mxv_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_mxv_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_mxv_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_mxv_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_mxv_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_mxv (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

Warning

These rountines 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

alpha	REAL factor alpha

Parameters

A	Pointer to the REAL array which stands for a n∗n full matrix
x	Pointer to the REAL array with length n
beta	REAL factor beta
y	Pointer to the REAL array with length n
n	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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array which stands a n∗n matrix
n	Dimension of the matrix
alpha	Scalar
beta	Scalar
c	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

a	Pointer to the REAL array which stands a n∗n matrix
n	Dimension of the matrix
alpha	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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array which stands a n∗n matrix
c	Pointer to the REAL array which stands a n∗n matrix
n	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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array which stands a n∗n matrix
c	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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array which stands a n∗n matrix
c	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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array which stands a n∗n matrix
c	Pointer to the REAL array which stands a n∗n matrix

Author

Li Zhao

Date

04/18/2021

Definition at line 341 of file BlaSmallMat.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

a	Pointer to the REAL array which stands a 5∗5 matrix
b	Pointer to the REAL array which stands a 5∗5 matrix
c	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 BlaSmallMat.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

a	Pointer to the REAL array which stands a 7∗7 matrix
b	Pointer to the REAL array which stands a 7∗7 matrix
c	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_mxv()

```
void fasp_blas_smat_mxv (
              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

a	Pointer to the REAL array which stands a n∗n matrix
b	Pointer to the REAL array with length n
c	Pointer to the REAL array with length n
n	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_mxv_nc2()

```
void fasp_blas_smat_mxv_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

a	Pointer to the REAL array which stands a 2∗2 matrix
b	Pointer to the REAL array with length 2
c	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_mxv_nc3()

```
void fasp_blas_smat_mxv_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

a	Pointer to the REAL array which stands a 3∗3 matrix
b	Pointer to the REAL array with length 3
c	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_mxv_nc4()

```
void fasp_blas_smat_mxv_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

a	Pointer to the REAL array which stands a 4∗4 matrix
b	Pointer to the REAL array with length 4
c	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_mxv_nc5()

```
void fasp_blas_smat_mxv_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

a	Pointer to the REAL array which stands a 5∗5 matrix
b	Pointer to the REAL array with length 5
c	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_mxv_nc7()

```
void fasp_blas_smat_mxv_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

a	Pointer to the REAL array which stands a 7∗7 matrix
b	Pointer to the REAL array with length 7
c	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∗n dense matrix.

Parameters

A	Pointer to the n∗n dense matrix
x	Pointer to the REAL array with length n
y	Pointer to the REAL array with length n
n	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

A	Pointer to the 2∗2 dense matrix
x	Pointer to the REAL array with length 3
y	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

A	Pointer to the 3∗3 dense matrix
x	Pointer to the REAL array with length 3
y	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

A	Pointer to the 4∗4 dense matrix
x	Pointer to the REAL array with length 4
y	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

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

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

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

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∗n dense matrix.

Parameters

A	Pointer to the n∗n dense matrix
x	Pointer to the REAL array with length n
y	Pointer to the REAL array with length n
n	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

A	Pointer to the 3∗3 dense matrix
x	Pointer to the REAL array with length 3
y	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

A	Pointer to the 3∗3 dense matrix
x	Pointer to the REAL array with length 3
y	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

A	Pointer to the 4∗4 dense matrix
x	Pointer to the REAL array with length 4
y	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.

```
#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_smat_axm (REAL       *a,
                         const INT   n,
                         const REAL  alpha)
{
    const INT  n2 = n*n;
    INT        i;

    for ( i = 0; i < n2; i++ ) a[i] *= alpha;

    return;
}

void fasp_blas_smat_add (const REAL  *a,
                         const REAL  *b,
                         const INT    n,
                         const REAL   alpha,
                         const REAL   beta,
                         REAL        *c)
{
    const INT  n2 = n*n;
    INT        i;

    for ( i = 0; i < n2; i++ ) c[i] = alpha * a[i] + beta * b[i];

    return;
}


void fasp_blas_smat_mxv_nc2 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL b0 = b[0], b1 = b[1];

    c[0] = a[0]*b0 + a[1]*b1;
    c[1] = a[2]*b0 + a[3]*b1;
}

void fasp_blas_smat_mxv_nc3 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL b0 = b[0], b1 = b[1], b2 = b[2];

    c[0] = a[0]*b0 + a[1]*b1 + a[2]*b2;
    c[1] = a[3]*b0 + a[4]*b1 + a[5]*b2;
    c[2] = a[6]*b0 + a[7]*b1 + a[8]*b2;
}

void fasp_blas_smat_mxv_nc4 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3];

    c[0] = a[0] *b0 + a[1] *b1 + a[2] *b2 + a[3] *b3;
    c[1] = a[4] *b0 + a[5] *b1 + a[6] *b2 + a[7] *b3;
    c[2] = a[8] *b0 + a[9] *b1 + a[10]*b2 + a[11]*b3;
    c[3] = a[12]*b0 + a[13]*b1 + a[14]*b2 + a[15]*b3;
}

```

```
void fasp_blas_smat_mxv_nc5 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL b0 = b[0], b1 = b[1], b2 = b[2];
    const REAL b3 = b[3], b4 = b[4];

    c[0] = a[0]*b0 + a[1]*b1 + a[2]*b2 + a[3]*b3 + a[4]*b4;
    c[1] = a[5]*b0 + a[6]*b1 + a[7]*b2 + a[8]*b3 + a[9]*b4;
    c[2] = a[10]*b0 + a[11]*b1 + a[12]*b2 + a[13]*b3 + a[14]*b4;
    c[3] = a[15]*b0 + a[16]*b1 + a[17]*b2 + a[18]*b3 + a[19]*b4;
    c[4] = a[20]*b0 + a[21]*b1 + a[22]*b2 + a[23]*b3 + a[24]*b4;
}

void fasp_blas_smat_mxv_nc7 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL b0 = b[0], b1 = b[1], b2 = b[2];
    const REAL b3 = b[3], b4 = b[4], b5 = b[5], b6 = b[6];

    c[0] = a[0]*b0  +  a[1]*b1 + a[2]*b2 + a[3]*b3  + a[4]*b4  + a[5]*b5 + a[6]*b6;
    c[1] = a[7]*b0  +  a[8]*b1 + a[9]*b2 + a[10]*b3 + a[11]*b4 + a[12]*b5 + a[13]*b6;
    c[2] = a[14]*b0 + a[15]*b1 + a[16]*b2 + a[17]*b3 + a[18]*b4 + a[19]*b5 + a[20]*b6;
    c[3] = a[21]*b0 + a[22]*b1 + a[23]*b2 + a[24]*b3 + a[25]*b4 + a[26]*b5 + a[27]*b6;
    c[4] = a[28]*b0 + a[29]*b1 + a[30]*b2 + a[31]*b3 + a[32]*b4 + a[33]*b5 + a[34]*b6;
    c[5] = a[35]*b0 + a[36]*b1 + a[37]*b2 + a[38]*b3 + a[39]*b4 + a[40]*b5 + a[41]*b6;
    c[6] = a[42]*b0 + a[43]*b1 + a[44]*b2 + a[45]*b3 + a[46]*b4 + a[47]*b5 + a[48]*b6;
}

void fasp_blas_smat_mxv (const REAL  *a,
                         const REAL  *b,
                         REAL        *c,
                         const INT    n)
{
    switch (n) {
    case 2:
        fasp_blas_smat_mxv_nc2(a, b, c);
        break;

    case 3:
        fasp_blas_smat_mxv_nc3(a, b, c);
        break;

    case 4:
        fasp_blas_smat_mxv_nc4(a, b, c);
        break;

    case 5:
        fasp_blas_smat_mxv_nc5(a, b, c);
        break;

    case 7:
        fasp_blas_smat_mxv_nc7(a, b, c);
        break;

    default:
        {
            INT i,j,in=0;
            REAL temp;

            for (i=0; i<n; ++i, in+=n) {
                temp = 0.0;
                for (j=0; j<n; ++j) temp += a[in+j]*b[j];
                c[i]=temp;
            } // end for i
        }
        break;
    }
    return;
}

void fasp_blas_smat_mul_nc2 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL a0 = a[0], a1 = a[1];
    const REAL a2 = a[2], a3 = a[3];

    const REAL b0 = b[0], b1 = b[1];
    const REAL b2 = b[2], b3 = b[3];

    c[0] = a0*b0 + a1*b2;
    c[1] = a0*b1 + a1*b3;
    c[2] = a2*b0 + a3*b2;
    c[3] = a2*b1 + a3*b3;

}

void fasp_blas_smat_mul_nc3 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL a0 = a[0], a1 = a[1], a2 = a[2];
    const REAL a3 = a[3], a4 = a[4], a5 = a[5];
    const REAL a6 = a[6], a7 = a[7], a8 = a[8];

    const REAL b0 = b[0], b1 = b[1], b2 = b[2];
    const REAL b3 = b[3], b4 = b[4], b5 = b[5];
    const REAL b6 = b[6], b7 = b[7], b8 = b[8];

    c[0] = a0*b0 + a1*b3 + a2*b6;
    c[1] = a0*b1 + a1*b4 + a2*b7;
    c[2] = a0*b2 + a1*b5 + a2*b8;

    c[3] = a3*b0 + a4*b3 + a5*b6;
    c[4] = a3*b1 + a4*b4 + a5*b7;
    c[5] = a3*b2 + a4*b5 + a5*b8;

    c[6] = a6*b0 + a7*b3 + a8*b6;
    c[7] = a6*b1 + a7*b4 + a8*b7;
    c[8] = a6*b2 + a7*b5 + a8*b8;
}

void fasp_blas_smat_mul_nc4 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL a0  = a[0],  a1  = a[1],  a2  = a[2],   a3  = a[3];
    const REAL a4  = a[4],  a5  = a[5],  a6  = a[6],   a7  = a[7];
    const REAL a8  = a[8],  a9  = a[9],  a10 = a[10], a11 = a[11];
    const REAL a12 = a[12], a13 = a[13], a14 = a[14], a15 = a[15];

    const REAL b0  = b[0],  b1  = b[1],  b2  = b[2],   b3 = b[3];
    const REAL b4  = b[4],  b5  = b[5],  b6  = b[6],   b7 = b[7];
    const REAL b8  = b[8],  b9  = b[9],  b10 = b[10], b11 = b[11];
    const REAL b12 = b[12], b13 = b[13], b14 = b[14], b15 = b[15];

    c[0] = a0*b0 + a1*b4 + a2*b8  + a3*b12;
    c[1] = a0*b1 + a1*b5 + a2*b9  + a3*b13;
    c[2] = a0*b2 + a1*b6 + a2*b10 + a3*b14;
    c[3] = a0*b3 + a1*b7 + a2*b11 + a3*b15;

    c[4] = a4*b0 + a5*b4 + a6*b8  + a7*b12;
    c[5] = a4*b1 + a5*b5 + a6*b9  + a7*b13;
    c[6] = a4*b2 + a5*b6 + a6*b10 + a7*b14;
    c[7] = a4*b3 + a5*b7 + a6*b11 + a7*b15;

    c[8]  = a8*b0 + a9*b4 + a10*b8  + a11*b12;
    c[9]  = a8*b1 + a9*b5 + a10*b9  + a11*b13;
    c[10] = a8*b2 + a9*b6 + a10*b10 + a11*b14;
    c[11] = a8*b3 + a9*b7 + a10*b11 + a11*b15;

    c[12] = a12*b0 + a13*b4 + a14*b8  + a15*b12;
    c[13] = a12*b1 + a13*b5 + a14*b9  + a15*b13;
    c[14] = a12*b2 + a13*b6 + a14*b10 + a15*b14;
    c[15] = a12*b3 + a13*b7 + a14*b11 + a15*b15;
}

void fasp_blas_smat_mul_nc5 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL a0  = a[0],  a1  = a[1],  a2  = a[2],  a3  = a[3],  a4  = a[4];
    const REAL a5  = a[5],  a6  = a[6],  a7  = a[7],  a8  = a[8],  a9  = a[9];
    const REAL a10 = a[10], a11 = a[11], a12 = a[12], a13 = a[13], a14 = a[14];
    const REAL a15 = a[15], a16 = a[16], a17 = a[17], a18 = a[18], a19 = a[19];
    const REAL a20 = a[20], a21 = a[21], a22 = a[22], a23 = a[23], a24 = a[24];

    const REAL b0  = b[0],  b1  = b[1],  b2  = b[2],  b3  = b[3],  b4  = b[4];
    const REAL b5  = b[5],  b6  = b[6],  b7  = b[7],  b8  = b[8],  b9  = b[9];
    const REAL b10 = b[10], b11 = b[11], b12 = b[12], b13 = b[13], b14 = b[14];
    const REAL b15 = b[15], b16 = b[16], b17 = b[17], b18 = b[18], b19 = b[19];
    const REAL b20 = b[20], b21 = b[21], b22 = b[22], b23 = b[23], b24 = b[24];

    c[0] = a0*b0 + a1*b5 + a2*b10 + a3*b15 + a4*b20;
    c[1] = a0*b1 + a1*b6 + a2*b11 + a3*b16 + a4*b21;
    c[2] = a0*b2 + a1*b7 + a2*b12 + a3*b17 + a4*b22;
    c[3] = a0*b3 + a1*b8 + a2*b13 + a3*b18 + a4*b23;
    c[4] = a0*b4 + a1*b9 + a2*b14 + a3*b19 + a4*b24;

    c[5] = a5*b0 + a6*b5 + a7*b10 + a8*b15 + a9*b20;
    c[6] = a5*b1 + a6*b6 + a7*b11 + a8*b16 + a9*b21;
    c[7] = a5*b2 + a6*b7 + a7*b12 + a8*b17 + a9*b22;
    c[8] = a5*b3 + a6*b8 + a7*b13 + a8*b18 + a9*b23;
    c[9] = a5*b4 + a6*b9 + a7*b14 + a8*b19 + a9*b24;

    c[10] = a10*b0 + a11*b5 + a12*b10 + a13*b15 + a14*b20;
    c[11] = a10*b1 + a11*b6 + a12*b11 + a13*b16 + a14*b21;
    c[12] = a10*b2 + a11*b7 + a12*b12 + a13*b17 + a14*b22;
    c[13] = a10*b3 + a11*b8 + a12*b13 + a13*b18 + a14*b23;
    c[14] = a10*b4 + a11*b9 + a12*b14 + a13*b19 + a14*b24;

    c[15] = a15*b0 + a16*b5 + a17*b10 + a18*b15 + a19*b20;
    c[16] = a15*b1 + a16*b6 + a17*b11 + a18*b16 + a19*b21;
    c[17] = a15*b2 + a16*b7 + a17*b12 + a18*b17 + a19*b22;
    c[18] = a15*b3 + a16*b8 + a17*b13 + a18*b18 + a19*b23;
    c[19] = a15*b4 + a16*b9 + a17*b14 + a18*b19 + a19*b24;

    c[20] = a20*b0 + a21*b5 + a22*b10 + a23*b15 + a24*b20;
    c[21] = a20*b1 + a21*b6 + a22*b11 + a23*b16 + a24*b21;
    c[22] = a20*b2 + a21*b7 + a22*b12 + a23*b17 + a24*b22;
    c[23] = a20*b3 + a21*b8 + a22*b13 + a23*b18 + a24*b23;
    c[24] = a20*b4 + a21*b9 + a22*b14 + a23*b19 + a24*b24;
}

void fasp_blas_smat_mul_nc7 (const REAL  *a,
                             const REAL  *b,
                             REAL        *c)
{
    const REAL a0  = a[0],  a1  = a[1],  a2  = a[2],  a3  = a[3],  a4  = a[4],  a5  = a[5],  a6  = a[6];
    const REAL a7  = a[7],  a8  = a[8],  a9  = a[9],  a10 = a[10], a11 = a[11], a12 = a[12], a13 = a[13];
    const REAL a14 = a[14], a15 = a[15], a16 = a[16], a17 = a[17], a18 = a[18], a19 = a[19], a20 = a[20];
    const REAL a21 = a[21], a22 = a[22], a23 = a[23], a24 = a[24], a25 = a[25], a26 = a[26], a27 = a[27];
    const REAL a28 = a[28], a29 = a[29], a30 = a[30], a31 = a[31], a32 = a[32], a33 = a[33], a34 = a[34];
    const REAL a35 = a[35], a36 = a[36], a37 = a[37], a38 = a[38], a39 = a[39], a40 = a[40], a41 = a[41];
    const REAL a42 = a[42], a43 = a[43], a44 = a[44], a45 = a[45], a46 = a[46], a47 = a[47], a48 = a[48];

    const REAL b0  = b[0],  b1  = b[1],  b2  = b[2],  b3  = b[3],  b4  = b[4],  b5  = b[5],  b6  = b[6];
    const REAL b7  = b[7],  b8  = b[8],  b9  = b[9],  b10 = b[10], b11 = b[11], b12 = b[12], b13 = b[13];
    const REAL b14 = b[14], b15 = b[15], b16 = b[16], b17 = b[17], b18 = b[18], b19 = b[19], b20 = b[20];
    const REAL b21 = b[21], b22 = b[22], b23 = b[23], b24 = b[24], b25 = b[25], b26 = b[26], b27 = b[27];
    const REAL b28 = b[28], b29 = b[29], b30 = b[30], b31 = b[31], b32 = b[32], b33 = b[33], b34 = b[34];
    const REAL b35 = b[35], b36 = b[36], b37 = b[37], b38 = b[38], b39 = b[39], b40 = b[40], b41 = b[41];
    const REAL b42 = b[42], b43 = b[43], b44 = b[44], b45 = b[45], b46 = b[46], b47 = b[47], b48 = b[48];

    c[0] = a0*b0 + a1*b7 +  a2*b14 + a3*b21 + a4*b28 + a5*b35 + a6*b42;
    c[1] = a0*b1 + a1*b8 +  a2*b15 + a3*b22 + a4*b29 + a5*b36 + a6*b43;
    c[2] = a0*b2 + a1*b9 +  a2*b16 + a3*b23 + a4*b30 + a5*b37 + a6*b44;
    c[3] = a0*b3 + a1*b10 + a2*b17 + a3*b24 + a4*b31 + a5*b38 + a6*b45;
    c[4] = a0*b4 + a1*b11 + a2*b18 + a3*b25 + a4*b32 + a5*b39 + a6*b46;
    c[5] = a0*b5 + a1*b12 + a2*b19 + a3*b26 + a4*b33 + a5*b40 + a6*b47;
    c[6] = a0*b6 + a1*b13 + a2*b20 + a3*b27 + a4*b34 + a5*b41 + a6*b48;

    c[7]  = a7*b0 + a8*b7 +  a9*b14 + a10*b21 + a11*b28 + a12*b35 + a13*b42;
    c[8]  = a7*b1 + a8*b8 +  a9*b15 + a10*b22 + a11*b29 + a12*b36 + a13*b43;
    c[9]  = a7*b2 + a8*b9 +  a9*b16 + a10*b23 + a11*b30 + a12*b37 + a13*b44;
    c[10] = a7*b3 + a8*b10 + a9*b17 + a10*b24 + a11*b31 + a12*b38 + a13*b45;
    c[11] = a7*b4 + a8*b11 + a9*b18 + a10*b25 + a11*b32 + a12*b39 + a13*b46;
    c[12] = a7*b5 + a8*b12 + a9*b19 + a10*b26 + a11*b33 + a12*b40 + a13*b47;
    c[13] = a7*b6 + a8*b13 + a9*b20 + a10*b27 + a11*b34 + a12*b41 + a13*b48;

    c[14] = a14*b0 + a15*b7 +  a16*b14 + a17*b21 + a18*b28 + a19*b35 + a20*b42;
    c[15] = a14*b1 + a15*b8 +  a16*b15 + a17*b22 + a18*b29 + a19*b36 + a20*b43;
    c[16] = a14*b2 + a15*b9 +  a16*b16 + a17*b23 + a18*b30 + a19*b37 + a20*b44;
    c[17] = a14*b3 + a15*b10 + a16*b17 + a17*b24 + a18*b31 + a19*b38 + a20*b45;
    c[18] = a14*b4 + a15*b11 + a16*b18 + a17*b25 + a18*b32 + a19*b39 + a20*b46;
    c[19] = a14*b5 + a15*b12 + a16*b19 + a17*b26 + a18*b33 + a19*b40 + a20*b47;
    c[20] = a14*b6 + a15*b13 + a16*b20 + a17*b27 + a18*b34 + a19*b41 + a20*b48;

    c[21] = a21*b0 + a22*b7 +  a23*b14 + a24*b21 + a25*b28 + a26*b35 + a27*b42;
    c[22] = a21*b1 + a22*b8 +  a23*b15 + a24*b22 + a25*b29 + a26*b36 + a27*b43;
    c[23] = a21*b2 + a22*b9 +  a23*b16 + a24*b23 + a25*b30 + a26*b37 + a27*b44;
    c[24] = a21*b3 + a22*b10 + a23*b17 + a24*b24 + a25*b31 + a26*b38 + a27*b45;
    c[25] = a21*b4 + a22*b11 + a23*b18 + a24*b25 + a25*b32 + a26*b39 + a27*b46;
    c[26] = a21*b5 + a22*b12 + a23*b19 + a24*b26 + a25*b33 + a26*b40 + a27*b47;
    c[27] = a21*b6 + a22*b13 + a23*b20 + a24*b27 + a25*b34 + a26*b41 + a27*b48;

    c[28] = a28*b0 + a29*b7 +  a30*b14 + a31*b21 + a32*b28 + a33*b35 + a34*b42;
    c[29] = a28*b1 + a29*b8 +  a30*b15 + a31*b22 + a32*b29 + a33*b36 + a34*b43;
    c[30] = a28*b2 + a29*b9 +  a30*b16 + a31*b23 + a32*b30 + a33*b37 + a34*b44;
    c[31] = a28*b3 + a29*b10 + a30*b17 + a31*b24 + a32*b31 + a33*b38 + a34*b45;
    c[32] = a28*b4 + a29*b11 + a30*b18 + a31*b25 + a32*b32 + a33*b39 + a34*b46;
    c[33] = a28*b5 + a29*b12 + a30*b19 + a31*b26 + a32*b33 + a33*b40 + a34*b47;
    c[34] = a28*b6 + a29*b13 + a30*b20 + a31*b27 + a32*b34 + a33*b41 + a34*b48;

    c[35] = a35*b0 + a36*b7 +  a37*b14 + a38*b21 + a39*b28 + a40*b35 + a41*b42;
    c[36] = a35*b1 + a36*b8 +  a37*b15 + a38*b22 + a39*b29 + a40*b36 + a41*b43;
    c[37] = a35*b2 + a36*b9 +  a37*b16 + a38*b23 + a39*b30 + a40*b37 + a41*b44;
    c[38] = a35*b3 + a36*b10 + a37*b17 + a38*b24 + a39*b31 + a40*b38 + a41*b45;
    c[39] = a35*b4 + a36*b11 + a37*b18 + a38*b25 + a39*b32 + a40*b39 + a41*b46;
    c[40] = a35*b5 + a36*b12 + a37*b19 + a38*b26 + a39*b33 + a40*b40 + a41*b47;
    c[41] = a35*b6 + a36*b13 + a37*b20 + a38*b27 + a39*b34 + a40*b41 + a41*b48;

    c[42] = a42*b0 + a43*b7 +  a44*b14 + a45*b21 + a46*b28 + a47*b35 + a48*b42;
    c[43] = a42*b1 + a43*b8 +  a44*b15 + a45*b22 + a46*b29 + a47*b36 + a48*b43;
    c[44] = a42*b2 + a43*b9 +  a44*b16 + a45*b23 + a46*b30 + a47*b37 + a48*b44;
    c[45] = a42*b3 + a43*b10 + a44*b17 + a45*b24 + a46*b31 + a47*b38 + a48*b45;
    c[46] = a42*b4 + a43*b11 + a44*b18 + a45*b25 + a46*b32 + a47*b39 + a48*b46;
    c[47] = a42*b5 + a43*b12 + a44*b19 + a45*b26 + a46*b33 + a47*b40 + a48*b47;
    c[48] = a42*b6 + a43*b13 + a44*b20 + a45*b27 + a46*b34 + a47*b41 + a48*b48;
}

void fasp_blas_smat_mul (const REAL  *a,
                         const REAL  *b,
                         REAL        *c,
                         const INT    n)
{

    switch (n) {
    case 2:
        fasp_blas_smat_mul_nc2(a, b, c); break;

    case 3:
        fasp_blas_smat_mul_nc3(a, b, c); break;

    case 5:
        fasp_blas_smat_mul_nc5(a, b, c); break;

    case 7:
        fasp_blas_smat_mul_nc7(a, b, c); break;

    default:  {
        const INT n2 = n*n;
        INT i,j,k;
        REAL temp;

        for (i=0; i<n2; i+=n) {
            for (j=0; j<n; ++j) {
                temp = 0.0; // Fixed by Chensong.  Feb/22/2011.
                for (k=0; k<n; ++k) temp += a[i+k]*b[k*n+j];
                c[i+j] = temp;
            } // end for j
        } // end for i
    }
        break;
    }
    return;
}

void fasp_blas_smat_ypAx_nc2 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1];

    y[0] += A[0]*x0 + A[1]*x1;
    y[1] += A[2]*x0 + A[3]*x1;

    return;
}

void fasp_blas_smat_ypAx_nc3 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2];

    y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2;
    y[1] += A[3]*x0 + A[4]*x1 + A[5]*x2;
    y[2] += A[6]*x0 + A[7]*x1 + A[8]*x2;
    return;
}

void fasp_blas_smat_ypAx_nc4 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
     const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];

    y[0] +=  A[0]*x0 + A[1]*x1 +  A[2]*x2 + A[3]*x3;
    y[1] +=  A[4]*x0 + A[5]*x1 +  A[6]*x2 + A[7]*x3;
    y[2] +=  A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
    y[3] += A[12]*x0 + A[13]*x1 +A[14]*x2 + A[15]*x3;
    return;
}

void fasp_blas_smat_ypAx_nc5 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];

    y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
    y[1] += A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
    y[2] += A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
    y[3] += A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
    y[4] += A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
    return;
}

void fasp_blas_smat_ypAx_nc7 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
    const REAL  x4 = x[4], x5 = x[5], x6 = x[6];

    y[0] += A[0]*x0 + A[1]*x1 +    A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5 +  A[6]*x6;
    y[1] += A[7]*x0 + A[8]*x1 +    A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
    y[2] += A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
    y[3] += A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
    y[4] += A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
    y[5] += A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
    y[6] += A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
    return;
}

void fasp_blas_smat_ypAx (const REAL  *A,
                          const REAL  *x,
                          REAL        *y,
                          const INT    n)
{
    switch (n) {
    case 1:
        {
            y[0] += A[0]*x[0];
            break;
        }
    case 2:
        {
            const REAL x0 = x[0], x1 = x[1];
            y[0] += A[0]*x0 + A[1]*x1;
            y[1] += A[2]*x0 + A[3]*x1;
            break;
        }
    case 3:
        {
            const REAL x0 = x[0], x1 = x[1], x2 = x[2];
            y[0] += A[0]*x0 + A[1]*x1 + A[2]*x2;
            y[1] += A[3]*x0 + A[4]*x1 + A[5]*x2;
            y[2] += A[6]*x0 + A[7]*x1 + A[8]*x2;
            break;
        }
    case 4:
        {
            const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            y[0] +=  A[0]*x0 + A[1]*x1 +  A[2]*x2 + A[3]*x3;
            y[1] +=  A[4]*x0 + A[5]*x1 +  A[6]*x2 + A[7]*x3;
            y[2] +=  A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
            y[3] += A[12]*x0 + A[13]*x1 +A[14]*x2 + A[15]*x3;
            break;
        }
    case 5:
        {
            const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
            y[0] +=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4;
            y[1] +=  A[5]*x0 +  A[6]*x1 +  A[7]*x2 +  A[8]*x3 +  A[9]*x4;
            y[2] += A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
            y[3] += A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
            y[4] += A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
            break;
        }
    case 6:
        {
            const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            const REAL x4 = x[4], x5 = x[5];
            y[0] +=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5;
            y[1] +=  A[6]*x0 +  A[7]*x1 +  A[8]*x2 +  A[9]*x3 + A[10]*x4 + A[11]*x5;
            y[2] += A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3 + A[16]*x4 + A[17]*x5;
            y[3] += A[18]*x0 + A[19]*x1 + A[20]*x2 + A[21]*x3 + A[22]*x4 + A[23]*x5;
            y[4] += A[24]*x0 + A[25]*x1 + A[26]*x2 + A[27]*x3 + A[28]*x4 + A[29]*x5;
            y[5] += A[30]*x0 + A[31]*x1 + A[32]*x2 + A[33]*x3 + A[34]*x4 + A[35]*x5;
            break;
        }
    case 7:
        {
            const REAL x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            const REAL x4 = x[4], x5 = x[5], x6 = x[6];
            y[0] +=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5 +  A[6]*x6;
            y[1] +=  A[7]*x0 +  A[8]*x1 +  A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
            y[2] += A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
            y[3] += A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
            y[4] += A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
            y[5] += A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
            y[6] += A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
            break;
        }
    default:  /* For everything beyond 7 */
        {
            INT i,j,k;

            for ( k = i = 0; i < n; i++, k+=n ) {
                for ( j = 0; j < n; j++ ) {
                        y[i] += A[k+j]*x[j];
                    }
                }
            break;
        }
    }

    return;
}

void fasp_blas_smat_ymAx_nc2 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1];

    y[0] -= A[0]*x0 + A[1]*x1;
    y[1] -= A[2]*x0 + A[3]*x1;

    return;
}

void fasp_blas_smat_ymAx_nc3 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2];

    y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2;
    y[1] -= A[3]*x0 + A[4]*x1 + A[5]*x2;
    y[2] -= A[6]*x0 + A[7]*x1 + A[8]*x2;

    return;
}

void fasp_blas_smat_ymAx_nc4 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];

    y[0] -=  A[0]*x0 + A[1]*x1 +  A[2]*x2 + A[3]*x3;
    y[1] -=  A[4]*x0 + A[5]*x1 +  A[6]*x2 + A[7]*x3;
    y[2] -=  A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
    y[3] -= A[12]*x0 + A[13]*x1 +A[14]*x2 + A[15]*x3;
    return;
}

void fasp_blas_smat_ymAx_nc5 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];

    y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2 + A[3]*x3 + A[4]*x4;
    y[1] -= A[5]*x0 + A[6]*x1 + A[7]*x2 + A[8]*x3 + A[9]*x4;
    y[2] -= A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
    y[3] -= A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
    y[4] -= A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;

    return;
}

void fasp_blas_smat_ymAx_nc7 (const REAL  *A,
                              const REAL  *x,
                              REAL        *y)
{
    const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
    const REAL  x4 = x[4], x5 = x[5], x6 = x[6];

    y[0] -= A[0]*x0 + A[1]*x1 +    A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5 +  A[6]*x6;
    y[1] -= A[7]*x0 + A[8]*x1 +    A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
    y[2] -= A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
    y[3] -= A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
    y[4] -= A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
    y[5] -= A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
    y[6] -= A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;

    return;
}

void fasp_blas_smat_ymAx (const REAL  *A,
                          const REAL  *x,
                          REAL        *y,
                          const INT    n)
{
    switch (n) {
    case 1:
        {
            y[0] -= A[0]*x[0];
            break;
        }
    case 2:
        {
            const REAL  x0 = x[0], x1 = x[1];
            y[0] -= A[0]*x0 + A[1]*x1;
            y[1] -= A[2]*x0 + A[3]*x1;
            break;
        }
    case 3:
        {
            const REAL  x0 = x[0], x1 = x[1], x2 = x[2];
            y[0] -= A[0]*x0 + A[1]*x1 + A[2]*x2;
            y[1] -= A[3]*x0 + A[4]*x1 + A[5]*x2;
            y[2] -= A[6]*x0 + A[7]*x1 + A[8]*x2;
            break;
        }
    case 4:
        {
            const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            y[0] -=  A[0]*x0 + A[1]*x1 +  A[2]*x2 + A[3]*x3;
            y[1] -=  A[4]*x0 + A[5]*x1 +  A[6]*x2 + A[7]*x3;
            y[2] -=  A[8]*x0 + A[9]*x1 + A[10]*x2 + A[11]*x3;
            y[3] -= A[12]*x0 + A[13]*x1 +A[14]*x2 + A[15]*x3;
            break;
        }
    case 5:
        {
            const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3], x4 = x[4];
            y[0] -=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4;
            y[1] -=  A[5]*x0 +  A[6]*x1 +  A[7]*x2 +  A[8]*x3 +  A[9]*x4;
            y[2] -= A[10]*x0 + A[11]*x1 + A[12]*x2 + A[13]*x3 + A[14]*x4;
            y[3] -= A[15]*x0 + A[16]*x1 + A[17]*x2 + A[18]*x3 + A[19]*x4;
            y[4] -= A[20]*x0 + A[21]*x1 + A[22]*x2 + A[23]*x3 + A[24]*x4;
            break;
        }
    case 6:
        {
            const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            const REAL  x4 = x[4], x5 = x[5];
            y[0] -=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5;
            y[1] -=  A[6]*x0 +  A[7]*x1 +  A[8]*x2 +  A[9]*x3 + A[10]*x4 + A[11]*x5;
            y[2] -= A[12]*x0 + A[13]*x1 + A[14]*x2 + A[15]*x3 + A[16]*x4 + A[17]*x5;
            y[3] -= A[18]*x0 + A[19]*x1 + A[20]*x2 + A[21]*x3 + A[22]*x4 + A[23]*x5;
            y[4] -= A[24]*x0 + A[25]*x1 + A[26]*x2 + A[27]*x3 + A[28]*x4 + A[29]*x5;
            y[5] -= A[30]*x0 + A[31]*x1 + A[32]*x2 + A[33]*x3 + A[34]*x4 + A[35]*x5;
            break;
        }
    case 7:
        {
            const REAL  x0 = x[0], x1 = x[1], x2 = x[2], x3 = x[3];
            const REAL  x4 = x[4], x5 = x[5], x6 = x[6];
            y[0] -=  A[0]*x0 +  A[1]*x1 +  A[2]*x2 +  A[3]*x3 +  A[4]*x4 +  A[5]*x5 +  A[6]*x6;
            y[1] -=  A[7]*x0 +  A[8]*x1 +  A[9]*x2 + A[10]*x3 + A[11]*x4 + A[12]*x5 + A[13]*x6;
            y[2] -= A[14]*x0 + A[15]*x1 + A[16]*x2 + A[17]*x3 + A[18]*x4 + A[19]*x5 + A[20]*x6;
            y[3] -= A[21]*x0 + A[22]*x1 + A[23]*x2 + A[24]*x3 + A[25]*x4 + A[26]*x5 + A[27]*x6;
            y[4] -= A[28]*x0 + A[29]*x1 + A[30]*x2 + A[31]*x3 + A[32]*x4 + A[33]*x5 + A[34]*x6;
            y[5] -= A[35]*x0 + A[36]*x1 + A[37]*x2 + A[38]*x3 + A[39]*x4 + A[40]*x5 + A[41]*x6;
            y[6] -= A[42]*x0 + A[43]*x1 + A[44]*x2 + A[45]*x3 + A[46]*x4 + A[47]*x5 + A[48]*x6;
            break;
        }
    default:  // Everything beyond 7
        {
            INT i,j,k;

            for ( k = i = 0; i < n; i++, k+=n ) {
                for ( j = 0; j < n; j++ ) {
                    y[i] -= A[k+j]*x[j];
                }
            }
            break;
        }
    }

    return;
}

void fasp_blas_smat_aAxpby (const REAL   alpha,
                            const REAL  *A,
                            const REAL  *x,
                            const REAL   beta,
                            REAL        *y,
                            const INT    n)
{
    INT     i,j,k;
    REAL    tmp = 0.0;

    if ( alpha == 0 ) {
        for (i = 0; i < n; i ++) y[i] *= beta;
        return;
    }

    // y := (beta/alpha)y
    tmp = beta / alpha;
    if ( tmp != 1.0 ) {
        for (i = 0; i < n; i ++) y[i] *= tmp;
    }

    // y := y + A*x
    for ( k = i = 0; i < n; i++, k+=n ) {
        for (j = 0; j < n; j ++) {
            y[i] += A[k+j]*x[j];
        }
    }

    // y := alpha*y
    if ( alpha != 1.0 ) {
        for ( i = 0; i < n; i ++ ) y[i] *= alpha;
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

a	Pointer to the REAL vector which stands for a n∗n full matrix
n	Size of full matrix
n2	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

a	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

a	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

a	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

a	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

a	Pointer to the REAL array which stands a n∗n matrix
n	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

a	Pointer to the REAL array which stands a n∗n matrix
n	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

a	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

a	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

a	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

a	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

a	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

a	Pointer to the REAL array which stands a n∗n matrix
n	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∗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.

```c

#include "fasp.h"
#include "fasp_functs.h"

#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}
/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_smat_inv_nc2 (REAL *a)
{
    const REAL a0 = a[0], a1 = a[1];
    const REAL a2 = a[2], a3 = a[3];

    const REAL det = a0*a3 - a1*a2;

    if ( ABS(det) < SMALLREAL ) {
        printf("### WARNING: Matrix is nearly singular, det = %e!  Ignore.\n", det);
        printf("##----------------------------------------------\n");
        printf("## %12.5e %12.5e \n", a0, a1);
        printf("## %12.5e %12.5e \n", a2, a3);
        printf("##----------------------------------------------\n");

        a[0] = 1.0; a[1] = 0.0;
        a[2] = 0.0; a[3] = 1.0;
    }
    else {
        REAL det_inv = 1.0 / det;
        a[0] =  a3 * det_inv; a[1] = -a1 * det_inv;
        a[2] = -a2 * det_inv; a[3] = a0 * det_inv;
    }
}

void fasp_smat_inv_nc3 (REAL *a)
{
    const REAL a0 = a[0], a1 = a[1], a2 = a[2];
    const REAL a3 = a[3], a4 = a[4], a5 = a[5];
    const REAL a6 = a[6], a7 = a[7], a8 = a[8];

    const REAL M0 = a4*a8-a5*a7, M3 = a2*a7-a1*a8, M6 = a1*a5-a2*a4;
    const REAL M1 = a5*a6-a3*a8, M4 = a0*a8-a2*a6, M7 = a2*a3-a0*a5;
    const REAL M2 = a3*a7-a4*a6, M5 = a1*a6-a0*a7, M8 = a0*a4-a1*a3;

    const REAL det = a0*M0+a3*M3+a6*M6;

    if ( ABS(det) < SMALLREAL ) {
        printf("### WARNING: Matrix is nearly singular, det = %e!  Ignore.\n", det);
        printf("##----------------------------------------------\n");
        printf("## %12.5e %12.5e %12.5e \n", a0, a1, a2);
        printf("## %12.5e %12.5e %12.5e \n", a3, a4, a5);
        printf("## %12.5e %12.5e %12.5e \n", a6, a7, a8);
        printf("##----------------------------------------------\n");

        a[0] = 1.0; a[1] = 0.0; a[2] = 0.0;
        a[3] = 0.0; a[4] = 1.0; a[5] = 0.0;
        a[6] = 0.0; a[7] = 0.0; a[8] = 1.0;
    }
    else {
        REAL det_inv = 1.0/det;
        a[0] = M0*det_inv; a[1] = M3*det_inv; a[2] = M6*det_inv;
        a[3] = M1*det_inv; a[4] = M4*det_inv; a[5] = M7*det_inv;
        a[6] = M2*det_inv; a[7] = M5*det_inv; a[8] = M8*det_inv;
    }
}

void fasp_smat_inv_nc4 (REAL *a)
{
    const REAL a11 = a[0],  a12 = a[1],  a13 = a[2],  a14 = a[3];
    const REAL a21 = a[4],  a22 = a[5],  a23 = a[6],  a24 = a[7];
    const REAL a31 = a[8],  a32 = a[9],  a33 = a[10], a34 = a[11];
    const REAL a41 = a[12], a42 = a[13], a43 = a[14], a44 = a[15];

    const REAL M11 = a22*a33*a44 + a23*a34*a42 + a24*a32*a43 - a22*a34*a43 - a23*a32*a44 - a24*a33*a42;
    const REAL M12 = a12*a34*a43 + a13*a32*a44 + a14*a33*a42 - a12*a33*a44 - a13*a34*a42 - a14*a32*a43;
    const REAL M13 = a12*a23*a44 + a13*a24*a42 + a14*a22*a43 - a12*a24*a43 - a13*a22*a44 - a14*a23*a42;
    const REAL M14 = a12*a24*a33 + a13*a22*a34 + a14*a23*a32 - a12*a23*a34 - a13*a24*a32 - a14*a22*a33;
    const REAL M21 = a21*a34*a43 + a23*a31*a44 + a24*a33*a41 - a21*a33*a44 - a23*a34*a41 - a24*a31*a43;
    const REAL M22 = a11*a33*a44 + a13*a34*a41 + a14*a31*a43 - a11*a34*a43 - a13*a31*a44 - a14*a33*a41;
    const REAL M23 = a11*a24*a43 + a13*a21*a44 + a14*a23*a41 - a11*a23*a44 - a13*a24*a41 - a14*a21*a43;
    const REAL M24 = a11*a23*a34 + a13*a24*a31 + a14*a21*a33 - a11*a24*a33 - a13*a21*a34 - a14*a23*a31;
    const REAL M31 = a21*a32*a44 + a22*a34*a41 + a24*a31*a42 - a21*a34*a42 - a22*a31*a44 - a24*a32*a41;
    const REAL M32 = a11*a34*a42 + a12*a31*a44 + a14*a32*a41 - a11*a32*a44 - a12*a34*a41 - a14*a31*a42;
    const REAL M33 = a11*a22*a44 + a12*a24*a41 + a14*a21*a42 - a11*a24*a42 - a12*a21*a44 - a14*a22*a41;
    const REAL M34 = a11*a24*a32 + a12*a21*a34 + a14*a22*a31 - a11*a22*a34 - a12*a24*a31 - a14*a21*a32;
    const REAL M41 = a21*a33*a42 + a22*a31*a43 + a23*a32*a41 - a21*a32*a43 - a22*a33*a41 - a23*a31*a42;
    const REAL M42 = a11*a32*a43 + a12*a33*a41 + a13*a31*a42 - a11*a33*a42 - a12*a31*a43 - a13*a32*a41;
    const REAL M43 = a11*a23*a42 + a12*a21*a43 + a13*a22*a41 - a11*a22*a43 - a12*a23*a41 - a13*a21*a42;
    const REAL M44 = a11*a22*a33 + a12*a23*a31 + a13*a21*a32 - a11*a23*a32 - a12*a21*a33 - a13*a22*a31;

    const REAL det = a11*M11 + a12*M21 + a13*M31 + a14*M41;

    if ( ABS(det) < SMALLREAL ) {
        printf("### WARNING: Matrix is nearly singular, det = %e!  Ignore.\n", det);
        printf("##----------------------------------------------\n");
        printf("## %12.5e %12.5e %12.5e %12.5e\n", a11, a12, a13, a14);
        printf("## %12.5e %12.5e %12.5e %12.5e\n", a21, a22, a23, a24);
        printf("## %12.5e %12.5e %12.5e %12.5e\n", a31, a32, a33, a34);
        printf("## %12.5e %12.5e %12.5e %12.5e\n", a41, a42, a43, a44);
        printf("##----------------------------------------------\n");

        a[0]  = 1.0;  a[1]  = 0.0;  a[2]  = 0.0;  a[3]  = 0.0;
        a[4]  = 0.0;  a[5]  = 1.0;  a[6]  = 0.0;  a[7]  = 0.0;
        a[8]  = 0.0;  a[9]  = 0.0;  a[10] = 1.0;  a[11] = 0.0;
        a[12] = 0.0;  a[13] = 0.0;  a[14] = 0.0;  a[15] = 1.0;
    }
    else {
        REAL det_inv = 1.0 / det;
        a[0]  = M11 * det_inv;  a[1]  = M12 * det_inv;  a[2]  = M13 * det_inv;  a[3]  = M14 * det_inv;
        a[4]  = M21 * det_inv;  a[5]  = M22 * det_inv;  a[6]  = M23 * det_inv;  a[7]  = M24 * det_inv;
        a[8]  = M31 * det_inv;  a[9]  = M32 * det_inv;  a[10] = M33 * det_inv;  a[11] = M34 * det_inv;
        a[12] = M41 * det_inv;  a[13] = M42 * det_inv;  a[14] = M43 * det_inv;  a[15] = M44 * det_inv;
    }
}

void fasp_smat_inv_nc5 (REAL *a)
{
    const REAL a0=a[0],   a1=a[1],   a2=a[2],   a3=a[3],   a4=a[4];
    const REAL a5=a[5],   a6=a[6],   a7=a[7],   a8=a[8],   a9=a[9];
    const REAL a10=a[10], a11=a[11], a12=a[12], a13=a[13], a14=a[14];
    const REAL a15=a[15], a16=a[16], a17=a[17], a18=a[18], a19=a[19];
    const REAL a20=a[20], a21=a[21], a22=a[22], a23=a[23], a24=a[24];

    REAL det0, det1, det2, det3, det4, det;

    det0  =  a6  * ( a12 * (a18*a24-a19*a23) + a17 * (a14*a23-a13*a24) + a22 * (a13*a19 - a14*a18) );
    det0 +=  a11 * ( a7  * (a19*a23-a18*a24) + a17 * (a8*a24 -a9*a23 ) + a22 * (a9*a18  - a8*a19)  );
    det0 +=  a16 * ( a7  * (a13*a24-a14*a23) + a12 * (a9*a23 -a8*a24 ) + a22 * (a8*a14  - a9*a13)  );
    det0 +=  a21 * ( a17 * (a9*a13 -a8*a14 ) + a7  * (a14*a18-a13*a19) + a12 * (a8*a19  - a9*a18)  );

    det1  =  a1  * ( a22 * (a14*a18-a13*a19) + a12 * (a19*a23-a18*a24) + a17 * (a13*a24 - a14*a23) );
    det1 +=  a11 * ( a17 * (a4*a23 - a3*a24) + a2  * (a18*a24-a19*a23) + a22 * (a3*a19  - a4*a18)  );
    det1 +=  a16 * ( a12 * (a3*a24 - a4*a23) + a2  * (a14*a23-a13*a24) + a22 * (a4*a13  - a3*a14)  );
    det1 +=  a21 * ( a2  * (a13*a19-a14*a18) + a12 * (a4*a18 -a3*a19 ) + a17 * (a3*a14  - a4*a13)  );

    det2  =  a1  * ( a7 * (a18*a24-a19*a23) + a17 * (a9*a23-a8*a24) + a22 * (a8*a19 - a9*a18) );
    det2 +=  a6  * ( a2 * (a19*a23-a18*a24) + a17 * (a3*a24-a4*a23) + a22 * (a4*a18 - a3*a19) );
    det2 +=  a16 * ( a2 * (a8*a24 -a9*a23 ) + a7  * (a4*a23-a3*a24) + a22 * (a3*a9  - a4*a8)  );
    det2 +=  a21 * ( a7 * (a3*a19 -a4*a18 ) + a2  * (a9*a18-a8*a19) + a17 * (a4*a8  - a3*a9)  );

    det3  =  a1  * ( a12* (a8*a24 -a9*a23)  + a7  * (a14*a23-a13*a24) + a22 * (a9*a13 - a8*a14) );
    det3 +=  a6  * ( a2 * (a13*a24-a14*a23) + a12 * (a4*a23 -a3*a24 ) + a22 * (a3*a14 - a4*a13) );
    det3 +=  a11 * ( a7 * (a3*a24 -a4*a23)  + a2  * (a9*a23 -a8*a24 ) + a22 * (a4*a8  - a3*a9)  );
    det3 +=  a21 * ( a2 * (a8*a14 -a9*a13)  + a7  * (a4*a13 -a3*a14 ) + a12 * (a3*a9  - a4*a8)  );

    det4  =  a1  * ( a7 * (a13*a19-a14*a18) + a12 * (a9*a18 -a8*a19 ) + a17 * (a8*a14 - a9*a13) );
    det4 +=  a6  * ( a12* (a3*a19 -a4*a18 ) + a17 * (a4*a13 -a3*a14 ) + a2  * (a14*a18- a13*a19));
    det4 +=  a11 * ( a2 * (a8*a19 -a9*a18 ) + a7  * (a4*a18 -a3*a19 ) + a17 * (a3*a9  - a4*a8)  );
    det4 +=  a16 * ( a7 * (a3*a14 -a4*a13 ) + a2  * (a9*a13 -a8*a14 ) + a12 * (a4*a8  - a3*a9)  );

    det = det0*a0 + det1*a5+ det2*a10 + det3*a15 + det4*a20;

    if ( ABS(det) < SMALLREAL ) {
        printf("### WARNING: Matrix is nearly singular, det = %e!  Ignore.\n", det);
        printf("##----------------------------------------------\n");
        printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n", a0,  a1,  a2,  a3,  a4);
        printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n", a5,  a6,  a7,  a8,  a9);
        printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n", a10, a11, a12, a13, a14);
        printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n", a15, a16, a17, a18, a19);
        printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n", a20, a21, a22, a23, a24);
        printf("##----------------------------------------------\n");

        a[0]  = 1.0;  a[1]  = 0.0;  a[2]  = 0.0;  a[3]  = 0.0;  a[4]  = 0.0;
        a[5]  = 0.0;  a[6]  = 1.0;  a[7]  = 0.0;  a[8]  = 0.0;  a[9]  = 0.0;
        a[10] = 0.0;  a[11] = 0.0;  a[12] = 1.0;  a[13] = 0.0;  a[14] = 0.0;
        a[15] = 0.0;  a[16] = 0.0;  a[17] = 0.0;  a[18] = 1.0;  a[19] = 0.0;
        a[20] = 0.0;  a[21] = 0.0;  a[22] = 0.0;  a[23] = 0.0;  a[24] = 1.0;
    }
    else {
        REAL det_inv = 1 / det;

        a[0] = a6 * (a12 * a18 * a24 - a12 * a19 * a23 - a17 * a13 * a24 + a17 * a14 * a23 + a22 * a13 * a19 - a22 * a14 * a18);
        a[0] += a11 * (a7 * a19 * a23 - a7 * a18 * a24 + a17 * a8 * a24 - a17 * a9 * a23 - a22 * a8 * a19 + a22 * a9 * a18);
        a[0] += a16 * (a7 * a13 * a24 - a7 * a14 * a23 - a12 * a8 * a24 + a12 * a9 * a23 + a22 * a8 * a14 - a22 * a9 * a13);
        a[0] += a21 * (a7 * a14 * a18 - a7 * a13 * a19 + a12 * a8 * a19 - a12 * a9 * a18 - a17 * a8 * a14 + a17 * a9 * a13);
        a[0] *= det_inv;

        a[1] = a1 * (a12 * a19 * a23 - a12 * a18 * a24 + a22 * a14 * a18 - a17 * a14 * a23 - a22 * a13 * a19 + a17 * a13 * a24);
        a[1] += a11 * (a22 * a3 * a19 + a2 * a18 * a24 - a17 * a3 * a24 - a22 * a4 * a18 - a2 * a19 * a23 + a17 * a4 * a23);
        a[1] += a16 * (a12 * a3 * a24 - a12 * a4 * a23 - a22 * a3 * a14 + a2 * a14 * a23 + a22 * a4 * a13 - a2 * a13 * a24);
        a[1] += a21 * (a12 * a4 * a18 - a12 * a3 * a19 - a2 * a14 * a18 - a17 * a4 * a13 + a2 * a13 * a19 + a17 * a3 * a14);
        a[1] *= det_inv;

        a[2] = a1 * (a7 * a18 * a24 - a7 * a19 * a23 - a17 * a8 * a24 + a17 * a9 * a23 + a22 * a8 * a19 - a22 * a9 * a18);
        a[2] += a6 * (a2 * a19 * a23 - a2 * a18 * a24 + a17 * a3 * a24 - a17 * a4 * a23 - a22 * a3 * a19 + a22 * a4 * a18);
        a[2] += a16 * (a2 * a8 * a24 - a2 * a9 * a23 - a7 * a3 * a24 + a7 * a4 * a23 + a22 * a3 * a9 - a22 * a4 * a8);
        a[2] += a21 * (a2 * a9 * a18 - a2 * a8 * a19 + a7 * a3 * a19 - a7 * a4 * a18 - a17 * a3 * a9 + a17 * a4 * a8);
        a[2] *= det_inv;

        a[3] = a1 * (a12 * a8 * a24 - a12 * a9 * a23 + a7 * a14 * a23 - a7 * a13 * a24 + a22 * a9 * a13 - a22 * a8 * a14);
        a[3] += a6 * (a12 * a4 * a23 - a12 * a3 * a24 + a22 * a3 * a14 - a22 * a4 * a13 + a2 * a13 * a24 - a2 * a14 * a23);
        a[3] += a11 * (a7 * a3 * a24 - a7 * a4 * a23 + a22 * a4 * a8 - a22 * a3 * a9 + a2 * a9 * a23 - a2 * a8 * a24);
        a[3] += a21 * (a12 * a3 * a9 - a12 * a4 * a8 + a2 * a8 * a14 - a2 * a9 * a13 + a7 * a4 * a13 - a7 * a3 * a14);
        a[3] *= det_inv;

        a[4] = a1 * (a7 * a13 * a19 - a7 * a14 * a18 - a12 * a8 * a19 + a12 * a9 * a18 + a17 * a8 * a14 - a17 * a9 * a13);
        a[4] += a6 * (a2 * a14 * a18 - a2 * a13 * a19 + a12 * a3 * a19 - a12 * a4 * a18 - a17 * a3 * a14 + a17 * a4 * a13);
        a[4] += a11 * (a2 * a8 * a19 - a2 * a9 * a18 - a7 * a3 * a19 + a7 * a4 * a18 + a17 * a3 * a9 - a17 * a4 * a8);
        a[4] += a16 * (a2 * a9 * a13 - a2 * a8 * a14 + a7 * a3 * a14 - a7 * a4 * a13 - a12 * a3 * a9 + a12 * a4 * a8);
        a[4] *= det_inv;

        a[5] = a5 * (a12 * a19 * a23 - a12 * a18 * a24 + a22 * a14 * a18 - a22 * a13 * a19 + a17 * a13 * a24 - a17 * a14 * a23);
        a[5] += a20 * (a12 * a9 * a18 - a12 * a8 * a19 + a7 * a13 * a19 - a18 * a7 * a14 + a17 * a8 * a14 - a9 * a17 * a13);
        a[5] += a15 * (a22 * a9 * a13 - a12 * a9 * a23 + a12 * a24 * a8 + a7 * a14 * a23 - a24 * a7 * a13 - a22 * a14 * a8);
        a[5] += a10 * (a18 * a7 * a24 - a18 * a22 * a9 - a17 * a8 * a24 + a17 * a9 * a23 + a22 * a8 * a19 - a19 * a23 * a7);
        a[5] *= det_inv;

        a[6] = a2 * (a19 * a23 * a10 - a14 * a23 * a15 - a18 * a24 * a10 + a18 * a14 * a20 - a13 * a19 * a20 + a24 * a13 * a15);
        a[6] += a12 * (a18 * a0 * a24 - a18 * a20 * a4 + a3 * a19 * a20 - a19 * a23 * a0 + a4 * a23 * a15 - a24 * a15 * a3);
        a[6] += a17 * (a4 * a13 * a20 - a13 * a24 * a0 + a14 * a23 * a0 - a3 * a14 * a20 + a24 * a3 * a10 - a4 * a23 * a10);
        a[6] += a22 * (a14 * a15 * a3 - a18 * a14 * a0 + a18 * a4 * a10 - a4 * a13 * a15 + a13 * a19 * a0 - a3 * a19 * a10);
        a[6] *= det_inv;

        a[7] = a0 * (a18 * a9 * a22 - a18 * a24 * a7 + a19 * a23 * a7 - a9 * a23 * a17 + a24 * a8 * a17 - a8 * a19 * a22);
        a[7] += a5 * (a2 * a18 * a24 - a2 * a19 * a23 + a17 * a4 * a23 - a17 * a3 * a24 + a22 * a3 * a19 - a22 * a4 * a18);
        a[7] += a15 * (a4 * a8 * a22 - a3 * a9 * a22 - a24 * a8 * a2 + a9 * a23 * a2 - a4 * a23 * a7 + a24 * a3 * a7);
        a[7] += a20 * (a18 * a4 * a7 - a18 * a9 * a2 + a9 * a3 * a17 - a4 * a8 * a17 + a8 * a19 * a2 - a3 * a19 * a7);
        a[7] *= det_inv;

        a[8] = a0 * (a12 * a9 * a23 - a12 * a24 * a8 + a22 * a14 * a8 - a7 * a14 * a23 + a24 * a7 * a13 - a9 * a22 * a13);
        a[8] += a5 * (a12 * a3 * a24 - a12 * a4 * a23 - a22 * a3 * a14 + a2 * a14 * a23 - a2 * a13 * a24 + a22 * a4 * a13);
        a[8] += a10 * (a22 * a9 * a3 - a4 * a22 * a8 + a4 * a7 * a23 - a2 * a9 * a23 + a24 * a2 * a8 - a7 * a24 * a3);
        a[8] += a20 * (a7 * a14 * a3 - a4 * a7 * a13 + a9 * a2 * a13 + a12 * a4 * a8 - a12 * a9 * a3 - a2 * a14 * a8);
        a[8] *= det_inv;

        a[9] = a0 * (a12 * a8 * a19 - a12 * a18 * a9 + a18 * a7 * a14 - a8 * a17 * a14 + a17 * a13 * a9 - a7 * a13 * a19);
        a[9] += a5 * (a2 * a13 * a19 - a2 * a14 * a18 - a12 * a3 * a19 + a12 * a4 * a18 + a17 * a3 * a14 - a17 * a4 * a13);
        a[9] += a10 * (a18 * a2 * a9 - a18 * a7 * a4 + a3 * a7 * a19 - a2 * a8 * a19 + a17 * a8 * a4 - a3 * a17 * a9);
        a[9] += a15 * (a8 * a2 * a14 - a12 * a8 * a4 + a12 * a3 * a9 - a3 * a7 * a14 + a7 * a13 * a4 - a2 * a13 * a9);
        a[9] *= det_inv;

        a[10] = a5 * (a18 * a24 * a11 - a24 * a13 * a16 + a14 * a23 * a16 - a19 * a23 * a11 + a13 * a19 * a21 - a18 * a14 * a21);
        a[10] += a10 * (a19 * a23 * a6 - a9 * a23 * a16 + a24 * a8 * a16 - a8 * a19 * a21 + a18 * a9 * a21 - a18 * a24 * a6);
        a[10] += a15 * (a24 * a13 * a6 - a14 * a23 * a6 - a24 * a8 * a11 + a9 * a23 * a11 + a14 * a8 * a21 - a13 * a9 * a21);
        a[10] += a20 * (a18 * a14 * a6 - a18 * a9 * a11 + a8 * a19 * a11 - a13 * a19 * a6 + a9 * a13 * a16 - a14 * a8 * a16);
        a[10] *= det_inv;

        a[11] = a4 * (a21 * a13 * a15 - a11 * a23 * a15 + a16 * a23 * a10 - a13 * a16 * a20 + a18 * a11 * a20 - a18 * a21 * a10);
        a[11] += a14 * (a18 * a0 * a21 - a1 * a18 * a20 + a16 * a3 * a20 - a23 * a0 * a16 + a1 * a23 * a15 - a21 * a3 * a15);
        a[11] += a19 * (a1 * a13 * a20 - a1 * a23 * a10 + a23 * a0 * a11 + a21 * a3 * a10 - a11 * a3 * a20 - a13 * a0 * a21);
        a[11] += a24 * (a13 * a0 * a16 - a18 * a0 * a11 + a11 * a3 * a15 + a1 * a18 * a10 - a1 * a13 * a15 - a16 * a3 * a10);
        a[11] *= det_inv;

        a[12] = a4 * (a5 * a21 * a18 - a18 * a20 * a6 + a20 * a16 * a8 - a5 * a16 * a23 + a15 * a6 * a23 - a21 * a15 * a8);
        a[12] += a9 * (a1 * a20 * a18 - a1 * a15 * a23 + a0 * a16 * a23 - a18 * a0 * a21 - a20 * a16 * a3 + a15 * a21 * a3);
        a[12] += a19 * (a20 * a6 * a3 - a5 * a21 * a3 + a0 * a21 * a8 - a23 * a0 * a6 + a1 * a5 * a23 - a1 * a20 * a8);
        a[12] += a24 * (a1 * a15 * a8 - a0 * a16 * a8 + a18 * a0 * a6 - a1 * a5 * a18 + a5 * a16 * a3 - a6
* a15 * a3);
        a[12] *= det_inv;

        a[13] = a0 * (a24 * a11 * a8 - a6 * a24 * a13 + a21 * a9 * a13 - a11 * a9 * a23 + a14 * a6 * a23 - a14 * a21 * a8);
        a[13] += a1 * (a5 * a13 * a24 - a5 * a14 * a23 + a14 * a20 * a8 + a10 * a9 * a23 - a24 * a10 * a8 - a20 * a9 * a13);
        a[13] += a3 * (a6 * a10 * a24 - a10 * a9 * a21 + a5 * a14 * a21 - a5 * a24 * a11 + a20 * a9 * a11 - a14 * a6 * a20);
        a[13] += a4 * (a5 * a11 * a23 - a5 * a21 * a13 + a21 * a10 * a8 - a6 * a10 * a23 + a20 * a6 * a13 - a11 * a20 * a8);
        a[13] *= det_inv;

        a[14] = a0 * (a13 * a19 * a6 - a14 * a18 * a6 - a11 * a19 * a8 + a14 * a16 * a8 + a11 * a18 * a9 - a13 * a16 * a9);
        a[14] += a1 * (a14 * a18 * a5 - a13 * a19 * a5 + a10 * a19 * a8 - a14 * a15 * a8 - a10 * a18 * a9 + a13 * a15 * a9);
        a[14] += a3 * (a11 * a19 * a5 - a11 * a15 * a9 + a10 * a16 * a9 - a10 * a19 * a6 + a14 * a15 * a6 - a14 * a16 * a5);
        a[14] += a4 * (a11 * a15 * a8 - a11 * a18 * a5 + a13 * a16 * a5 - a13 * a15 * a6 + a10 * a18 * a6 - a10 * a16 * a8);
        a[14] *= det_inv;

        a[15] = a5 * (a19 * a22 * a11 - a24 * a17 * a11 + a12 * a24 * a16 - a22 * a14 * a16 - a12 * a19 * a21 + a17 * a14 * a21);
        a[15] += a10 * (a24 * a17 * a6 - a19 * a22 * a6 - a24 * a7 * a16 + a22 * a9 * a16 + a19 * a7 * a21 - a17 * a9 * a21);
        a[15] += a15 * (a22 * a14 * a6 - a9 * a22 * a11 + a24 * a7 * a11 - a12 * a24 * a6 - a7 * a14 * a21 + a12 * a9 * a21);
        a[15] += a20 * (a12 * a19 * a6 - a17 * a14 * a6 - a19 * a7 * a11 + a9 * a17 * a11 + a7 * a14 * a16 - a12 * a9 * a16);
        a[15] *= det_inv;

        a[16] = a0 * (a11 * a17 * a24 - a11 * a19 * a22 - a12 * a16 * a24 + a12 * a19 * a21 + a14 * a16 * a22 - a14 * a17 * a21);
        a[16] += a1 * (a10 * a19 * a22 - a10 * a17 * a24 + a12 * a15 * a24 - a12 * a19 * a20 - a14 * a15 * a22 + a14 * a17 * a20);
        a[16] += a2 * (a10 * a16 * a24 - a10 * a19 * a21 - a11 * a15 * a24 + a11 * a19 * a20 + a14 * a15 * a21 - a14 * a16 * a20);
        a[16] += a4 * (a10 * a17 * a21 * +a11 * a15 * a22 - a11 * a17 * a20 - a12 * a15 * a21 + a12 * a16 * a20 - a10 * a16 * a22);
        a[16] *= det_inv;

        a[17] = a0 * (a21 * a9 * a17 - a6 * a24 * a17 + a19 * a6 * a22 - a0 * a16 * a9 * a22 + a24 * a16 * a7 - a19 * a21 * a7);
        a[17] += a1 * (a5 * a24 * a17 - a5 * a19 * a22 + a19 * a20 * a7 - a20 * a9 * a17 + a15 * a9 * a22 - a24 * a15 * a7);
        a[17] += a2 * (a5 * a19 * a21 - a19 * a6 * a20 - a5 * a24 * a16 + a24 * a6 * a15 - a15 * a9 * a21 + a20 * a9 * a16);
        a[17] += a4 * (a16 * a5 * a22 - a6 * a15 * a22 + a20 * a6 * a17 - a5 * a21 * a17 - a6 * a15 * a22 + a21 * a15 * a7 - a16 * a20 * a7);
        a[17] *= det_inv;

        a[18] = a0 * (a12 * a24 * a6 - a14 * a22 * a6 - a11 * a24 * a7 + a14 * a21 * a7 + a11 * a22 * a9 - a12 * a21 * a9);
        a[18] += a1 * (a14 * a22 * a5 - a12 * a24 * a5 + a10 * a24 * a7 - a14 * a20 * a7 - a10 * a22 * a9 + a12 * a20 * a9);
        a[18] += a2 * (a11 * a24 * a5 - a11 * a20 * a9 + a14 * a20 * a6 - a14 * a21 * a5 + a10 * a21 * a9 - a10 * a24 * a6);
        a[18] += a4 * (a11 * a20 * a7 - a11 * a22 * a5 + a12 * a21 * a5 + a10 * a22 * a6 - a12 * a20 * a6 - a10 * a21 * a7);
        a[18] *= det_inv;

        a[19] = a0 * (a12 * a16 * a9 - a6 * a12 * a19 + a6 * a17 * a14 - a17 * a11 * a9 + a11 * a7 * a19 - a16 * a7 * a14);
        a[19] += a1 * (a5 * a12 * a19 - a5 * a17 * a14 - a12 * a15 * a9 + a17 * a10 * a9 + a15 * a7 * a14 - a10 * a7 * a19);
        a[19] += a2 * (a11 * a15 * a9 - a5 * a11 * a19 + a5 * a16 * a14 - a6 * a15 * a14 + a6 * a10 * a19 - a16 * a10 * a9);
        a[19] += a4 * (a5 * a17 * a11 - a5 * a12 * a16 + a12 * a6 * a15 + a10 * a7 * a16 - a17 * a6 * a10 - a15 * a7 * a11);
        a[19] *= det_inv;

        a[20] = a5 * (a12 * a18 * a21 - a12 * a23 * a16 + a22 * a13 * a16 - a18 * a22 * a11 + a23 * a17 * a11 - a17 * a13 * a21);
        a[20] += a15 * (a12 * a23 * a6 - a12 * a8 * a21 + a8 * a22 * a11 - a23 * a7 * a11 + a7 * a13 * a21 - a22 * a13 * a6);
        a[20] += a20 * (a12 * a8 * a16 - a12 * a18 * a6 + a18 * a7 * a11 - a8 * a17 * a11 + a17 * a13 * a6 - a7 * a13 * a16);
        a[20] += a10 * (a17 * a8 * a21 - a22 * a8 * a16 - a18 * a7 * a21 + a18 * a22 * a6 + a23 * a7 * a16 - a23 * a17 * a6);
        a[20] *= det_inv;

        a[21] = a0 * (a12 * a23 * a16 - a12 * a18 * a21 + a17 * a13 * a21 + a18 * a22 * a11 - a23 * a17 * a11 - a22 * a13 * a16);
        a[21] += a1 * (a12 * a18 * a20 - a12 * a23 * a15 + a22 * a13 * a15 + a23 * a17 * a10 - a17 * a13 * a20 - a18 * a22 * a10);
        a[21] += a2 * (a18 * a21 * a10 - a18 * a11 * a20 - a21 * a13 * a15 + a16 * a13 * a20 - a23 * a16 * a10 + a23 * a11 * a15);
        a[21] += a3 * (a17 * a11 * a20 - a12 * a16 * a20 + a12 * a21 * a15 - a21 * a17 * a10 - a22 * a11 * a15 + a16 * a22 * a10);
        a[21] *= det_inv;

        a[22] = a0 * (a18 * a21 * a7 - a18 * a6 * a22 + a23 * a6 * a17 + a16 * a8 * a22 - a21 * a8 * a17 - a23 * a16 * a7);
        a[22] += a1 * (a5 * a18 * a22 - a5 * a23 * a17 - a15 * a8 * a22 + a20 * a8 * a17 - a18 * a20 * a7 + a23 * a15 * a7);
        a[22] += a3 * (a16 * a20 * a7 + a6 * a15 * a22 - a6 * a20 * a17 - a5 * a16 * a22 + a5 * a21 * a17 - a21 * a15 * a7);
        a[22] += a2 * (a5 * a23 * a16 - a5 * a18 * a21 + a18 * a6 * a20 + a15 * a8 * a21 - a20 * a8 * a16 - a23 * a6 * a15);
        a[22] *= det_inv;

        a[23] = a0 * (a12 * a21 * a8 - a22 * a11 * a8 + a11 * a7 * a23 - a6 * a12 * a23 - a21 * a7 * a13 + a6 * a22 * a13);
        a[23] += a1 * (a5 * a12 * a23 - a5 * a22 * a13 - a10 * a7 * a23 + a20 * a7 * a13 + a22 * a10 * a8 - a12 * a20 * a8);
        a[23] += a2 * (a5 * a21 * a13 + a11 * a20 * a8 + a6 * a10 * a23 - a5 * a11 * a23 - a21 * a10 * a8 - a6 * a20 * a13);
        a[23] += a3 * (a5 * a22 * a11 - a5 * a12 * a21 + a10 * a7 * a21 - a22 * a6 * a10 - a20 * a7 * a11 + a12 * a6 * a20);
        a[23] *= det_inv;

        a[24] = a0 * (a17 * a11 * a8 - a11 * a7 * a18 + a6 * a12 * a18 - a12 * a16 * a8 + a16 * a7 * a13 - a6 * a17 * a13);
        a[24] += a1 * (a5 * a17 * a13 - a5 * a12 * a18 + a10 * a7 * a18 + a12 * a15 * a8 - a17 * a10 * a8 - a15 * a7 * a13);
        a[24] += a2 * (a5 * a11 * a18 - a5 * a16 * a13 + a16 * a10 * a8 + a6 * a15 * a13 - a11 * a15 * a8 - a6 * a10 * a18);
        a[24] += a3 * (a5 * a12 * a16 + a17 * a6 * a10 - a5 * a17 * a11 - a12 * a6 * a15 - a10 * a7 * a16 + a15 * a7 * a11);
        a[24] *= det_inv;

    }

    printf("### DEBUG: Check inverse matrix...\n");
    printf("##----------------------------------------------\n");
    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
        a0 * a[0] + a1 * a[5] + a2 * a[10] + a3 * a[15] + a4 * a[20],
        a0 * a[1] + a1 * a[6] + a2 * a[11] + a3 * a[16] + a4 * a[21],
        a0 * a[2] + a1 * a[7] + a2 * a[12] + a3 * a[17] + a4 * a[22],
        a0 * a[3] + a1 * a[8] + a2 * a[13] + a3 * a[18] + a4 * a[23],
        a0 * a[4] + a1 * a[9] + a2 * a[14] + a3 * a[19] + a4 * a[24]);
    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
        a5 * a[0] + a6 * a[5] + a7 * a[10] + a8 * a[15] + a9 * a[20],
        a5 * a[1] + a6 * a[6] + a7 * a[11] + a8 * a[16] + a9 * a[21],
        a5 * a[2] + a6 * a[7] + a7 * a[12] + a8 * a[17] + a9 * a[22],
        a5 * a[3] + a6 * a[8] + a7 * a[13] + a8 * a[18] + a9 * a[23],
        a5 * a[4] + a6 * a[9] + a7 * a[14] + a8 * a[19] + a9 * a[24]);
    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
        a10 * a[0] + a11 * a[5] + a12 * a[10] + a13 * a[15] + a14 * a[20],
        a10 * a[1] + a11 * a[6] + a12 * a[11] + a13 * a[16] + a14 * a[21],
        a10 * a[2] + a11 * a[7] + a12 * a[12] + a13 * a[17] + a14 * a[22],
        a10 * a[3] + a11 * a[8] + a12 * a[13] + a13 * a[18] + a14 * a[23],
        a10 * a[4] + a11 * a[9] + a12 * a[14] + a13 * a[19] + a14 * a[24]);
    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
        a15 * a[0] + a16 * a[5] + a17 * a[10] + a18 * a[15] + a19 * a[20],
        a15 * a[1] + a16 * a[6] + a17 * a[11] + a18 * a[16] + a19 * a[21],
        a15 * a[2] + a16 * a[7] + a17 * a[12] + a18 * a[17] + a19 * a[22],
        a15 * a[3] + a16 * a[8] + a17 * a[13] + a18 * a[18] + a19 * a[23],
        a15 * a[4] + a16 * a[9] + a17 * a[14] + a18 * a[19] + a19 * a[24]);
    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
        a20 * a[0] + a21 * a[5] + a22 * a[10] + a23 * a[15] + a24 * a[20],
        a20 * a[1] + a21 * a[6] + a22 * a[11] + a23 * a[16] + a24 * a[21],
        a20 * a[2] + a21 * a[7] + a22 * a[12] + a23 * a[17] + a24 * a[22],
        a20 * a[3] + a21 * a[8] + a22 * a[13] + a23 * a[18] + a24 * a[23],
        a20 * a[4] + a21 * a[9] + a22 * a[14] + a23 * a[19] + a24 * a[24]);
    printf("##----------------------------------------------\n");
}

void fasp_smat_inv_nc7 (REAL *a)
{
    fasp_smat_invp_nc(a,7);
}

void fasp_smat_inv_nc (REAL      *a,
                       const INT  n)
{
    INT i,j,k,l,u,kn,in;
    REAL alinv;

    for (k=0; k<n; ++k) {

        kn = k*n;
        l  = kn+k;

        if (ABS(a[l]) < SMALLREAL) {
            printf("### ERROR: Diagonal entry is close to zero!  ");
            printf("diag_%d = %.2e!  [%s]\n", k, a[l], __FUNCTION__);
            exit(ERROR_SOLVER_EXIT);
        }
        alinv = 1.0/a[l];
        a[l] = alinv;

        for (j=0; j<k; ++j) {
            u = kn+j; a[u] *= alinv;
        }

        for (j=k+1; j<n; ++j) {
            u = kn+j; a[u] *= alinv;
        }

        for (i=0; i<k; ++i) {
            in = i*n;
            for (j=0; j<n; ++j)
                if (j!=k) {
                    u = in+j; a[u] -= a[in+k]*a[kn+j];
                } // end if (j!=k)
        }

        for (i=k+1; i<n; ++i) {
            in = i*n;
            for (j=0; j<n; ++j)
                if (j!=k) {
                    u = in+j; a[u] -= a[in+k]*a[kn+j];
                } // end if (j!=k)
        }

        for (i=0; i<k; ++i) {
            u=i*n+k; a[u] *= -alinv;
        }

        for (i=k+1; i<n; ++i) {
            u=i*n+k; a[u] *= -alinv;
        }

    } // end for (k=0; k<n; ++k)
}

SHORT fasp_smat_invp_nc (REAL      *a,
                         const INT  n)
{
    INT   i, j, k, l, ll, u;
    INT   icol = 0, irow = 0;
    REAL  vmax, dum, pivinv, temp;

    INT *work  = (INT *)fasp_mem_calloc(3*n,sizeof(INT));
    INT *indxc = work, *indxr = work+n, *ipiv = work+2*n;

    // ipiv, indxr, and indxc are used for book-keeping on the pivoting.
    for ( j=0; j<n; j++ ) ipiv[j] = 0;

#if DEBUG_MODE > 1
    printf("### DEBUG: Matrix block\n");
    for ( i = 0; i < n; ++i ) {
        for ( j = 0; j < n; ++j ) {
            printf(" %10.5e,", a[i * n + j]);
        }
        printf("\n");
    }
#endif

    // This is the main loop over the columns to be reduced.
    for ( i=0; i<n; i++ ) {

        // This is the outer loop of the search for a pivot element.
        vmax = 0.0;
        for ( j=0; j<n; j++ ) {
            if ( ipiv[j] != 1 ) {
                for ( k=0; k<n; k++ ) {
                    if ( ipiv[k] == 0 ) {
                        u = j*n+k;
                        if ( ABS(a[u]) >= vmax ) {
                            vmax = ABS(a[u]); irow = j; icol = k;
                        }
                    }
                } // end for k
            }
        } // end for j

        ++(ipiv[icol]);

        // We now have the pivot element, so we interchange rows, if needed, to put
        // the pivot element on the diagonal.  The columns are not physically
        // interchanged, only relabeled:  indxc[i], the column of the ith pivot
        // element, is the ith column that is reduced, while indxr[i] is the row in
        // which that pivot element was originally located.  If indxr[i] != indxc[i]
        // there is an implied column interchange.  With this form of bookkeeping,
        // the inverse matrix will be scrambled by columns.
        if ( irow != icol ) {
            for ( l=0; l<n; l++ ) SWAP(a[irow*n+l],a[icol*n+l]);
        }

        indxr[i] = irow; indxc[i] = icol;
        u = icol*n+icol;
        if ( ABS(a[u]) < SMALLREAL ) {
            printf("### WARNING: The matrix is nearly singular!\n");
            return ERROR_SOLVER_EXIT;
        }
        pivinv = 1.0/a[u]; a[u]=1.0;
        for ( l=0; l<n; l++ ) a[icol*n+l] *= pivinv;

        for ( ll=0; ll<n; ll++ ) {
            if ( ll != icol ) {
                u = ll*n+icol;
                dum = a[u]; a[u] = 0.0;
                for ( l=0; l<n; l++ ) a[ll*n+l] -= a[icol*n+l]*dum;
            }
        }
    }
    // This is the end of the main loop over columns of the reduction.

    // It only remains to unscramble the matrix in view of the column interchanges.
    for ( l=n-1; l>=0; l-- ) {
        if ( indxr[l] != indxc[l] )
            for ( k=0; k<n; k++ ) SWAP(a[k*n+indxr[l]],a[k*n+indxc[l]]);
    } // And we are done.

    fasp_mem_free(work); work = NULL;

    return FASP_SUCCESS;
}

SHORT fasp_smat_inv (REAL      *a,
                     const INT  n)
{
    SHORT status = FASP_SUCCESS;

    switch (n) {

        case 2:
            fasp_smat_inv_nc2(a);
            break;

        case 3:
            fasp_smat_inv_nc3(a);
            break;

        case 4:
            fasp_smat_inv_nc4(a);
            break;

        case -5:
            fasp_smat_inv_nc5(a);
            break;

        default:
            status = fasp_smat_invp_nc(a,n);
            break;

    }

    return status;
}

REAL fasp_smat_Linf (const REAL  *A,
                     const INT    n)
{

    REAL norm = 0.0, value;

    INT i,j;

    for ( i = 0; i < n; i++ ) {
        for ( value = 0.0, j = 0; j < n; j++ ) {
            value = value + ABS(A[i*n+j]);
        }
        norm = MAX(norm, value);
    }

    return norm;
}

void fasp_smat_identity_nc2 (REAL *a)
{
    memset(a, 0X0, 4*sizeof(REAL));

    a[0] = 1.0; a[3] = 1.0;
}

void fasp_smat_identity_nc3 (REAL *a)
{
    memset(a, 0X0, 9*sizeof(REAL));

    a[0] = 1.0; a[4] = 1.0; a[8] = 1.0;
}

void fasp_smat_identity_nc5 (REAL *a)
{
    memset(a, 0X0, 25*sizeof(REAL));

    a[0]  = 1.0;
    a[6]  = 1.0;
    a[12] = 1.0;
    a[18] = 1.0;
    a[24] = 1.0;
}

void fasp_smat_identity_nc7 (REAL *a)
{
    memset(a, 0X0, 49*sizeof(REAL));

    a[0]  = 1.0;
    a[8]  = 1.0;
    a[16] = 1.0;
    a[24] = 1.0;
    a[32] = 1.0;
    a[40] = 1.0;
    a[48] = 1.0;
}

void fasp_smat_identity (REAL      *a,
                         const INT  n,
                         const INT  n2)
{
    memset(a, 0X0, n2*sizeof(REAL));

    switch (n) {

        case 2:  {
            a[0] = 1.0;
            a[3] = 1.0;
        }
            break;

        case 3:  {
            a[0] = 1.0;
            a[4] = 1.0;
            a[8] = 1.0;
        }
            break;

        case 4:  {
            a[0] = 1.0;
            a[5] = 1.0;
            a[10] = 1.0;
            a[15] = 1.0;
        }
            break;

        case 5:  {
            a[0] = 1.0;
            a[6] = 1.0;
            a[12] = 1.0;
            a[18] = 1.0;
            a[24] = 1.0;
        }
            break;

        case 6:  {
            a[0] = 1.0;
            a[7] = 1.0;
            a[14] = 1.0;
            a[21] = 1.0;
            a[28] = 1.0;
            a[35] = 1.0;
        }
            break;

        case 7:  {
            a[0] = 1.0;
            a[8] = 1.0;
            a[16] = 1.0;
            a[24] = 1.0;
            a[32] = 1.0;
            a[40] = 1.0;
            a[48] = 1.0;
        }
            break;

        default:  {
            INT l;
            for (l = 0; l < n; l ++) a[l*n+l] = 1.0;
        }
            break;
    }

}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

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

A	Pointer to the full matrix
pivot	Pivoting positions
n	Size of matrix A

Returns

FASP_SUCCESS if successed; otherwise, error information.

Note

Use Doolittle's method to decompose the n x 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, ..., n - 1 evaluate in order the following pair of expressions U[k][j] = A[k][j] - (L[k][0]∗U[0][j] + ... + L[k][k-1]∗U[k-1][j]) for j = k, k+1, ... , n-1 L[i][k] = (A[i][k] - (L[i][0]∗U[0][k] + . + L[i][k-1]∗U[k-1][k])) / U[k][k] for i = k+1, ... , 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

A	Pointer to the full matrix
b	Right hand side array (b is used as the working array!!!)
pivot	Pivoting positions
x	Pointer to the solution array
n	Size of matrix A

Returns

FASP_SUCCESS if successed; 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.

```

#include <math.h>

#include "fasp.h"

/*----------------------------------*/
/*--      Public Functions        --*/
/*----------------------------------*/

SHORT fasp_smat_lu_decomp (REAL       *A,
                           INT         pivot[],
                           const INT   n)
{
    INT i, j, k;
    REAL *p_k=NULL, *p_row=NULL, *p_col=NULL;
    REAL max;

    /* For each row and column, k = 0, ..., n-1, */
    for (k = 0, p_k = A; k < n; p_k += n, k++) {

        // find the pivot row
        pivot[k] = k;
        max = fabs( *(p_k + k) );
        for (j = k + 1, p_row = p_k + n; j < n; ++j, p_row += n) {
            if ( max < fabs(*(p_row + k)) ) {
                max = fabs(*(p_row + k));
                pivot[k] = j;
                p_col = p_row;
            }
        }

        // if the pivot row differs from the current row, interchange the two rows.
        if (pivot[k] != k)
            for (j = 0; j < n; ++j) {
                max = *(p_k + j);
                *(p_k + j) = *(p_col + j);
                *(p_col + j) = max;
            }

        // if the matrix is singular, return error
        if ( fabs( *(p_k + k) ) < SMALLREAL ) return -1;

        // otherwise find the lower triangular matrix elements for column k.
        for (i = k+1, p_row = p_k + n; i < n; p_row += n, ++i) {
            *(p_row + k) /= *(p_k + k);
        }

        // update remaining matrix
        for (i = k+1, p_row = p_k + n; i < n; p_row += n, ++i)
            for (j = k+1; j < n; ++j)
                *(p_row + j) -= *(p_row + k) * *(p_k + j);

    }

    return FASP_SUCCESS;
}

SHORT fasp_smat_lu_solve (const REAL  *A,
                          REAL         b[],
                          const INT    pivot[],
                          REAL         x[],
                          const INT    n)
{
    INT         i, k;
    REAL        dum;
    const REAL *p_k;

    /* solve Ly = b    */
    for (k = 0, p_k = A; k < n; p_k += n, k++) {
        if (pivot[k] != k) {dum = b[k]; b[k] = b[pivot[k]]; b[pivot[k]] = dum; }
        x[k] = b[k];
        for (i = 0; i < k; ++i) x[k] -= x[i] * *(p_k + i);
    }

    /* solve Ux = y */
    for (k = n-1, p_k = A + n*(n-1); k >= 0; k--, p_k -= n) {
        if (pivot[k] != k) {dum = b[k]; b[k] = b[pivot[k]]; b[pivot[k]] = dum; }
        for (i = k + 1; i < n; ++i) x[k] -= x[i] * *(p_k + i);
        if (*(p_k + k) == 0.0) return -1;
        x[k] /= *(p_k + k);
    }

    return FASP_SUCCESS;
}

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/

/*

//A simple test example can be written as the following
INT main (INT argc, const char * argv[])
{
REAL A[3][3] = {{0.0, 1.0, 4.0},
{4.0, 1.0, 0.0},
{1.0, 4.0, 1.0}};

REAL b[3] = {1, 1, 1}, x[3];
```

```

INT pivot[3];

INT ret, i, j;

ret = lu_decomp(&A[0][0], pivot, 3); // LU decomposition

ret = lu_solve(&A[0][0], b, pivot, x, 3); // Solve decomposed Ax=b

return 1;
}

*/
```

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

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

A	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_getdiaginv (const dBSRmat ∗A)

  Get D^{-1} of matrix A.
- dBSRmat fasp_dbsr_diaginv (const dBSRmat ∗A)

  Compute B := D^{-1}∗A, where 'D' is the block diagonal part of A.
- dBSRmat fasp_dbsr_diaginv2 (const dBSRmat ∗A, REAL ∗diaginv)

  Compute B := D^{-1}∗A, where 'D' is the block diagonal part of A.
- dBSRmat fasp_dbsr_diaginv3 (const dBSRmat ∗A, REAL ∗diaginv)

  Compute B := D^{-1}∗A, where 'D' is the block diagonal part of A.
- dBSRmat fasp_dbsr_diaginv4 (const dBSRmat ∗A, REAL ∗diaginv)

  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 = diag(L^{-1}) and DU = 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 = diag(L^{-1}) and DU = diag(U^{-1}).
- dBSRmat fasp_dbsr_perm (const dBSRmat ∗A, const INT ∗P)

  Apply permutation of A, i.e. Aperm=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

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

ROW	Number of rows of block
COL	Number of columns of block
NNZ	Number of nonzero blocks
nb	Dimension of each block
storage_manner	Storage manner for each sub-block
A	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

A	Pointer to the dBSRmat matrix
B	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

ROW	Number of rows of block
COL	Number of columns of block
NNZ	Number of nonzero blocks
nb	Dimension of each block
storage_manner	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

A	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

A	Pointer to the dBSRmat matrix
diaginv	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

A	Pointer to the dBSRmat matrix
diaginv	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

A	Pointer to the dBSRmat matrix
diaginv	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 = diag(L^{-1}) and DU = diag(U^{-1}).

Parameters

A	Pointer to the dBSRmat matrix
DL	Pointer to the diag(L^{-1})
DU	Pointer to the 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 = diag(L^{-1}) and DU = diag(U^{-1}).

Parameters

A	Pointer to the dBSRmat matrix
DL	Pointer to the diag(L^{-1})
DU	Pointer to the 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

A	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

A	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

A	Pointer to dBSRmat BSR matrix
B	Pointer to dBSRmat BSR matrix
Is	Pointer to selected rows
Js	Pointer to selected columns
m	Number of selected rows
n	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

n	Number of blocks to get
A	Pointer to the 'dBSRmat' type matrix
diag	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

A	Pointer to the original dBSRmat matrix
P	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

A	Pointer to the dBSRmat matrix
AT	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.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

dBSRmat fasp_dbsr_create (const INT  ROW,
                          const INT  COL,
                          const INT  NNZ,
                          const INT  nb,
                          const INT  storage_manner)
{
    dBSRmat A;

    if ( ROW > 0 ) {
        A.IA = (INT*)fasp_mem_calloc(ROW+1, sizeof(INT));
    }
    else {
        A.IA = NULL;
    }

    if ( NNZ > 0 ) {
        A.JA = (INT*)fasp_mem_calloc(NNZ ,sizeof(INT));
    }
    else {
        A.JA = NULL;
    }

    if ( nb > 0 && NNZ > 0) {
        A.val = (REAL*)fasp_mem_calloc(NNZ*nb*nb, sizeof(REAL));
    }
    else {
        A.val = NULL;
    }

    A.storage_manner = storage_manner;
    A.ROW = ROW;
    A.COL = COL;
    A.NNZ = NNZ;
    A.nb  = nb;

    return A;
}

void fasp_dbsr_alloc (const INT  ROW,
                      const INT  COL,
                      const INT  NNZ,
                      const INT  nb,
                      const INT  storage_manner,
                      dBSRmat   *A)
{
    if ( ROW > 0 ) {
        A->IA = (INT*)fasp_mem_calloc(ROW+1, sizeof(INT));
    }
    else {
        A->IA = NULL;
    }

    if ( NNZ > 0 ) {
        A->JA = (INT*)fasp_mem_calloc(NNZ, sizeof(INT));
    }
    else {
        A->JA = NULL;
    }

    if ( nb > 0 ) {
        A->val = (REAL*)fasp_mem_calloc(NNZ*nb*nb, sizeof(REAL));
    }
    else {
        A->val = NULL;
    }

    A->storage_manner = storage_manner;
    A->ROW = ROW;
    A->COL = COL;
    A->NNZ = NNZ;
    A->nb  = nb;

    return;
}

void fasp_dbsr_free (dBSRmat *A)
{
    if (A==NULL) return;

    fasp_mem_free(A->IA);  A->IA  = NULL;
    fasp_mem_free(A->JA);  A->JA  = NULL;
    fasp_mem_free(A->val); A->val = NULL;

    A->ROW = 0;
    A->COL = 0;
    A->NNZ = 0;
    A->nb  = 0;
    A->storage_manner = 0;
}

void fasp_dbsr_cp (const dBSRmat *A,
                   dBSRmat       *B)
{
    B->ROW = A->ROW;
    B->COL = A->COL;
    B->NNZ = A->NNZ;
    B->nb  = A->nb;
    B->storage_manner = A->storage_manner;

    memcpy(B->IA,A->IA,(A->ROW+1)*sizeof(INT));
    memcpy(B->JA,A->JA,(A->NNZ)*sizeof(INT));
    memcpy(B->val,A->val,(A->NNZ)*(A->nb)*(A->nb)*sizeof(REAL));
}

INT fasp_dbsr_trans (const dBSRmat *A,
                     dBSRmat       *AT)
{
    const INT n = A->ROW, m = A->COL, nnz = A->NNZ, nb = A->nb;

    INT status = FASP_SUCCESS;
    INT i,j,k,p,inb,jnb,nb2;

    AT->ROW = m;
    AT->COL = n;
    AT->NNZ = nnz;
    AT->nb  = nb;
    AT->storage_manner = A->storage_manner;

    AT->IA  = (INT*)fasp_mem_calloc(m+1,sizeof(INT));
    AT->JA  = (INT*)fasp_mem_calloc(nnz,sizeof(INT));
    nb2     = nb*nb;

    if (A->val) {
        AT->val = (REAL*)fasp_mem_calloc(nnz*nb2,sizeof(REAL));
    }
    else {
        AT->val = NULL;
    }

    // first pass:  find the number of nonzeros in the first m-1 columns of A
    // Note:  these numbers are stored in the array AT.IA from 1 to m-1
    fasp_iarray_set(m+1, AT->IA, 0);

    for ( j=0; j<nnz; ++j ) {
        i=A->JA[j]; // column number of A = row number of A'
        if (i<m-1) AT->IA[i+2]++;
    }

    for ( i=2; i<=m; ++i ) AT->IA[i]+=AT->IA[i-1];

    // second pass:  form A'
    if ( A->val ) {
        for ( i=0; i<n; ++i ) {
            INT ibegin=A->IA[i], iend1=A->IA[i+1];
            for ( p=ibegin; p<iend1; p++ ) {
                j=A->JA[p]+1;
                k=AT->IA[j];
                AT->JA[k]=i;
                for ( inb=0; inb<nb; inb++ )
                    for ( jnb=0; jnb<nb; jnb++ )
                        AT->val[nb2*k + inb*nb + jnb] = A->val[nb2*p + jnb*nb + inb];
                AT->IA[j]=k+1;
            } // end for p
        } // end for i

    }
    else {
        for ( i=0; i<n; ++i ) {
            INT ibegin=A->IA[i], iend1=A->IA[i+1];
            for ( p=ibegin; p<iend1; p++ ) {
                j=A->JA[p]+1;
                k=AT->IA[j];
                AT->JA[k]=i;
                AT->IA[j]=k+1;
            } // end for p
        } // end of i

    } // end if

    return (status);
}

SHORT fasp_dbsr_getblk (const dBSRmat  *A,
                        const INT      *Is,
                        const INT      *Js,
                        const INT       m,
                        const INT       n,
                        dBSRmat        *B)
{
    INT   status = FASP_SUCCESS;
    INT   i,j,k,nnz=0;
    INT   *col_flag;
    SHORT use_openmp = FALSE;

    const INT nb = A->nb;
    const INT nb2=nb*nb;

#ifdef _OPENMP
    INT myid, mybegin, stride_i, myend, nthreads;
    if ( n > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // create colum flags
    col_flag = (INT*)fasp_mem_calloc(A->COL,sizeof(INT));

    B->ROW=m; B->COL=n; B->nb=nb; B->storage_manner=A->storage_manner;

    B->IA = (INT*)fasp_mem_calloc(m+1,sizeof(INT));

    if ( use_openmp ) {
#ifdef _OPENMP
        stride_i = n/nthreads;
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            mybegin = myid*stride_i;
            if ( myid < nthreads-1 )  myend = mybegin+stride_i;
            else myend = n;
            for ( i = mybegin; i < myend; ++i ) {
                col_flag[Js[i]]=i+1;
            }
        }
#endif
    }
    else {
        for ( i=0; i<n; ++i ) col_flag[Js[i]]=i+1;
    }

    // first pass:  count nonzeros for sub matrix
    B->IA[0] = 0;
    for ( i=0; i<m; ++i ) {
        for ( k=A->IA[Is[i]]; k<A->IA[Is[i]+1]; ++k ) {
            j=A->JA[k];
            if (col_flag[j]>0) nnz++;
        } /* end for k */
        B->IA[i+1] = nnz;
    } /* end for i */
    B->NNZ = nnz;

    // allocate
    B->JA  = (INT*)fasp_mem_calloc(nnz,sizeof(INT));
    B->val = (REAL*)fasp_mem_calloc(nnz*nb2,sizeof(REAL));

    // second pass:  copy data to B
    nnz = 0;
    for ( i=0; i<m; ++i)  {
        for ( k=A->IA[Is[i]]; k<A->IA[Is[i]+1]; ++k ) {
            j = A->JA[k];
            if ( col_flag[j] > 0 ) {
                B->JA[nnz]=col_flag[j]-1;
                memcpy(B->val+nnz*nb2, A->val+k*nb2, nb2*sizeof(REAL));
                nnz++;
            }
        } /* end for k */
    } /* end for i */

    fasp_mem_free(col_flag); col_flag = NULL;

    return(status);
}

SHORT fasp_dbsr_diagpref (dBSRmat *A)
{
    SHORT          status = FASP_SUCCESS;
    const INT      num_rowsA = A -> ROW;
    const INT      num_colsA = A -> COL;
    const INT      nb        = A->nb;
    const INT      nb2       = nb*nb;

    const INT     *A_i       = A -> IA;
    INT           *A_j       = A -> JA;
    REAL          *A_data    = A -> val;

    INT   i, j, tempi, row_size;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend, ibegin, iend;
    INT nthreads = fasp_get_num_threads();
#endif

    /* the matrix should be square */
    if (num_rowsA != num_colsA) return ERROR_INPUT_PAR;

#ifdef _OPENMP
    if (num_rowsA > OPENMP_HOLDS) {
        REAL *tempd = (REAL*)fasp_mem_calloc(nb2*nthreads, sizeof(REAL));
#pragma omp parallel for private (myid,mybegin,myend,i,j,tempi,ibegin,iend)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                ibegin = A_i[i+1]; iend = A_i[i];
                for (j = ibegin; j < iend; j ++) {
                    if (A_j[j] == i) {
                        if (j != ibegin) {
                            // swap index
                            tempi  = A_j[ibegin];
                            A_j[ibegin] = A_j[j];
                            A_j[j] = tempi;
                            // swap block
                            memcpy(tempd+myid*nb2,    A_data+ibegin*nb2, (nb2)*sizeof(REAL));
                            memcpy(A_data+ibegin*nb2, A_data+j*nb2,      (nb2)*sizeof(REAL));
                            memcpy(A_data+j*nb2,      tempd+myid*nb2,    (nb2)*sizeof(REAL));
                        }
                        break;
                    }
                    /* diagonal element is missing */
                    if (j == iend-1) {
                        status = -2;
                        break;
                    }
                }
            }
        }
        fasp_mem_free(tempd); tempd = NULL;
    }
    else {
#endif
        REAL *tempd = (REAL*)fasp_mem_calloc(nb2, sizeof(REAL));
        for (i = 0; i < num_rowsA; i ++) {
            row_size = A_i[i+1] - A_i[i];
            for (j = 0; j < row_size; j ++) {
                if (A_j[j] == i) {
                    if (j != 0) {
                        // swap index
                        tempi  = A_j[0];
                        A_j[0] = A_j[j];
                        A_j[j] = tempi;
                        // swap block
                        memcpy(tempd, A_data, (nb2)*sizeof(REAL));
                        memcpy(A_data, A_data+j*nb2, (nb2)*sizeof(REAL));
                        memcpy(A_data+j*nb2, tempd, (nb2)*sizeof(REAL));
                    }
                    break;
                }
                /* diagonal element is missing */
                if (j == row_size-1) return -2;
            }
            A_j    += row_size;
            A_data += row_size*nb2;
        }
        fasp_mem_free(tempd); tempd = NULL;
#ifdef _OPENMP
    }
#endif

    if (status < 0) return status;
    else            return FASP_SUCCESS;
}

00486 dvector fasp_dbsr_getdiaginv (const dBSRmat *A)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    size = ROW*nb2;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    dvector diaginv;

    INT i,k;

    // Variables for OpenMP
    SHORT nthreads = 1, use_openmp = FALSE;
    INT   myid, mybegin, myend;

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // allocate memory
    diaginv.row = size;
    diaginv.val = (REAL *)fasp_mem_calloc(size, sizeof(REAL));

    // get all the diagonal sub-blocks
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend, k)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    if (JA[k] == i)
                        memcpy(diaginv.val+i*nb2, val+k*nb2, nb2*sizeof(REAL));
                }
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv.val+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }
    // compute the inverses of all the diagonal sub-blocks
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            if (nb > 1) {
                for (i = mybegin; i < myend; ++i) {
                    fasp_smat_inv(diaginv.val+i*nb2, nb);
                }
            }
            else {
                for (i = mybegin; i < myend; ++i) {
                    // zero-diagonal should be tested previously
                    diaginv.val[i] = 1.0 / diaginv.val[i];
                }
            }
        }
    }
    else {
        if (nb > 1) {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv.val+i*nb2, nb);
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv.val[i] = 1.0 / diaginv.val[i];
            }
        }
    }

    return (diaginv);
}

dBSRmat fasp_dbsr_diaginv (const dBSRmat *A)
{
    // members of A
    const INT     ROW  = A->ROW;
    const INT     COL  = A->COL;
    const INT     NNZ  = A->NNZ;
    const INT     nb   = A->nb;
    const INT     nb2  = nb*nb;
    const INT     size = ROW*nb2;
    const INT    *IA   = A->IA;
    const INT    *JA   = A->JA;
    REAL         *val  = A->val;

    // create a dBSRmat B
    dBSRmat B    = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
    INT    *IAb  = B.IA;
    INT    *JAb  = B.JA;
    REAL   *valb = B.val;

    INT     i, j, k, m, l;

    // variables for OpenMP
    SHORT   nthreads = 1, use_openmp = FALSE;
    INT     myid, mybegin, myend;

    // allocate memory
    REAL *diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));

    if ( IAb ) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
    else goto FINISHED;

    if ( JAb ) memcpy(JAb, JA, NNZ*sizeof(INT));
    else goto FINISHED;

#ifdef _OPENMP
```

```
    if (ROW > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // get all the diagonal sub-blocks
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend, k)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    if (JA[k] == i)
                        memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
                }
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }

    // compute the inverses of all the diagonal sub-blocks
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            if (nb > 1) {
                for (i = mybegin; i < myend; ++i) {
                    fasp_smat_inv(diaginv+i*nb2, nb);
                }
            }
            else {
                for (i = mybegin; i < myend; ++i) {
                    // zero-diagonal should be tested previously
                    diaginv[i] = 1.0 / diaginv[i];
                }
            }
        }
    }
    else {
        if (nb > 1) {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv+i*nb2, nb);
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv[i] = 1.0 / diaginv[i];
            }
        }
    }

    // compute D^{-1}*A
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, k, m, j, l)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    m = k*nb2;
                    j = JA[k];
                    if (j == i) {
                        // Identity sub-block
                        memset(valb+m, 0X0, nb2*sizeof(REAL));
                        for (l = 0; l < nb; l ++) valb[m+l*nb+l] = 1.0;
                    }
                    else {
                        fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                    }
                }
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                m = k*nb2;
                j = JA[k];
                if (j == i) {
                    // Identity sub-block
                    memset(valb+m, 0X0, nb2*sizeof(REAL));
                    for (l = 0; l < nb; l ++) valb[m+l*nb+l] = 1.0;
                }
                else {
                    fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                }
            }
        }
    }

FINISHED:
    fasp_mem_free(diaginv); diaginv = NULL;

    return (B);
}

dBSRmat fasp_dbsr_diaginv2 (const dBSRmat *A,
                            REAL          *diaginv)
{
    // members of A
    const INT ROW = A->ROW;
    const INT COL = A->COL;
    const INT NNZ = A->NNZ;
    const INT nb  = A->nb, nbp1 = nb+1;
    const INT nb2 = nb*nb;

    INT    *IA  = A->IA;
    INT    *JA  = A->JA;
    REAL   *val = A->val;

    dBSRmat B;
    INT    *IAb  = NULL;
    INT    *JAb  = NULL;
    REAL   *valb = NULL;

    INT i,k,m,l,ibegin,iend;

    // Variables for OpenMP
    SHORT nthreads = 1, use_openmp = FALSE;
    INT myid, mybegin, myend;

#ifdef _OPENMP
    if (ROW > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // Create a dBSRmat 'B'
    B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);
    IAb  = B.IA;
    JAb  = B.JA;
    valb = B.val;

    if (IAb) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
    else goto FINISHED;

    if (JAb) memcpy(JAb, JA, NNZ*sizeof(INT));
    else goto FINISHED;

    // compute D^{-1}*A
    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private (myid, i, mybegin, myend, ibegin, iend, k, m, l)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                ibegin = IA[i]; iend = IA[i+1];
                for (k = ibegin; k < iend; ++k) {
                    m = k*nb2;
                    if (JA[k] != i) {
                        fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                    }
                    else {
                        // Identity sub-block
                        memset(valb+m, 0X0, nb2*sizeof(REAL));
                        for (l = 0; l < nb; l ++) valb[m+l*nbp1] = 1.0;
                    }
                }
            }
        }
    }
    else {
        // compute D^{-1}*A
        for (i = 0; i < ROW; ++i) {
            ibegin = IA[i]; iend = IA[i+1];
            for (k = ibegin; k < iend; ++k) {
                m = k*nb2;
                if (JA[k] != i) {
                    fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                }
                else {
                    // Identity sub-block
                    memset(valb+m, 0X0, nb2*sizeof(REAL));
                    for (l = 0; l < nb; l ++) valb[m+l*nbp1] = 1.0;
                }
            }
        }
    }

FINISHED:
    return (B);
}

dBSRmat fasp_dbsr_diaginv3 (const dBSRmat *A,
                            REAL          *diaginv)
{
    dBSRmat B;
    // members of A
    INT     ROW = A->ROW;
    INT     ROW_plus_one = ROW+1;
    INT     COL = A->COL;
    INT     NNZ = A->NNZ;
    INT     nb  = A->nb;
    INT    *IA  = A->IA;
    INT    *JA  = A->JA;
    REAL   *val = A->val;

    INT    *IAb  = NULL;
    INT    *JAb  = NULL;
    REAL   *valb = NULL;

    INT     nb2  = nb*nb;
    INT     i,j,k,m;

    SHORT   use_openmp = FALSE;

#ifdef _OPENMP
    INT myid, mybegin, myend, stride_i, nthreads = 1;
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // Create a dBSRmat 'B'
    B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);

    IAb  = B.IA;
    JAb  = B.JA;
    valb = B.val;

    fasp_iarray_cp(ROW_plus_one, IA, IAb);
    fasp_iarray_cp(NNZ, JA, JAb);

    switch (nb) {

        case 2:
            // main loop
            if (use_openmp) {
#ifdef _OPENMP
                stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if (myid < nthreads-1)  myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks

                        k = IA[i];
                        m = k*4;
                        memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
                        fasp_smat_identity_nc2(valb+m);

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc2(diaginv+i*4);
                        // compute D^{-1}*A
                        for (k = IA[i]+1; k < IA[i+1]; ++k)
                        {
                            m = k*4;
                            j = JA[k];
                            fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
                        }
                    }// end of main loop
                }
#endif
            }
            else {
                // main loop
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    k = IA[i];
                    m = k*4;
                    memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
                    fasp_smat_identity_nc2(valb+m);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc2(diaginv+i*4);
                    // compute D^{-1}*A
                    for (k = IA[i]+1; k < IA[i+1]; ++k) {
                        m = k*4;
                        fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        case 3:
            // main loop
            if (use_openmp) {
#ifdef _OPENMP
                stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if (myid < nthreads-1)  myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            if (JA[k] == i) {
                                m = k*9;
                                memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
                                fasp_smat_identity_nc3(valb+m);
                            }
                        }
                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc3(diaginv+i*9);
                        // compute D^{-1}*A
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            m = k*9;
                            j = JA[k];
                            if (j != i) fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
                        }
                    }// end of main loop
                }
#endif
            }

            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        if (JA[k] == i) {
                            m = k*9;
                            memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
                            fasp_smat_identity_nc3(valb+m);
                        }
                    }
#if DEBUG_MODE > 0
                    printf("### DEBUG: row, col = %d\n", i);
#endif
                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc3(diaginv+i*9);

                    // compute D^{-1}*A
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        m = k*9;
                        j = JA[k];
                        if (j != i) fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        case -5:
            // main loop
            if (use_openmp) {
#ifdef _OPENMP
                stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if (myid < nthreads-1)  myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            if (JA[k] == i) {
                                m = k*25;
                                memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
                                fasp_smat_identity_nc5(valb+m);
                            }
                        }

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc5(diaginv+i*25);

                        // compute D^{-1}*A
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            m = k*25;
                            j = JA[k];
                            if (j != i) fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
                        }
                    }// end of main loop
                }
#endif
            }

            else {

                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        if (JA[k] == i)
                        {
                            m = k*25;
                            memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
                            fasp_smat_identity_nc5(valb+m);
                        }
                    }

                    // compute the inverses of the diagonal sub-blocks
                    // fasp_smat_inv_nc5(diaginv+i*25); // Not numerically stable!!!  --zcs 04/26/2021
                    fasp_smat_invp_nc(diaginv + i * 25, 5);

#if 0
                    REAL aa[25], bb[25]; // for debug inverse of diag
                    for (k = 0; k < 25; k++) bb[k] = diaginv[i * 25 + k]; // before inversion
                    for (k = 0; k < 25; k++) aa[k] = diaginv[i * 25 + k]; // aftger inversion

                    printf("### DEBUG: Check inverse matrix...\n");
                    printf("##----------------------------------------------\n");
                    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
                        bb[0]* aa[0] + bb[1] * aa[5] + bb[2] * aa[10] + bb[3] * aa[15] + bb[4] * aa[20],
                        bb[0]* aa[1] + bb[1] * aa[6] + bb[2] * aa[11] + bb[3] * aa[16] + bb[4] * aa[21],
                        bb[0]* aa[2] + bb[1] * aa[7] + bb[2] * aa[12] + bb[3] * aa[17] + bb[4] * aa[22],
                        bb[0]* aa[3] + bb[1] * aa[8] + bb[2] * aa[13] + bb[3] * aa[18] + bb[4] * aa[23],
                        bb[0]* aa[4] + bb[1] * aa[9] + bb[3] * aa[14] + bb[3] * aa[19] + bb[4] * aa[24]);
                    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
                        bb[5]* aa[0] + bb[6] * aa[5] + bb[7] * aa[10] + bb[8] * aa[15] + bb[9] * aa[20],
                        bb[5]* aa[1] + bb[6] * aa[6] + bb[7] * aa[11] + bb[8] * aa[16] + bb[9] * aa[21],
                        bb[5]* aa[2] + bb[6] * aa[7] + bb[7] * aa[12] + bb[8] * aa[17] + bb[9] * aa[22],
                        bb[5]* aa[3] + bb[6] * aa[8] + bb[7] * aa[13] + bb[8] * aa[18] + bb[9] * aa[23],
                        bb[5]* aa[4] + bb[6] * aa[9] + bb[7] * aa[14] + bb[8] * aa[19] + bb[9] * aa[24]);
                    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
                        bb[10]* aa[0] + bb[11] * aa[5] + bb[12] * aa[10] + bb[13] * aa[15] + bb[14] * aa[20],
                        bb[10]* aa[1] + bb[11] * aa[6] + bb[12] * aa[11] + bb[13] * aa[16] + bb[14] * aa[21],
                        bb[10]* aa[2] + bb[11] * aa[7] + bb[12] * aa[12] + bb[13] * aa[17] + bb[14] * aa[22],
                        bb[10]* aa[3] + bb[11] * aa[8] + bb[12] * aa[13] + bb[13] * aa[18] + bb[14] * aa[23],
                        bb[10]* aa[4] + bb[11] * aa[9] + bb[12] * aa[14] + bb[13] * aa[19] + bb[14] * aa[24]);
                    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
                        bb[15]* aa[0] + bb[16] * aa[5] + bb[17] * aa[10] + bb[18] * aa[15] + bb[19] * aa[20],
                        bb[15]* aa[1] + bb[16] * aa[6] + bb[17] * aa[11] + bb[18] * aa[16] + bb[19] * aa[21],
                        bb[15]* aa[2] + bb[16] * aa[7] + bb[17] * aa[12] + bb[18] * aa[17] + bb[19] * aa[22],
                        bb[15]* aa[3] + bb[16] * aa[8] + bb[17] * aa[13] + bb[18] * aa[18] + bb[19] * aa[23],
                        bb[15]* aa[4] + bb[16] * aa[9] + bb[17] * aa[14] + bb[18] * aa[19] + bb[19] * aa[24]);
                    printf("## %12.5e %12.5e %12.5e %12.5e %12.5e\n",
                        bb[20]* aa[0] + bb[21] * aa[5] + bb[22] * aa[10] + bb[23] * aa[15] + bb[24] * aa[20],
                        bb[20]* aa[1] + bb[21] * aa[6] + bb[22] * aa[11] + bb[23] * aa[16] + bb[24] * aa[21],
                        bb[20]* aa[2] + bb[21] * aa[7] + bb[22] * aa[12] + bb[23] * aa[17] + bb[24] * aa[22],
                        bb[20]* aa[3] + bb[21] * aa[8] + bb[22] * aa[13] + bb[23] * aa[18] + bb[24] * aa[23],
                        bb[20]* aa[4] + bb[21] * aa[9] + bb[22] * aa[14] + bb[23] * aa[19] + bb[24] * aa[24]);
                    printf("##----------------------------------------------\n");
#endif

                    // compute D^{-1}*A
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        m = k*25;
                        j = JA[k];
                        if (j != i) fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        case -7:
            // main loop
            if (use_openmp) {
#ifdef _OPENMP
                stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if (myid < nthreads-1)  myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            if (JA[k] == i) {
                                m = k*49;
                                memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
                                fasp_smat_identity_nc7(valb+m);
                            }
                        }

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc7(diaginv+i*49);

                        // compute D^{-1}*A
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            m = k*49;
                            j = JA[k];
                            if (j != i) fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
                        }
                    }// end of main loop
                }
#endif
            }

            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        if (JA[k] == i) {
                            m = k*49;
                            memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
                            fasp_smat_identity_nc7(valb+m);
                        }
                    }

                    // compute the inverses of the diagonal sub-blocks
                    // fasp_smat_inv_nc7(diaginv+i*49); // Not numerically stable!!!  --zcs 04/26/2021
                    fasp_smat_invp_nc(diaginv + i * 49, 7);

                    // compute D^{-1}*A
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        m = k*49;
                        j = JA[k];
                        if (j != i) fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        default:
            // main loop
            if (use_openmp) {
#ifdef _OPENMP
                stride_i = ROW/nthreads;
#pragma omp parallel private(myid, mybegin, myend,i,k,m,j) num_threads(nthreads)
                {
                    myid = omp_get_thread_num();
                    mybegin = myid*stride_i;
                    if (myid < nthreads-1)  myend = mybegin+stride_i;
                    else myend = ROW;
                    for (i=mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            if (JA[k] == i) {
                                m = k*nb2;
                                memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
                                fasp_smat_identity(valb+m, nb, nb2);
                            }
                        }

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv(diaginv+i*nb2, nb);

                        // compute D^{-1}*A
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            m = k*nb2;
                            j = JA[k];
                            if (j != i) fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                        }
                    }// end of main loop
                }
#endif
```

```
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        if (JA[k] == i) {
                            m = k*nb2;
                            memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
                            fasp_smat_identity(valb+m, nb, nb2);
                        }
                    }

                    // compute the inverses of the diagonal sub-blocks
                    // fasp_smat_inv(diaginv+i*nb2, nb); // Not numerically stable!!!  --zcs 04/26/2021
                    fasp_smat_invp_nc(diaginv + i * nb2, nb);

                    // compute D^{-1}*A
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        m = k*nb2;
                        j = JA[k];
                        if (j != i) fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                    }
                } // end of main loop
            }

            break;
    }

    return (B);
}

01260 dBSRmat fasp_dbsr_diaginv4 (const dBSRmat *A,
                               REAL          *diaginv)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     COL = A->COL;
    const INT     NNZ = A->NNZ;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    dBSRmat B;
    INT    *IAb  = NULL;
    INT    *JAb  = NULL;
    REAL   *valb = NULL;

    INT i,k,m;
    INT ibegin, iend;

    // Variables for OpenMP
    SHORT nthreads = 1, use_openmp = FALSE;
    INT myid, mybegin, myend;

#ifdef _OPENMP
    if (ROW > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // Create a dBSRmat 'B'
    B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);

    IAb  = B.IA;
    JAb  = B.JA;
    valb = B.val;

    if (IAb) memcpy(IAb, IA, (ROW+1)*sizeof(INT));
    else goto FINISHED;

    if (JAb) memcpy(JAb, JA, NNZ*sizeof(INT));
    else goto FINISHED;

    switch (nb) {

        case 2:
            if (use_openmp) {
#ifdef _openmp
```

```
#pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
#endif
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i = mybegin; i < myend; ++i) {
                        ibegin = IA[i]; iend = IA[i+1];
                        // get the diagonal sub-blocks (It is the first block of each row)
                        m = ibegin*4;
                        memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
                        fasp_smat_identity_nc2(valb+m);

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc2(diaginv+i*4); // fixed by Zheng Li

                        // compute D^{-1}*A
                        for (k = ibegin+1; k < iend; ++k) {
                            m = k*4;
                            fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
                        }
                    }
                }// end of main loop
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    ibegin = IA[i]; iend = IA[i+1];
                    // get the diagonal sub-blocks (It is the first block of each row)
                    m = ibegin*4;
                    memcpy(diaginv+i*4, val+m, 4*sizeof(REAL));
                    fasp_smat_identity_nc2(valb+m);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc2(diaginv+i*4); // fixed by Zheng Li

                    // compute D^{-1}*A
                    for (k = ibegin+1; k < iend; ++k) {
                        m = k*4;
                        fasp_blas_smat_mul_nc2(diaginv+i*4, val+m, valb+m);
                    }
                } // end of main loop
            }

            break;

        case 3:
            if (use_openmp) {
#ifdef _openmp
#pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
#endif
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i = mybegin; i < myend; ++i) {
                        ibegin = IA[i]; iend = IA[i+1];
                        // get the diagonal sub-blocks (It is the first block of each row)
                        m = ibegin*9;
                        memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
                        fasp_smat_identity_nc3(valb+m);
                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc3(diaginv+i*9);
                        // compute D^{-1}*A
                        for (k = ibegin+1; k < iend; ++k) {
                            m = k*9;
                            fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
                        }
                    }
                }// end of main loop
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    ibegin = IA[i]; iend = IA[i+1];
                    // get the diagonal sub-blocks (It is the first block of each row)
                    m = ibegin*9;
                    memcpy(diaginv+i*9, val+m, 9*sizeof(REAL));
                    fasp_smat_identity_nc3(valb+m);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc3(diaginv+i*9);

                    // compute D^{-1}*A
                    for (k = ibegin+1; k < iend; ++k) {
                        m = k*9;
                        fasp_blas_smat_mul_nc3(diaginv+i*9, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        case 5:
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
#endif
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i = mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        ibegin = IA[i]; iend = IA[i+1];
                        m = ibegin*25;
                        memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
                        fasp_smat_identity_nc5(valb+m);

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc5(diaginv+i*25);

                        // compute D^{-1}*A
                        for (k = ibegin+1; k < iend; ++k) {
                            m = k*25;
                            fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    ibegin = IA[i]; iend = IA[i+1];
                    m = ibegin*25;
                    memcpy(diaginv+i*25, val+m, 25*sizeof(REAL));
                    fasp_smat_identity_nc5(valb+m);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc5(diaginv+i*25);

                    // compute D^{-1}*A
                    for (k = ibegin+1; k < iend; ++k) {
                        m = k*25;
                        fasp_blas_smat_mul_nc5(diaginv+i*25, val+m, valb+m);
                    }
                }// end of main loop
            }
            break;

        case 7:
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend, ibegin, iend, m, k)
#endif
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i = mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        ibegin = IA[i]; iend = IA[i+1];
                        m = ibegin*49;
                        memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
                        fasp_smat_identity_nc7(valb+m);

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv_nc7(diaginv+i*49);

                        // compute D^{-1}*A
                        for (k = ibegin+1; k < iend; ++k) {
                            m = k*49;
                            fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
                        }
                    }
                }// end of main loop
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    ibegin = IA[i]; iend = IA[i+1];
                    m = ibegin*49;
                    memcpy(diaginv+i*49, val+m, 49*sizeof(REAL));
                    fasp_smat_identity_nc7(valb+m);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv_nc7(diaginv+i*49);

                    // compute D^{-1}*A
                    for (k = ibegin+1; k < iend; ++k) {
                        m = k*49;
                        fasp_blas_smat_mul_nc7(diaginv+i*49, val+m, valb+m);
                    }
                }// end of main loop
            }

            break;

        default:
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, m, k)
#endif
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i = mybegin; i < myend; ++i) {
                        // get the diagonal sub-blocks
                        ibegin = IA[i]; iend = IA[i+1];
                        m = ibegin*nb2;
                        memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
                        fasp_smat_identity(valb+m, nb, nb2);

                        // compute the inverses of the diagonal sub-blocks
                        fasp_smat_inv(diaginv+i*nb2, nb);

                        // compute D^{-1}*A
                        for (k = ibegin+1; k < iend; ++k) {
                            m = k*nb2;
                            fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                        }
                    }
                }// end of main loop
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    // get the diagonal sub-blocks
                    ibegin = IA[i]; iend = IA[i+1];
                    m = ibegin*nb2;
                    memcpy(diaginv+i*nb2, val+m, nb2*sizeof(REAL));
                    fasp_smat_identity(valb+m, nb, nb2);

                    // compute the inverses of the diagonal sub-blocks
                    fasp_smat_inv(diaginv+i*nb2, nb);

                    // compute D^{-1}*A
                    for (k = ibegin+1; k < iend; ++k) {
                        m = k*nb2;
                        fasp_blas_smat_mul(diaginv+i*nb2, val+m, valb+m, nb);
                    }
                } // end of main loop
            }

            break;
    }

FINISHED:
    return (B);
}

void fasp_dbsr_getdiag (INT             n,
                        const dBSRmat *A,
                        REAL           *diag )
{
    const INT nb2 = A->nb*A->nb;

    INT i,k;

    if ( n==0 || n>A->ROW || n>A->COL ) n = MIN(A->ROW,A->COL);

#ifdef _OPENMP
#pragma omp parallel for private(i,k) if(n>OPENMP_HOLDS)
#endif
    for (i = 0; i < n; ++i) {
        for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
            if (A->JA[k] == i) {
                memcpy(diag+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
                break;
            }
        }
    }
}

dBSRmat fasp_dbsr_diagLU (const dBSRmat *A,
                          REAL          *DL,
                          REAL          *DU)
{

    // members of A
    const INT  ROW = A->ROW;
    const INT  ROW_plus_one = ROW+1;
    const INT  COL = A->COL;
    const INT  NNZ = A->NNZ;
    const INT  nb  = A->nb;

    const INT  *IA  = A->IA;
    const INT  *JA  = A->JA;
    const REAL *val = A->val;

    INT  *IAb  = NULL;
    INT  *JAb  = NULL;
    REAL *valb = NULL;

    INT nb2  = nb*nb;
    INT i, j, k;

    // Create a dBSRmat 'B'
    dBSRmat B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);

    IAb  = B.IA;
    JAb  = B.JA;
    valb = B.val;

    fasp_iarray_cp(ROW_plus_one, IA, IAb);
    fasp_iarray_cp(NNZ, JA, JAb);

    // work array
    REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));

    // get DL and DU
    switch (nb) {

        case 2:

            for (i=0; i<ROW; i++){

                for (j=IA[i]; j<IA[i+1]; j++){

                    if (JA[j] == i){

                        temp[0] = val[j*nb2];
                        temp[1] = val[j*nb2+1];
                        temp[2] = val[j*nb2+2];
                        temp[3] = val[j*nb2+3];

                        // form DL
                        DL[i*nb2]   = 1.0;
                        DL[i*nb2+1] = 0.0;
                        DL[i*nb2+2] = -temp[2]/temp[0];
                        DL[i*nb2+3] = 1.0;
                        //DL[i*nb2+2] = -temp[2]/(temp[0]*s);
                        //DL[i*nb2+3] = 1.0/s;

                        // form DU
                        DU[i*nb2]   = 1.0;
                        DU[i*nb2+1] = -temp[1]/temp[0];
                        DU[i*nb2+2] = 0.0;
                        DU[i*nb2+3] = 1.0;

                        break;

                    } // end of if (JA[j] == i)

                } // end of for (j=IA[i]; j<IA[i+1]; j++)

            } // end of for (i=0; i<ROW; i++)

            break;

        case 3:

            for (i=0; i<ROW; i++){

                for (j=IA[i]; j<IA[i+1]; j++){

                    if (JA[j] == i){

                        temp[0] = val[j*nb2];
                        temp[1] = val[j*nb2+1];
                        temp[2] = val[j*nb2+2];
                        temp[3] = val[j*nb2+3];
                        temp[4] = val[j*nb2+4];
                        temp[5] = val[j*nb2+5];
                        temp[6] = val[j*nb2+6];
                        temp[7] = val[j*nb2+7];
                        temp[8] = val[j*nb2+8];

                        // some auxiliry variables
                        REAL s22 = temp[4] - ((temp[1]*temp[3])/temp[0]);
                        REAL s23 = temp[5] - ((temp[2]*temp[3])/temp[0]);
                        REAL s32 = temp[7] - ((temp[1]*temp[6])/temp[0]);

                        // form DL
                        DL[i*nb2]   = 1.0;
                        DL[i*nb2+1] = 0.0;
                        DL[i*nb2+2] = 0.0;
                        DL[i*nb2+3] = -temp[3]/temp[0];
                        DL[i*nb2+4] = 1.0;
                        DL[i*nb2+5] = 0.0;
                        DL[i*nb2+6] = -temp[6]/temp[0] + (temp[3]/temp[0])*(s32/s22);
                        DL[i*nb2+7] = -s32/s22;
                        DL[i*nb2+8] = 1.0;

                        // form DU
                        DU[i*nb2]   = 1.0;
                        DU[i*nb2+1] = -temp[1]/temp[0];
                        DU[i*nb2+2] = -temp[2]/temp[0] + (temp[1]/temp[0])*(s23/s22);
                        DU[i*nb2+3] = 0.0;
                        DU[i*nb2+4] = 1.0;
                        DU[i*nb2+5] = -s23/s22;
                        DU[i*nb2+6] = 0.0;
                        DU[i*nb2+7] = 0.0;
                        DU[i*nb2+8] = 1.0;

                        break;

                    } // end of if (JA[j] == i)

                } // end of for (j=IA[i]; j<IA[i+1]; j++)

            } // end of for (i=0; i<ROW; i++)

            break;

        default:
            printf("### ERROR: Only works for nb = 2, 3!  [%s]\n", __FUNCTION__);
            break;

    } // end of switch

    // compute B = DL*A*DU
    switch (nb) {

        case 2:

            for (i=0; i<ROW; i++){

                for (j=IA[i]; j<IA[i+1]; j++){

                    k = JA[j];

                    // left multiply DL
                    fasp_blas_smat_mul_nc2(DL+i*nb2, val+j*nb2, temp);

                    // right multiply DU
                    fasp_blas_smat_mul_nc2(temp, DU+k*nb2, valb+j*nb2);

                    // for diagonal block, set it to be diagonal matrix
                    if (JA[j] == i){

                        valb[j*nb2+1] = 0.0;
                        valb[j*nb2+2] = 0.0;

                    } // end if (JA[j] == i)


                } // end for (j=IA[i]; j<IA[i+1]; j++)

            } // end of for (i=0; i<ROW; i++)

            break;

        case 3:

            for (i=0; i<ROW; i++){

                for (j=IA[i]; j<IA[i+1]; j++){

                    k = JA[j];

                    // left multiply DL
                    fasp_blas_smat_mul_nc3(DL+i*nb2, val+j*nb2, temp);

                    // right multiply DU
                    fasp_blas_smat_mul_nc3(temp, DU+k*nb2, valb+j*nb2);

                    // for diagonal block, set it to be diagonal matrix
                    if (JA[j] == i){

                        valb[j*nb2+1] = 0.0;
                        valb[j*nb2+2] = 0.0;
                        valb[j*nb2+3] = 0.0;
                        valb[j*nb2+5] = 0.0;
                        valb[j*nb2+6] = 0.0;
                        valb[j*nb2+7] = 0.0;
                        if (ABS(valb[j*nb2+4]) < SMALLREAL) valb[j*nb2+4] = SMALLREAL;
                        if (ABS(valb[j*nb2+8]) < SMALLREAL) valb[j*nb2+8] = SMALLREAL;

                    } // end if (JA[j] == i)

                } // end for (j=IA[i]; j<IA[i+1]; j++)

            } // end of for (i=0; i<ROW; i++)

            break;

        default:
            printf("### ERROR: Only works for nb = 2, 3!  [%s]\n", __FUNCTION__);
            break;

    }

    // return
    return B;

}

dBSRmat fasp_dbsr_diagLU2 (dBSRmat *A,
                           REAL    *DL,
                           REAL    *DU)
{

    // members of A
    INT  ROW = A->ROW;
    INT  ROW_plus_one = ROW+1;
    INT  COL = A->COL;
    INT  NNZ = A->NNZ;
    INT  nb  = A->nb;
    INT  *IA  = A->IA;
    INT  *JA  = A->JA;
    REAL *val = A->val;

    INT  *IAb  = NULL;
    INT  *JAb  = NULL;
    REAL *valb = NULL;

    INT nb2  = nb*nb;
    INT i,j,k;

    REAL sqt3, sqt4, sqt8;

    // Create a dBSRmat 'B'
    dBSRmat B = fasp_dbsr_create(ROW, COL, NNZ, nb, 0);

    REAL *temp = (REAL *)fasp_mem_calloc(nb*nb, sizeof(REAL));

    IAb  = B.IA;
    JAb  = B.JA;
    valb = B.val;

    fasp_iarray_cp(ROW_plus_one, IA, IAb);
    fasp_iarray_cp(NNZ, JA, JAb);

    // get DL and DU
    switch (nb) {
        case 2:
            for (i=0; i<ROW; i++){
                for (j=IA[i]; j<IA[i+1]; j++){
                    if (JA[j] == i){
                        REAL temp0 = val[j*nb2];
                        REAL temp1 = val[j*nb2+1];
                        REAL temp2 = val[j*nb2+2];
                        REAL temp3 = val[j*nb2+3];

                        if (ABS(temp3) < SMALLREAL) temp3 = SMALLREAL;

                        sqt3 = sqrt(ABS(temp3));

                        // form DL
                        DL[i*nb2]   = 1.0;
                        DL[i*nb2+1] = 0.0;
                        DL[i*nb2+2] = -temp2/temp0/sqt3;
                        DL[i*nb2+3] = 1.0/sqt3;

                        // form DU
                        DU[i*nb2]   = 1.0;
                        DU[i*nb2+1] = -temp1/temp0/sqt3;
                        DU[i*nb2+2] = 0.0;
                        DU[i*nb2+3] = 1.0/sqt3;
                        break;

                    }
                }
            }

            break;

        case 3:
            for (i=0; i<ROW; i++){
                for (j=IA[i]; j<IA[i+1]; j++){
                    if (JA[j] == i){
                        REAL temp0 = val[j*nb2];
                        REAL temp1 = val[j*nb2+1];
                        REAL temp2 = val[j*nb2+2];
                        REAL temp3 = val[j*nb2+3];
                        REAL temp4 = val[j*nb2+4];
                        REAL temp5 = val[j*nb2+5];
                        REAL temp6 = val[j*nb2+6];
                        REAL temp7 = val[j*nb2+7];
                        REAL temp8 = val[j*nb2+8];

                        if (ABS(temp4) < SMALLREAL)  temp4 = SMALLREAL;
                        if (ABS(temp8) < SMALLREAL)  temp8 = SMALLREAL;

                        sqt4 = sqrt(ABS(temp4));
                        sqt8 = sqrt(ABS(temp8));

                        // some auxiliary variables
                        REAL s22 = temp4 - ((temp1*temp3)/temp0);
                        REAL s23 = temp5 - ((temp2*temp3)/temp0);
                        REAL s32 = temp7 - ((temp1*temp6)/temp0);

                        // form DL
                        DL[i*nb2]   = 1.0;
                        DL[i*nb2+1] = 0.0;
                        DL[i*nb2+2] = 0.0;
                        DL[i*nb2+3] = -temp3/temp0/sqt4;
                        DL[i*nb2+4] = 1.0/sqt4;
                        DL[i*nb2+5] = 0.0;
                        DL[i*nb2+6] = (-temp6/temp0 + (temp3/temp0)*(s32/s22))/sqt8;
                        DL[i*nb2+7] = -s32/s22/sqt8;
                        DL[i*nb2+8] = 1.0/sqt8;

                        // form DU
                        DU[i*nb2]   = 1.0;
                        DU[i*nb2+1] = -temp1/temp0/sqt4;
                        DU[i*nb2+2] = (-temp2/temp0 + (temp1/temp0)*(s23/s22))/sqt8;
                        DU[i*nb2+3] = 0.0;
                        DU[i*nb2+4] = 1.0/sqt4;
                        DU[i*nb2+5] = -s23/s22/sqt8;
                        DU[i*nb2+6] = 0.0;
                        DU[i*nb2+7] = 0.0;
                        DU[i*nb2+8] = 1.0/sqt8;

                        break;

                    }
                }
            }

            break;

        default:
            printf("### ERROR: Only works for nb = 2, 3!  [%s]\n", __FUNCTION__);
            break;

    } // end of switch

    // compute B = DL*A*DU
    switch (nb) {

        case 2:
            for (i=0; i<ROW; i++){
                for (j=IA[i]; j<IA[i+1]; j++){
                    k = JA[j];
                    // left multiply DL
                    fasp_blas_smat_mul_nc2(DL+i*nb2, val+j*nb2, temp);
                    // right multiply DU
                    fasp_blas_smat_mul_nc2(temp, DU+k*nb2, valb+j*nb2);
                    // for diagonal block, set it to be diagonal matrix
                    if (JA[j] == i){
                        valb[j*nb2+1] = 0.0;
                        valb[j*nb2+2] = 0.0;
                        if (ABS(valb[j*nb2+3]) < SMALLREAL) valb[j*nb2+3] = SMALLREAL;
                    }
                }
            }

            break;

        case 3:
            for (i=0; i<ROW; i++){
                for (j=IA[i]; j<IA[i+1]; j++){
                    k = JA[j];
                    // left multiply DL
                    fasp_blas_smat_mul_nc3(DL+i*nb2, val+j*nb2, temp);
                    // right multiply DU
                    fasp_blas_smat_mul_nc3(temp, DU+k*nb2, valb+j*nb2);
                    // for diagonal block, set it to be diagonal matrix
                    if (JA[j] == i){
                        valb[j*nb2+1] = 0.0;
                        valb[j*nb2+2] = 0.0;
                        valb[j*nb2+3] = 0.0;
                        valb[j*nb2+5] = 0.0;
                        valb[j*nb2+6] = 0.0;
                        valb[j*nb2+7] = 0.0;
                        if (ABS(valb[j*nb2+4]) < SMALLREAL) valb[j*nb2+4] = SMALLREAL;
                        if (ABS(valb[j*nb2+8]) < SMALLREAL) valb[j*nb2+8] = SMALLREAL;
                    }
                }
            }
            break;

        default:
            printf("### ERROR: Only works for nb = 2, 3!  [%s]\n", __FUNCTION__);
            break;
    }

    // return
    return B;

}

dBSRmat fasp_dbsr_perm (const dBSRmat *A,
                        const INT     *P)
{
    const INT   n = A->ROW, nnz = A->NNZ;
    const INT  *ia= A->IA, *ja = A->JA;
    const REAL *Aval = A->val;
    const INT   nb = A->nb, nb2 = nb*nb;
    const INT   manner = A->storage_manner;
    SHORT       nthreads = 1, use_openmp = FALSE;

    INT i,j,k,jaj,i1,i2,start,jj;

#ifdef _OPENMP
    if ( MIN(n, nnz) > OPENMP_HOLDS ) {
        use_openmp = 0;//TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    dBSRmat Aperm = fasp_dbsr_create(n,n,nnz,nb,manner);

    // form the transpose of P
    INT *Pt = (INT*)fasp_mem_calloc(n,sizeof(INT));

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) Pt[P[i]] = i;
        }
    }
    else {
        for (i=0; i<n; ++i) Pt[P[i]] = i;
    }

    // compute IA of P*A (row permutation)
    Aperm.IA[0] = 0;
    for (i=0; i<n; ++i) {
        k = P[i];
        Aperm.IA[i+1] = Aperm.IA[i]+(ia[k+1]-ia[k]);
    }

    // perform actual P*A
    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, i1, i2, k, start, j, jaj, jj)
#endif
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
                i1 = Aperm.IA[i]; i2 = Aperm.IA[i+1]-1;
                k = P[i];
                start = ia[k];
                for (j=i1; j<=i2; ++j) {
                    jaj = start+j-i1;
                    Aperm.JA[j] = ja[jaj];
                    for (jj=0; jj<nb2;++jj)
                        Aperm.val[j*nb2+jj] = Aval[jaj*nb2+jj];
                }
            }
        }
    }
    else {
        for (i=0; i<n; ++i) {
            i1 = Aperm.IA[i]; i2 = Aperm.IA[i+1]-1;
            k = P[i];
            start = ia[k];
            for (j=i1; j<=i2; ++j) {
                jaj = start+j-i1;
                Aperm.JA[j] = ja[jaj];
                for (jj=0; jj<nb2;++jj)
                    Aperm.val[j*nb2+jj] = Aval[jaj*nb2+jj];
            }
        }
    }

    // perform P*A*P' (column permutation)
    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, k, j)
#endif
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
            for (k=mybegin; k<myend; ++k) {
                j = Aperm.JA[k];
                Aperm.JA[k] = Pt[j];
            }
        }
    }
    else {
        for (k=0; k<nnz; ++k) {
            j = Aperm.JA[k];
            Aperm.JA[k] = Pt[j];
        }
    }

    fasp_mem_free(Pt); Pt = NULL;

    return(Aperm);
}

INT fasp_dbsr_merge_col (dBSRmat *A)
{
    INT           count = 0;
    const INT     num_rowsA = A -> ROW;
    const INT     nb = A->nb;
    const INT     nb2 = nb*nb;
    INT          *A_i    = A -> IA;
    INT          *A_j    = A -> JA;
    REAL         *A_data = A -> val;

    INT   i, ii, j, jj, ibegin, iend, iend1;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

#ifdef _OPENMP
    if (num_rowsA > OPENMP_HOLDS) {
#pragma omp parallel for private (myid,mybegin,myend,i,ii,j,jj,ibegin,iend,iend1)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                ibegin = A_i[i]; iend = A_i[i+1]; iend1 =  iend-1;
                for (j = ibegin; j < iend1; j ++) {
                    if (A_j[j] > -1) {
                        for (jj = j+1; jj < iend; jj ++) {
                            if (A_j[j] == A_j[jj]) {
                                // add jj col to j
                                for ( ii=0; ii <nb2; ii++ )
                                    A_data[j*nb2 +ii] += A_data[ jj*nb2+ii];
                                A_j[jj] = -1;
                                count ++;
                            }
                        }
                    }
                }
            }
        }
    }
    else {
#endif
        for (i = 0; i < num_rowsA; i ++) {
            ibegin = A_i[i]; iend = A_i[i+1]; iend1 =  iend-1;
            for (j = ibegin; j < iend1; j ++) {
                if (A_j[j] > -1) {
                    for (jj = j+1; jj < iend; jj ++) {
                        if (A_j[j] == A_j[jj]) {
                            // add jj col to j
                            for ( ii=0; ii <nb2; ii++ )
                                A_data[j*nb2 +ii] += A_data[ jj*nb2+ii];
                            printf("### WARNING: Same col indices at %d, col %d (%d %d)!\n",
                                   i, A_j[j], j, jj );
                            A_j[jj] = -1;
                            count ++;
                        }
                    }
                }
            }
        }
#ifdef _OPENMP
    }
#endif

    if ( count > 0 ) {
        INT *tempA_i = (INT*)fasp_mem_calloc(num_rowsA+1, sizeof(INT));
        memcpy(tempA_i, A_i, (num_rowsA+1 )*sizeof(INT));
        jj = 0;     A_i[0] = jj;
        for (i = 0; i < num_rowsA; i ++) {
            ibegin = tempA_i[i]; iend = tempA_i[i+1];
            for (j = ibegin; j < iend; j ++) {
                if (A_j[j] > -1 ) {
                    memcpy(A_data +jj*nb2, A_data+j*nb2, (nb2)*sizeof(REAL));
                    A_j[jj] = A_j[j];
                    jj++;
                }
            }
            A_i[i+1]    = jj;
        }
        A-> NNZ = jj;
        fasp_mem_free(tempA_i); tempA_i = NULL;

        printf("### WARNING: %d col indices have been merged!\n", count);
    }

    return count;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.75 BlaSparseCheck.c File Reference

Check properties of sparse matrices.

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

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 chechs 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

A	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

A	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

A	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

A	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.

```c

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_check_diagpos (const dCSRmat *A)
{
    const INT m = A->row;
    INT i, num_neg;

#if DEBUG_MODE > 1
    printf("### DEBUG: nr = %d, nc = %d, nnz = %d\n", A->row, A->col, A->nnz);
#endif

    // store diagonal of A
    dvector         diag;
    fasp_dcsr_getdiag(m,A,&diag);

    // check positiveness of entries of diag
    for (num_neg=i=0;i<m;++i) {
        if (diag.val[i]<0) num_neg++;
    }

    printf("Number of negative diagonal entries = %d\n", num_neg);

    fasp_dvec_free(&diag);

    return num_neg;
}

SHORT fasp_check_diagzero (const dCSRmat *A)
{
    const INT     m   = A->row;
    const INT    *ia = A->IA, *ja = A->JA;
    const REAL   *aj = A->val;

    SHORT         status = FASP_SUCCESS;
    INT           i,j,k,begin_row,end_row;

    for ( i = 0; i < m; ++i ) {
        begin_row = ia[i]; end_row = ia[i+1];
        for ( k = begin_row; k < end_row; ++k ) {
            j = ja[k];
            if ( i == j ) {
                if ( ABS(aj[k]) < SMALLREAL ) {
                    printf("### ERROR: diag[%d] = %e, close to zero!\n", i, aj[k]);
                    status = ERROR_DATA_ZERODIAG;
                    goto FINISHED;
                }
            }
        } // end for k
    } // end for i

 FINISHED:
    return status;
}

INT fasp_check_diagdom (const dCSRmat *A)
{
    const INT   nn  = A->row;
    const INT   nnz = A->IA[nn]-A->IA[0];
    INT         i, j, k;
    REAL        sum;

    INT *rowp = (INT *)fasp_mem_calloc(nnz,sizeof(INT));

    for ( i=0; i<nn; ++i ) {
        for ( j=A->IA[i]; j<A->IA[i+1]; ++j ) rowp[j]=i;
    }

    for ( k=0, i=0; i<nn; ++i ) {
        sum = 0.0;
        for ( j=A->IA[i]; j<A->IA[i+1]; ++j ) {
            if ( A->JA[j]==i ) sum += A->val[j];
            if ( A->JA[j]!=i ) sum -= fabs(A->val[j]);
        }
        if ( sum<-SMALLREAL ) ++k;
    }

    printf("Percentage of the diagonal-dominant rows is %3.2lf%s\n",
           100.0*(REAL)(nn-k)/(REAL)nn,"%");

    fasp_mem_free(rowp); rowp = NULL;

    return k;
}

INT fasp_check_symm (const dCSRmat *A)
{
    const REAL symmetry_tol = 1.0e-12;

    INT  *rowp,*rows[2],*cols[2];
    INT   i,j,mdi,mdj;
    INT   nns[2],tnizs[2];
    INT   type=0;

    REAL  maxdif,dif;
    REAL *vals[2];

    const INT nn  = A->row;
    const INT nnz = A->IA[nn]-A->IA[0];

    if (nnz!=A->nnz) {
        printf("### ERROR: nnz=%d, ia[n]-ia[0]=%d, mismatch!\n",A->nnz,nnz);
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    rowp=(INT *)fasp_mem_calloc(nnz,sizeof(INT));

    for (i=0;i<nn;++i) {
        for (j=A->IA[i];j<A->IA[i+1];++j) rowp[j]=i;
    }

    rows[0]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
    cols[0]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
    vals[0]=(REAL *)fasp_mem_calloc(nnz,sizeof(REAL));

    memcpy(rows[0],rowp,nnz*sizeof(INT));
    memcpy(cols[0],A->JA,nnz*sizeof(INT));
    memcpy(vals[0],A->val,nnz*sizeof(REAL));

    nns[0]=nn;
    nns[1]=A->col;
    tnizs[0]=nnz;

    rows[1]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
    cols[1]=(INT *)fasp_mem_calloc(nnz,sizeof(INT));
    vals[1]=(REAL *)fasp_mem_calloc(nnz,sizeof(REAL));

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

    memcpy(rows[0],rows[1],nnz*sizeof(INT));
    memcpy(cols[0],cols[1],nnz*sizeof(INT));
    memcpy(vals[0],vals[1],nnz*sizeof(REAL));
    nns[0]=A->col;
    nns[1]=nn;

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

    maxdif=0.;
    mdi=0;
    mdj=0;
    for (i=0;i<nnz;++i) {
        rows[0][i]=rows[1][i]-rows[0][i];
        if (rows[0][i]!=0) {
            type=-1;
            mdi=rows[1][i];
            break;
        }

        cols[0][i]=cols[1][i]-cols[0][i];
        if (cols[0][i]!=0) {
            type=-2;
            mdj=cols[1][i];
            break;
        }

        if (fabs(vals[0][i])>SMALLREAL||fabs(vals[1][i])>SMALLREAL) {
            dif=fabs(vals[1][i]-vals[0][i])/(fabs(vals[0][i])+fabs(vals[1][i]));
            if (dif>maxdif) {
                maxdif=dif;
                mdi=rows[0][i];
                mdj=cols[0][i];
            }
        }
    }

    if (maxdif>symmetry_tol) type=-3;

    switch (type) {
    case 0:
        printf("Matrix is symmetric with max relative difference is %1.3le\n",maxdif);
        break;
    case -1:
        printf("Matrix has nonsymmetric pattern, check the %d-th, %d-th and %d-th rows and cols\n",
               mdi-1,mdi,mdi+1);
        break;
    case -2:
        printf("Matrix has nonsymmetric pattern, check the %d-th, %d-th and %d-th cols and rows\n",
               mdj-1,mdj,mdj+1);
        break;
    case -3:
        printf("Matrix is nonsymmetric with max relative difference is %1.3le\n",maxdif);
        break;
    default:
        break;
    }

    fasp_mem_free(rowp);    rowp    = NULL;
    fasp_mem_free(rows[0]); rows[0] = NULL;
    fasp_mem_free(rows[1]); rows[1] = NULL;
    fasp_mem_free(cols[0]); cols[0] = NULL;
    fasp_mem_free(cols[1]); cols[1] = NULL;
    fasp_mem_free(vals[0]); vals[0] = NULL;
    fasp_mem_free(vals[1]); vals[1] = NULL;

    return type;
}

void fasp_check_dCSRmat (const dCSRmat *A)
{
    INT i;

    if ( (A->IA == NULL) || (A->JA == NULL) || (A->val == NULL) ) {
        printf("### ERROR: Something is wrong with the matrix!\n");
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    if ( A->row != A->col ) {
        printf("### ERROR: Non-square CSR matrix!\n");
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    if ( ( A->nnz <= 0 ) || ( A->row == 0 ) || ( A->col == 0 ) ) {
        printf("### ERROR: Empty CSR matrix!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    for ( i = 0; i < A->nnz; ++i ) {
        if ( ( A->JA[i] < 0 ) || ( A->JA[i] >= A->col ) ) {
            printf("### ERROR: Wrong CSR matrix format!\n");
            fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
        }
    }
}

SHORT fasp_check_iCSRmat (const iCSRmat *A)
{
    INT i;

    if ( (A->IA == NULL) || (A->JA == NULL) || (A->val == NULL) ) {
        printf("### ERROR: Something is wrong with the matrix!\n");
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    if (A->row != A->col) {
        printf("### ERROR: Non-square CSR matrix!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    if ( (A->nnz==0) || (A->row==0) || (A->col==0) ) {
        printf("### ERROR: Empty CSR matrix!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    for (i=0;i<A->nnz;++i) {
        if ( (A->JA[i]<0) || (A->JA[i]-A->col>=0) ) {
            printf("### ERROR: Wrong CSR matrix format!\n");
            fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
        }
    }

    return FASP_SUCCESS;
}

void fasp_check_ordering (dCSRmat *A)
{
    const INT n = A->col;
    INT i, j, j1, j2, start, end;

    for ( i=0; i<n; ++i ) {

        start = A->IA[i];
        end   = A->IA[i+1] - 1;

        for ( j=start; j<end-1; ++j ) {
            j1 = A->JA[j]; j2 = A->JA[j + 1];
            if ( j1 >= j2 ) {
                printf("### ERROR: Order in row %10d is wrong!  %10d, %10d\n", i, j1, j2);
                fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
            }
        }

    }

}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

m	Number of rows
n	Number of columns
nnz	Number of nonzeros
A	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

m	Number of rows
n	Number of columns
nnz	Number of nonzeros

Returns

A 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

A	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

A	Pointer to IJ matrix
offset	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.

```

#include <math.h>
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

dCOOmat fasp_dcoo_create (const INT  m,
                          const INT  n,
                          const INT  nnz)
{
    dCOOmat A;

    A.rowind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
    A.colind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
    A.val    = (REAL *)fasp_mem_calloc(nnz, sizeof(REAL));

    A.row = m; A.col = n; A.nnz = nnz;

    return A;
}

void fasp_dcoo_alloc (const INT  m,
                      const INT  n,
                      const INT  nnz,
                         dCOOmat    *A)
{

    if ( nnz > 0 ) {
        A->rowind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
        A->colind = (INT *)fasp_mem_calloc(nnz, sizeof(INT));
        A->val    = (REAL*)fasp_mem_calloc(nnz,sizeof(REAL));
    }
    else {
        A->rowind = NULL;
        A->colind = NULL;
        A->val    = NULL;
    }

    A->row = m; A->col = n; A->nnz = nnz;

    return;
}

void fasp_dcoo_free (dCOOmat *A)
{
    if (A==NULL) return;

    fasp_mem_free(A->rowind); A->rowind = NULL;
    fasp_mem_free(A->colind); A->colind = NULL;
    fasp_mem_free(A->val);    A->val    = NULL;
}

void fasp_dcoo_shift (dCOOmat    *A,
                         const INT  offset)
{
    const INT nnz = A->nnz;
    INT       i, *ai = A->rowind, *aj = A->colind;

    // Variables for OpenMP
    SHORT nthreads = 1, use_openmp = FALSE;
    INT myid, mybegin, myend;

#ifdef _OPENMP
    if (nnz > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend)
#endif
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
                ai[i]+=offset; aj[i]+=offset;
            }
        }
    }
    else {
        for (i=0;i<nnz;++i) {
            ai[i]+=offset; aj[i]+=offset;
        }
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 bandwith of matrix.
- dCSRmat fasp_dcsr_perm (dCSRmat ∗A, INT ∗P)

  Apply permutation of A, i.e. Aperm=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(aij)<=dtol.
- SHORT fasp_dcsr_compress_inplace (dCSRmat ∗A, const REAL dtol)

  Compress a CSR matrix A IN PLACE by dropping small entries abs(aij)<=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}AD^{-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 x 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

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

A	Input dCSRmat
rowmax	Max row nonzeros of A
groups	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

A	Input dCSRmat
S	Input iCSRmat Strong Coupled Matrix of A.
flags	Flags for the independent group
groups	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

A	Input dCSRmat
theta	Strength threshold parameter
rowmax	Max row nonzeros of A
groups	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

m	Number of rows
n	Number of columns
nnz	Number of nonzeros
A	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

A	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 abs(aij)<=dtol.

Parameters

A	Pointer to dCSRmat CSR matrix
B	Pointer to dCSRmat CSR matrix
dtol	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 abs(aij)<=dtol.

Parameters

A	Pointer to dCSRmat CSR matrix
dtol	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

A	Pointer to the dCSRmat matrix
B	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

m	Number of rows
n	Number of columns
nnz	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

A	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

A	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

A	Pointer to dCSRmat matrix
B	Pointer to dCSRmat matrix
Is	Pointer to selected rows
Js	Pointer to selected columns
m	Number of selected rows
n	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

n	Index of a column of A (0 <= n <= A.col-1)
A	Pointer to dCSRmat CSR matrix
col	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

n	Number of diagonal entries to get (if n=0, then get all diagonal entries)
A	Pointer to dCSRmat CSR matrix
diag	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

A	Input dCSRmat
flags	flags for the independent group
groups	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. Aperm=PAP' by the orders given in P.

Parameters

A	Pointer to the original dCSRmat matrix
P	Pointer to orders

Returns

The 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

A	Pointer to the original dCSRmat matrix
p	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

A	Pointer to the dCSRmat matrix
value	Set a value on diag(A) 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

A	Pointer to CSR matrix
offset	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

A	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

A	Pointer to the dCSRmat matrix
isym	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

A	Pointer to the dCSRmat matrix
diag	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

A	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

A	Pointer to the dCSRmat matrix
AT	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

row	Pointers of the rows of the matrix and its transpose
col	Pointers of the columns of the matrix and its transpose
val	Pointers to the values of the matrix and its transpose
nn	Pointer to the number of rows/columns of A and A'
tniz	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

A	Pointer to matrix in dCSRmat for transpose, INPUT
p	Permutation, INPUT
AT	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=transpose(A)p, where p is a permutation. If p=NULL then p=I is assumed. Applying twice this procedure one gets At=transpose(transpose(A)p)p = transpose(p)Ap, which is the same A with rows and columns permutted according to p.

If A=NULL, then only transposes/permutes the structure of A.

For p=NULL, applying this two times A-->AT-->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�C269, 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

A	Pointer to the iCSRmat matrix
B	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

m	Number of rows
n	Number of columns
nnz	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

A	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

A	Pointer to the iCSRmat matrix A
AT	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.

```
#include <math.h>
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"
```

```

#if MULTI_COLOR_ORDER
static void generate_S_theta(dCSRmat*, iCSRmat*, REAL);
#endif

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

dCSRmat fasp_dcsr_create(const INT m, const INT n, const INT nnz)
{
    dCSRmat A;

    if (m > 0) {
        A.IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    } else {
        A.IA = NULL;
    }

    if (n > 0) {
        A.JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    } else {
        A.JA = NULL;
    }

    if (nnz > 0) {
        A.val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
    } else {
        A.val = NULL;
    }

    A.row = m;
    A.col = n;
    A.nnz = nnz;

#if MULTI_COLOR_ORDER
    A.color = 0;
    A.IC    = NULL;
    A.ICMAP = NULL;
#endif

    return A;
}

iCSRmat fasp_icsr_create(const INT m, const INT n, const INT nnz)
{
    iCSRmat A;

    if (m > 0) {
        A.IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    } else {
        A.IA = NULL;
    }

    if (n > 0) {
        A.JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    } else {
        A.JA = NULL;
    }

    if (nnz > 0) {
        A.val = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    } else {
        A.val = NULL;
    }

    A.row = m;
    A.col = n;
    A.nnz = nnz;

    return A;
}

void fasp_dcsr_alloc(const INT m, const INT n, const INT nnz, dCSRmat* A)
{
    if (m <= 0 || n <= 0) {
        printf("### ERROR: Matrix dim %d, %d must be positive!  [%s]\n", m, n,
               __FUNCTION__);
        return;
    }

    if (m > 0) {
        A->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    } else {
        A->IA = NULL;
    }

    if (nnz > 0) {
        A->JA  = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
        A->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));
    } else {
        A->JA  = NULL;
        A->val = NULL;
    }

    A->row = m;
    A->col = n;
    A->nnz = nnz;

#if MULTI_COLOR_ORDER
    A->color = 0;
    A->IC    = NULL;
    A->ICMAP = NULL;
#endif

    return;
}

void fasp_dcsr_free(dCSRmat* A)
{
    if (A == NULL) return;

    fasp_mem_free(A->IA);
    A->IA = NULL;
    fasp_mem_free(A->JA);
    A->JA = NULL;
    fasp_mem_free(A->val);
    A->val = NULL;

#if MULTI_COLOR_ORDER
    fasp_mem_free(A->IC);
    A->IC = NULL;
    fasp_mem_free(A->ICMAP);
    A->ICMAP = NULL;
#endif

    A->col = 0;
    A->row = 0;
    A->nnz = 0;
    A      = NULL;
}

void fasp_icsr_free(iCSRmat* A)
{
    if (A == NULL) return;

    fasp_mem_free(A->IA);
    A->IA = NULL;
    fasp_mem_free(A->JA);
    A->JA = NULL;
    fasp_mem_free(A->val);
    A->val = NULL;
    A->col = 0;
    A->row = 0;
    A->nnz = 0;
    A      = NULL;
}

INT fasp_dcsr_bandwidth(const dCSRmat* A)
{
    const INT  row = A->row;
    const INT* ia  = A->IA;
    INT        i, max;

    for (max = i = 0; i < row; ++i) max = MAX(max, ia[i + 1] - ia[i]);

    return (max);
}

dCSRmat fasp_dcsr_perm(dCSRmat* A, INT* P)
{
    const INT   n = A->row, nnz = A->nnz;
    const INT * ia = A->IA, *ja = A->JA;
    const REAL* Aval = A->val;
    INT         i, j, k, jaj, i1, i2, start;
    SHORT       nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (MIN(n, nnz) > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    dCSRmat Aperm = fasp_dcsr_create(n, n, nnz);

    // form the transpose of P
    INT* Pt = (INT*)fasp_mem_calloc(n, sizeof(INT));

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) Pt[P[i]] = i;
        }
    } else {
        for (i = 0; i < n; ++i) Pt[P[i]] = i;
    }

    // compute IA of P*A (row permutation)
    Aperm.IA[0] = 0;
    for (i = 0; i < n; ++i) {
        k                = P[i];
        Aperm.IA[i + 1] = Aperm.IA[i] + (ia[k + 1] - ia[k]);
    }

    // perform actual P*A
    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i1, i2, k, start, j, jaj)
#endif
        for (myid = 0; myid < nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                i1    = Aperm.IA[i];
                i2    = Aperm.IA[i + 1] - 1;
                k     = P[i];
                start = ia[k];
                for (j = i1; j <= i2; ++j) {
                    jaj          = start + j - i1;
                    Aperm.JA[j]  = ja[jaj];
                    Aperm.val[j] = Aval[jaj];
                }
            }
        }
    } else {
        for (i = 0; i < n; ++i) {
            i1    = Aperm.IA[i];
            i2    = Aperm.IA[i + 1] - 1;
            k     = P[i];
            start = ia[k];
            for (j = i1; j <= i2; ++j) {
                jaj          = start + j - i1;
                Aperm.JA[j]  = ja[jaj];
                Aperm.val[j] = Aval[jaj];
            }
        }
    }

    // perform P*A*P' (column permutation)
    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, k, j)
#endif
        for (myid = 0; myid < nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
            for (k = mybegin; k < myend; ++k) {
                j           = Aperm.JA[k];
                Aperm.JA[k] = Pt[j];
            }
        }
    } else {
        for (k = 0; k < nnz; ++k) {
            j           = Aperm.JA[k];
            Aperm.JA[k] = Pt[j];
        }
    }

    fasp_mem_free(Pt);
    Pt = NULL;

    return (Aperm);
}

void fasp_dcsr_sort(dCSRmat* A)
{
    const INT n = A->col;
    INT       i, j, start, row_length;

    // temp memory for sorting rows of A
    INT * index, *ja;
    REAL* a;

    index = (INT*)fasp_mem_calloc(n, sizeof(INT));
    ja    = (INT*)fasp_mem_calloc(n, sizeof(INT));
    a     = (REAL*)fasp_mem_calloc(n, sizeof(REAL));

    for (i = 0; i < n; ++i) {
        start      = A->IA[i];
        row_length = A->IA[i + 1] - start;

        for (j = 0; j < row_length; ++j) index[j] = j;

        fasp_aux_iQuickSortIndex(&(A->JA[start]), 0, row_length - 1, index);

        for (j = 0; j < row_length; ++j) {
            ja[j] = A->JA[start + index[j]];
            a[j]  = A->val[start + index[j]];
        }

        for (j = 0; j < row_length; ++j) {
            A->JA[start + j]  = ja[j];
            A->val[start + j] = a[j];
        }
    }

    // clean up memory
    fasp_mem_free(index);
    index = NULL;
    fasp_mem_free(ja);
    ja = NULL;
    fasp_mem_free(a);
    a = NULL;
}

SHORT fasp_dcsr_getblk(const dCSRmat* A, const INT* Is, const INT* Js, const INT m,
                       const INT n, dCSRmat* B)
{
    SHORT use_openmp = FALSE;
    SHORT status     = FASP_SUCCESS;
    INT   i, j, k, nnz = 0;
    INT*  col_flag;

#ifdef _OPENMP
    INT stride_i, mybegin, myend, myid, nthreads;
    if (n > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    // create column flags
    col_flag = (INT*)fasp_mem_calloc(A->col, sizeof(INT));

    B->row = m;
    B->col = n;

    B->IA  = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));
    B->JA  = (INT*)fasp_mem_calloc(A->nnz, sizeof(INT));
    B->val = (REAL*)fasp_mem_calloc(A->nnz, sizeof(REAL));

#if MULTI_COLOR_ORDER
    B->color = 0;
    B->IC    = NULL;
    B->ICMAP = NULL;
#endif

    if (use_openmp) {
#ifdef _OPENMP
        stride_i = n / nthreads;
#pragma omp parallel private(myid, mybegin, myend, i) num_threads(nthreads)
        {
            myid    = omp_get_thread_num();
            mybegin = myid * stride_i;
            if (myid < nthreads - 1)
                myend = mybegin + stride_i;
            else
                myend = n;
            for (i = mybegin; i < myend; ++i) {
                col_flag[Js[i]] = i + 1;
            }
        }
#endif
    } else {
        for (i = 0; i < n; ++i) col_flag[Js[i]] = i + 1;
    }

    // Count nonzeros for sub matrix and fill in
    B->IA[0] = 0;
    for (i = 0; i < m; ++i) {
        for (k = A->IA[Is[i]]; k < A->IA[Is[i] + 1]; ++k) {
            j = A->JA[k];
            if (col_flag[j] > 0) {
                B->JA[nnz]  = col_flag[j] - 1;
                B->val[nnz] = A->val[k];
                nnz++;
            }
        } /* end for k */
        B->IA[i + 1] = nnz;
    } /* end for i */
    B->nnz = nnz;

    // re-allocate memory space
    B->JA  = (INT*)fasp_mem_realloc(B->JA, sizeof(INT) * nnz);
    B->val = (REAL*)fasp_mem_realloc(B->val, sizeof(REAL) * nnz);

    fasp_mem_free(col_flag);
    col_flag = NULL;

    return (status);
}

void fasp_dcsr_getdiag(INT n, const dCSRmat* A, dvector* diag)
{
    INT i, k, j, ibegin, iend;

    SHORT nthreads = 1, use_openmp = FALSE;

    if (n == 0 || n > A->row || n > A->col) n = MIN(A->row, A->col);

#ifdef _OPENMP
    if (n > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    fasp_dvec_alloc(n, diag);

    if (use_openmp) {
        INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, ibegin, iend, k, j)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                ibegin = A->IA[i];
                iend   = A->IA[i + 1];
                for (k = ibegin; k < iend; ++k) {
                    j = A->JA[k];
                    if ((j - i) == 0) {
                        diag->val[i] = A->val[k];
                        break;
                    } // end if
                }     // end for k
            }         // end for i
        }
    } else {
        for (i = 0; i < n; ++i) {
            ibegin = A->IA[i];
            iend   = A->IA[i + 1];
            for (k = ibegin; k < iend; ++k) {
                j = A->JA[k];
                if ((j - i) == 0) {
                    diag->val[i] = A->val[k];
                    break;
                } // end if
            }     // end for k
        }         // end for i
    }
}

void fasp_dcsr_getcol(const INT n, const dCSRmat* A, REAL* col)
{
    INT i, j, row_begin, row_end;
    INT nrow = A->row, ncol = A->col;
    INT status = FASP_SUCCESS;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (nrow > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    // check the column index n
    if (n < 0 || n >= ncol) {
        printf("### ERROR: Illegal column index %d!  [%s]\n", n, __FUNCTION__);
        status = ERROR_DUMMY_VAR;
        goto FINISHED;
    }

    // get the column
    if (use_openmp) {
        INT mybegin, myend, myid;

#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, j, row_begin, row_end)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                col[i]    = 0.0;
                row_begin = A->IA[i];
                row_end   = A->IA[i + 1];
                for (j = row_begin; j < row_end; ++j) {
                    if (A->JA[j] == n) {
                        col[i] = A->val[j];
                    }
                } // end for j
            }     // end for i
        }
    } else {
        for (i = 0; i < nrow; ++i) {
            // set the entry to zero
            col[i]    = 0.0;
            row_begin = A->IA[i];
            row_end   = A->IA[i + 1];
            for (j = row_begin; j < row_end; ++j) {
                if (A->JA[j] == n) {
                    col[i] = A->val[j];
                }
            } // end for j
        }     // end for i
    }

FINISHED:
    fasp_chkerr(status, __FUNCTION__);
```

```
}

void fasp_dcsr_diagpref(dCSRmat* A)
{
    const INT num_rowsA = A->row;
    REAL*     A_data    = A->val;
    INT*      A_i       = A->IA;
    INT*      A_j       = A->JA;

    // Local variable
    INT  i, j;
    INT  tempi, row_size;
    REAL tempd;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend, ibegin, iend;
    INT nthreads = fasp_get_num_threads();
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

#ifdef _OPENMP
    if (num_rowsA > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, i, j, ibegin, iend, tempi, tempd, mybegin, myend)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, num_rowsA, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                ibegin = A_i[i];
                iend   = A_i[i + 1];
                // check whether the first entry is already diagonal
                if (A_j[ibegin] != i) {
                    for (j = ibegin + 1; j < iend; j++) {
                        if (A_j[j] == i) {
#if DEBUG_MODE > 2
                            printf("### DEBUG: Switch entry_%d with entry_0\n", j);
#endif
                            tempi       = A_j[ibegin];
                            A_j[ibegin] = A_j[j];
                            A_j[j]      = tempi;

                            tempd          = A_data[ibegin];
                            A_data[ibegin] = A_data[j];
                            A_data[j]      = tempd;
                            break;
                        }
                    }
                    if (j == iend) {
                        printf("### ERROR: Diagonal entry %d is zero!\n", i);
                        fasp_chkerr(ERROR_MISC, __FUNCTION__);
                    }
                }
            }
        }
    } else {
#endif
        for (i = 0; i < num_rowsA; i++) {
            row_size = A_i[i + 1] - A_i[i];
            // check whether the first entry is already diagonal
            if (A_j[0] != i) {
                for (j = 1; j < row_size; j++) {
                    if (A_j[j] == i) {
#if DEBUG_MODE > 2
                        printf("### DEBUG: Switch entry_%d with entry_0\n", j);
#endif
                        tempi  = A_j[0];
                        A_j[0] = A_j[j];
                        A_j[j] = tempi;

                        tempd     = A_data[0];
                        A_data[0] = A_data[j];
                        A_data[j] = tempd;

                        break;
                    }
                }
                if (j == row_size) {
                    printf("### ERROR: Diagonal entry %d is zero!\n", i);
                    fasp_chkerr(ERROR_MISC, __FUNCTION__);
                }
            }
            A_j += row_size;
            A_data += row_size;
        }
#ifdef _OPENMP
    }
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
}

SHORT fasp_dcsr_regdiag(dCSRmat* A, const REAL value)
{
    const INT  m  = A->row;
    const INT *ia = A->IA, *ja = A->JA;
    REAL*      aj = A->val;

    // Local variables
    INT   i, j, k, begin_row, end_row;
    SHORT status = ERROR_UNKNOWN;

    for (i = 0; i < m; ++i) {
        begin_row = ia[i];
        end_row   = ia[i + 1];
        for (k = begin_row; k < end_row; ++k) {
            j = ja[k];
            if (i == j) {
                if (aj[k] < 0.0)
                    goto FINISHED;
                else if (aj[k] < SMALLREAL)
                    aj[k] = value;
            }
        } // end for k
    }     // end for i

    status = FASP_SUCCESS;

FINISHED:
    return status;
}

void fasp_icsr_cp(const iCSRmat* A, iCSRmat* B)
{
    B->row = A->row;
    B->col = A->col;
    B->nnz = A->nnz;

    fasp_iarray_cp(A->row + 1, A->IA, B->IA);
    fasp_iarray_cp(A->nnz, A->JA, B->JA);
    fasp_iarray_cp(A->nnz, A->val, B->val);
}

void fasp_dcsr_cp(const dCSRmat* A, dCSRmat* B)
{
    B->row = A->row;
    B->col = A->col;
    B->nnz = A->nnz;

    fasp_iarray_cp(A->row + 1, A->IA, B->IA);
    fasp_iarray_cp(A->nnz, A->JA, B->JA);
    fasp_darray_cp(A->nnz, A->val, B->val);
}

void fasp_icsr_trans(const iCSRmat* A, iCSRmat* AT)
{
    const INT n = A->row, m = A->col, nnz = A->nnz, m1 = m - 1;

    // Local variables
    INT i, j, k, p;
    INT ibegin, iend;

#if DEBUG_MODE > 1
    printf("### DEBUG: m=%d, n=%d, nnz=%d\n", m, n, nnz);
#endif

    AT->row = m;
    AT->col = n;
    AT->nnz = nnz;

    AT->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));

    AT->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));

    if (A->val) {
        AT->val = (INT*)fasp_mem_calloc(nnz, sizeof(INT));
    } else {
        AT->val = NULL;
    }

    // first pass:  find the Number of nonzeros in the first m-1 columns of A
    // Note:  these Numbers are stored in the array AT.IA from 1 to m-1
    fasp_iarray_set(m + 1, AT->IA, 0);

    for (j = 0; j < nnz; ++j) {
        i = A->JA[j]; // column Number of A = row Number of A'
        if (i < m1) AT->IA[i + 2]++;
    }

    for (i = 2; i <= m; ++i) AT->IA[i] += AT->IA[i - 1];

    // second pass:  form A'
    if (A->val != NULL) {
        for (i = 0; i < n; ++i) {
            ibegin = A->IA[i];
            iend   = A->IA[i + 1];
            for (p = ibegin; p < iend; p++) {
                j           = A->JA[p] + 1;
                k           = AT->IA[j];
                AT->JA[k]   = i;
                AT->val[k] = A->val[p];
                AT->IA[j]   = k + 1;
            } // end for p
        }     // end for i
    } else {
        for (i = 0; i < n; ++i) {
            ibegin = A->IA[i];
            iend   = A->IA[i + 1];
            for (p = ibegin; p < iend; p++) {
                j         = A->JA[p] + 1;
                k         = AT->IA[j];
                AT->JA[k] = i;
                AT->IA[j] = k + 1;
            } // end for p
        }     // end for i
    }         // end if
}

INT fasp_dcsr_trans(const dCSRmat* A, dCSRmat* AT)
{
    const INT n = A->row, m = A->col, nnz = A->nnz;

    // Local variables
    INT i, j, k, p;

    AT->row = m;
    AT->col = n;
    AT->nnz = nnz;

    AT->IA = (INT*)fasp_mem_calloc(m + 1, sizeof(INT));

    AT->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));

    if (A->val) {
        AT->val = (REAL*)fasp_mem_calloc(nnz, sizeof(REAL));

    } else {
        AT->val = NULL;
    }

#if MULTI_COLOR_ORDER
    AT->color = 0;
    AT->IC    = NULL;
    AT->ICMAP = NULL;
#endif

    // first pass:  find the Number of nonzeros in the first m-1 columns of A
    // Note:  these Numbers are stored in the array AT.IA from 1 to m-1

    // fasp_iarray_set(m+1, AT->IA, 0);
    memset(AT->IA, 0, sizeof(INT) * (m + 1));

    for (j = 0; j < nnz; ++j) {
        i = A->JA[j]; // column Number of A = row Number of A'
        if (i < m - 1) AT->IA[i + 2]++;
    }

    for (i = 2; i <= m; ++i) AT->IA[i] += AT->IA[i - 1];

    // second pass:  form A'
    if (A->val) {
        for (i = 0; i < n; ++i) {
            INT ibegin = A->IA[i], iend = A->IA[i + 1];
            for (p = ibegin; p < iend; p++) {
                j          = A->JA[p] + 1;
                k          = AT->IA[j];
                AT->JA[k]  = i;
                AT->val[k] = A->val[p];
                AT->IA[j]  = k + 1;
            } // end for p
        }     // end for i
    } else {
        for (i = 0; i < n; ++i) {
            INT ibegin = A->IA[i], iend1 = A->IA[i + 1];
            for (p = ibegin; p < iend1; p++) {
                j         = A->JA[p] + 1;
                k         = AT->IA[j];
                AT->JA[k] = i;
                AT->IA[j] = k + 1;
            } // end for p
        }     // end of i
    }         // end if

    return FASP_SUCCESS;
}

void fasp_dcsr_transpose(INT* row[2], INT* col[2], REAL* val[2], INT* nn, INT* tniz)
{
    const INT nca = nn[1]; // Number of columns

    INT* izc    = (INT*)fasp_mem_calloc(nn[1], sizeof(INT));
    INT* izcaux = (INT*)fasp_mem_calloc(nn[1], sizeof(INT));

    // Local variables
    INT i, m, itmp;

    // first pass:  to set order right
    for (i = 0; i < tniz[0]; ++i) izc[col[0][i]]++;

    izcaux[0] = 0;
    for (i = 1; i < nca; ++i) izcaux[i] = izcaux[i - 1] + izc[i - 1];

    // second pass:  form transpose
    memset(izc, 0, nca * sizeof(INT));

    for (i = 0; i < tniz[0]; ++i) {
        m             = col[0][i];
        itmp          = izcaux[m] + izc[m];
        row[1][itmp] = m;
        col[1][itmp] = row[0][i];
        val[1][itmp] = val[0][i];
        izc[m]++;
    }

    fasp_mem_free(izc);
    izc = NULL;
    fasp_mem_free(izcaux);
    izcaux = NULL;
}

void fasp_dcsr_compress(const dCSRmat* A, dCSRmat* B, const REAL dtol)
{
    INT i, j, k;
    INT ibegin, iend1;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (B->nnz > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    INT* index = (INT*)fasp_mem_calloc(A->nnz, sizeof(INT));

    B->row = A->row;
    B->col = A->col;

    B->IA = (INT*)fasp_mem_calloc(A->row + 1, sizeof(INT));

    B->IA[0] = A->IA[0];

    // first pass:  determine the size of B
    k = 0;
    for (i = 0; i < A->row; ++i) {
        ibegin = A->IA[i];
        iend1  = A->IA[i + 1];
        for (j = ibegin; j < iend1; ++j)
            if (ABS(A->val[j]) > dtol) {
                index[k] = j;
                ++k;
            } /* end of j */
        B->IA[i + 1] = k;
    } /* end of i */
    B->nnz = k;
    B->JA  = (INT*)fasp_mem_calloc(B->nnz, sizeof(INT));
    B->val = (REAL*)fasp_mem_calloc(B->nnz, sizeof(REAL));

    // second pass:  generate the index and element to B
    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, i, mybegin, myend)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, B->nnz, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                B->JA[i]  = A->JA[index[i]];
                B->val[i] = A->val[index[i]];
            }
        }
    } else {
        for (i = 0; i < B->nnz; ++i) {
            B->JA[i]  = A->JA[index[i]];
            B->val[i] = A->val[index[i]];
        }
    }

    fasp_mem_free(index);
    index = NULL;
}

SHORT fasp_dcsr_compress_inplace(dCSRmat* A, const REAL dtol)
{
    const INT row = A->row;
    const INT nnz = A->nnz;

    INT   i, j, k;
    INT   ibegin, iend = A->IA[0];
    SHORT status = FASP_SUCCESS;
    k            = 0;
    for (i = 0; i < row; ++i) {
        ibegin = iend;
        iend   = A->IA[i + 1];
        for (j = ibegin; j < iend; ++j)
            if (ABS(A->val[j]) > dtol || i == A->JA[j]) {
                A->JA[k]  = A->JA[j];
                A->val[k] = A->val[j];
                ++k;
            } /* end of j */
        A->IA[i + 1] = k;
    } /* end of i */

    if (k <= nnz) {
        A->nnz = k;
        A->JA  = (INT*)fasp_mem_realloc(A->JA, k * sizeof(INT));
        A->val = (REAL*)fasp_mem_realloc(A->val, k * sizeof(REAL));
    } else {
        printf("### WARNING: Size of compressed matrix is bigger than original!\n");
        status = ERROR_UNKNOWN;
    }

    return (status);
}

void fasp_dcsr_shift(dCSRmat* A, const INT offset)
{
    const INT nnz = A->nnz;
    const INT n   = A->row + 1;
    INT       i, *ai = A->IA, *aj = A->JA;
    SHORT     nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (MIN(n, nnz) > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                ai[i] += offset;
            }
        }
    } else {
        for (i = 0; i < n; ++i) ai[i] += offset;
    }

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, nnz, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                aj[i] += offset;
            }
        }
    } else {
        for (i = 0; i < nnz; ++i) aj[i] += offset;
    }
}

void fasp_dcsr_symdiagscale(dCSRmat* A, const dvector* diag)
{
    // information about matrix A
    const INT  n    = A->row;
    const INT* IA   = A->IA;
    const INT* JA   = A->JA;
    REAL*      val = A->val;
    REAL*      work;

    SHORT nthreads = 1, use_openmp = FALSE;

    // local variables
    INT i, j, k, row_start, row_end;

#ifdef _OPENMP
    if (n > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (diag->row != n) {
        printf("### ERROR: Size of diag = %d != size of matrix = %d!", diag->row, n);
        fasp_chkerr(ERROR_MISC, __FUNCTION__);
    }

    // work space
    work = (REAL*)fasp_mem_calloc(n, sizeof(REAL));

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) work[i] = sqrt(diag->val[i]);
        }
    } else {
        // square root of diagonal entries
        for (i = 0; i < n; i++) work[i] = sqrt(diag->val[i]);
    }

    if (use_openmp) {
        INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, row_start, row_end, i, j, k)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                row_start = IA[i];
                row_end   = IA[i + 1];
                for (j = row_start; j < row_end; j++) {
                    k      = JA[j];
                    val[j] = val[j] / (work[i] * work[k]);
                }
            }
        }
    } else {
        // main loop
        for (i = 0; i < n; i++) {
            row_start = IA[i];
            row_end   = IA[i + 1];
            for (j = row_start; j < row_end; j++) {
                k      = JA[j];
                val[j] = val[j] / (work[i] * work[k]);
            }
        }
    }

    // free work space
    fasp_mem_free(work);
    work = NULL;
}

dCSRmat fasp_dcsr_sympart(dCSRmat* A)
{
    // local variable
    dCSRmat AT;

    // return variable
    dCSRmat SA;

#if MULTI_COLOR_ORDER
    AT.IC    = NULL;
    SA.IC    = NULL;
    AT.ICMAP = NULL;
    SA.ICMAP = NULL;
#endif

    // get the transpose of A
    fasp_dcsr_trans(A, &AT);

    // get symmetrized A
    fasp_blas_dcsr_add(A, 1.0, &AT, 0.0, &SA);

    // clean
    fasp_dcsr_free(&AT);

    // return
    return SA;
}

void fasp_dcsr_transz(dCSRmat* A, INT* p, dCSRmat* AT)
{
    /* tested for permutation and transposition */
    /* transpose or permute; if A.val is null ===> transpose the
structure only */
    const INT   n = A->row, m = A->col, nnz = A->nnz;
    const INT * ia = NULL, *ja = NULL;
    const REAL* a  = NULL;
    INT         m1 = m + 1;
    ia              = A->IA;
    ja              = A->JA;
    a               = A->val;
    /* introducing few extra pointers hould not hurt too much the speed */
    INT * iat = NULL, *jat = NULL;
    REAL* at = NULL;

    /* loop variables */
    INT i, j, jp, pi, iabeg, iaend, k;

    /* initialize */
    AT->row = m;
    AT->col = n;
    AT->nnz = nnz;

    /* all these should be allocated or change this to allocate them here */
    iat = AT->IA;
    jat = AT->JA;
    at  = AT->val;
    for (i = 0; i < m1; ++i) iat[i] = 0;
    iaend = ia[n];
    for (i = 0; i < iaend; ++i) {
        j = ja[i] + 2;
        if (j < m1) iat[j]++;
    }
    iat[0] = 0;
    iat[1] = 0;
    if (m != 1) {
        for (i = 2; i < m1; ++i) {
            iat[i] += iat[i - 1];
        }
    }

    if (p && a) {
        /* so we permute and also use matrix entries */
        for (i = 0; i < n; ++i) {
            pi    = p[i];
            iabeg = ia[pi];
            iaend = ia[pi + 1];
            if (iaend > iabeg) {
                for (jp = iabeg; jp < iaend; ++jp) {
                    j      = ja[jp] + 1;
                    k      = iat[j];
                    jat[k] = i;
                    at[k]  = a[jp];
                    iat[j] = k + 1;
                }
            }
        }
    } else if (a && !p) {
        /* transpose values, no permutation */
        for (i = 0; i < n; ++i) {
            iabeg = ia[i];
            iaend = ia[i + 1];
            if (iaend > iabeg) {
                for (jp = iabeg; jp < iaend; ++jp) {
                    j      = ja[jp] + 1;
                    k      = iat[j];
                    jat[k] = i;
                    at[k]  = a[jp];
                    iat[j] = k + 1;
                }
            }
        }
    } else if (!a && p) {
        /* Only integers and permutation (only a is null) */
        for (i = 0; i < n; ++i) {
            pi    = p[i];
            iabeg = ia[pi];
            iaend = ia[pi + 1];
            if (iaend > iabeg) {
                for (jp = iabeg; jp < iaend; ++jp) {
                    j      = ja[jp] + 1;
                    k      = iat[j];
                    jat[k] = i;
                    iat[j] = k + 1;
                }
            }
        }
    } else {
        /* Only integers and no permutation (both a and p are null */
        for (i = 0; i < n; ++i) {
            iabeg = ia[i];
            iaend = ia[i + 1];
            if (iaend > iabeg) {
                for (jp = iabeg; jp < iaend; ++jp) {
                    j      = ja[jp] + 1;
                    k      = iat[j];
                    jat[k] = i;
                    iat[j] = k + 1;
                }
            }
        }
    }

    return;
}

dCSRmat fasp_dcsr_permz(dCSRmat* A, INT* p)
{
    const INT n = A->row, nnz = A->nnz;
    dCSRmat   Aperm1, Aperm;

    Aperm1 = fasp_dcsr_create(n, n, nnz);
    Aperm  = fasp_dcsr_create(n, n, nnz);

    fasp_dcsr_transz(A, p, &Aperm1);
    fasp_dcsr_transz(&Aperm1, p, &Aperm);

    // clean up
    fasp_dcsr_free(&Aperm1);

    return (Aperm);
}

void fasp_dcsr_sortz(dCSRmat* A, const SHORT isym)
{
    const INT n = A->row, m = A->col, nnz = A->nnz;
    dCSRmat   AT = fasp_dcsr_create(m, n, nnz);

    /* watch carefully who is a pointer and who is not in fasp_dcsr_transz() */
    fasp_dcsr_transz(A, NULL, &AT);

    /* if the matrix is symmetric, then only one transpose is needed
and now we just copy */
    if ((m == n) && (isym))
        fasp_dcsr_cp(&AT, A);
    else
        fasp_dcsr_transz(&AT, NULL, A);

    // clean up
    fasp_dcsr_free(&AT);
}

void fasp_dcsr_multicoloring(dCSRmat* A, INT* flags, INT* groups)
{
#if MULTI_COLOR_ORDER
    INT  k, i, j, pre, group;
    INT  iend;
    INT  icount;
    INT  front, rear;
    INT  n    = A->row;
    INT* IA   = A->IA;
    INT* JA   = A->JA;
    INT* cq   = (INT*)malloc(sizeof(INT) * (n + 1));
    INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));

#ifdef _OPENMP
#pragma omp parallel for private(k)
#endif
    for (k = 0; k < n; k++) cq[k] = k;

    group = 0;
    for (k = 0; k < n; k++) {
        if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
    }

    A->IC    = (INT*)malloc(sizeof(INT) * (group + 2));
    A->ICMAP = (INT*)malloc(sizeof(INT) * (n));

    front = n - 1;
    rear  = n - 1;

    memset(newr, -1, sizeof(INT) * (n + 1));
    memset(A->ICMAP, 0, sizeof(INT) * n);

    group     = 0;
    icount    = 0;
    A->IC[0] = 0;
    pre       = 0;

    do {
        front++;
        if (front == n) front = 0;
        i = cq[front];
        if (i <= pre) {
            A->IC[group]      = icount;
            A->ICMAP[icount] = i;
            group++;
            icount++;
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        } else if (newr[i] == group) {
            rear++;
            if (rear == n) rear = 0;
            cq[rear] = i;
        } else {
            A->ICMAP[icount] = i;
            icount++;
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        }
        pre = i;

    } while (rear != front);

    A->IC[group] = icount;
    A->color     = group;
    free(cq);
    free(newr);
    *groups = group;
#else
    printf("### ERROR: %s has not been defined!\n", __FUNCTION__);
#endif
}

void dCSRmat_Multicoloring(dCSRmat* A, INT* rowmax, INT* groups)
{
#if MULTI_COLOR_ORDER
    INT  k, i, j, pre, group;
    INT  igold, iend, iavg;
    INT  icount;
    INT  front, rear;
    INT  n  = A->row;
    INT* IA = A->IA;
    INT* JA = A->JA;

    INT* cq   = (INT*)malloc(sizeof(INT) * (n + 1));
    INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));

    for (k = 0; k < n; k++) cq[k] = k;

    group = 0;

    for (k = 0; k < n; k++) {
        if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
    }
    *rowmax = group;
#if 0
    iavg = IA[n]/n ;
    igold = (INT)MAX(iavg,group*0.618) +1;
    igold =group ;
#endif

    A->IC    = (INT*)malloc(sizeof(INT) * (group + 2));
    A->ICMAP = (INT*)malloc(sizeof(INT) * (n));

    front = n - 1;
    rear  = n - 1;

    memset(newr, -1, sizeof(INT) * (n + 1));
    memset(A->ICMAP, 0, sizeof(INT) * n);

    group    = 0;
    icount   = 0;
    A->IC[0] = 0;
    pre      = 0;

    do {
        // front = (front+1)%n;
        front++;
        if (front == n) front = 0; // front = front < n ?  front :  0 ;
        i = cq[front];

        if (i <= pre) {
            A->IC[group]      = icount;
            A->ICMAP[icount] = i;
            group++;
            icount++;
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        } else if (newr[i] == group) {
            // rear = (rear +1)%n;
            rear++;
            if (rear == n) rear = 0;
            cq[rear] = i;
        } else {
            A->ICMAP[icount] = i;
            icount++;
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        }
        pre = i;
    } while (rear != front);

    A->IC[group] = icount;
    A->color     = group;

#if 0
    for(i=0; i < A->color; i++ ){
        for(j=A -> IC[i]; j < A-> IC[i+1];j++)
            printf("color %d  ICMAP[%d] = %d \n", i,j,A-> ICMAP[j]);
            printf( "A.color = %d A.row= %d %d\n",A -> color,A -> row,A-> IC[i+1] - A-> IC[i] );
        getchar();
    }
#endif

    // 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 );
    free(cq);
    free(newr);
    *groups = group;
#endif
}

#if MULTI_COLOR_ORDER
static void generate_S_theta(dCSRmat* A, iCSRmat* S, REAL theta)
{
    const INT row = A->row, col = A->col;
    const INT row_plus_one = row + 1;
    const INT nnz          = A->IA[row] - A->IA[0];

    INT   index, i, j, begin_row, end_row;
    INT * ia = A->IA, *ja = A->JA;
    REAL* aj = A->val;

    // get the diagnal entry of A
    // dvector diag; fasp_dcsr_getdiag(0, A, &diag);

    /* generate S */
    REAL row_abs_sum;

    // copy the structure of A to S
    S->row = row;
    S->col = col;
    S->nnz = nnz;
    S->val = NULL;

    S->IA = (INT*)fasp_mem_calloc(row_plus_one, sizeof(INT));

    S->JA = (INT*)fasp_mem_calloc(nnz, sizeof(INT));

    fasp_iarray_cp(row_plus_one, ia, S->IA);
    fasp_iarray_cp(nnz, ja, S->JA);
```

```

#ifdef _OPENMP
#pragma omp parallel for private(i, j, begin_row, end_row, row_abs_sum)
#endif
    for (i = 0; i < row; ++i) {
        /* compute scaling factor and row sum */
        row_abs_sum = 0;
        begin_row   = ia[i];
        end_row     = ia[i + 1];
        for (j = begin_row; j < end_row; j++) {
            row_abs_sum += ABS(aj[j]);
        }
        row_abs_sum = row_abs_sum * theta;

        /* deal with  the element of S */
        for (j = begin_row; j < end_row; j++) {
            if ((row_abs_sum >= ABS(aj[j])) && (ja[j] != i)) {
                S->JA[j] = -1;
            }
        }
    } // end for i

    /* Compress the strength matrix */
    index = 0;
    for (i = 0; i < row; ++i) {
        S->IA[i]  = index;
        begin_row = ia[i];
        end_row   = ia[i + 1] - 1;
        for (j = begin_row; j <= end_row; j++) {
            if (S->JA[j] > -1) {
                S->JA[index] = S->JA[j];
                index++;
            }
        }
    }

    if (index > 0) {
        S->IA[row] = index;
        S->nnz     = index;
        S->JA      = (INT*)fasp_mem_realloc(S->JA, index * sizeof(INT));
    } else {
        S->nnz = 0;
        S->JA  = NULL;
    }
}
#endif

void dCSRmat_Multicoloring_Strong_Coupled(dCSRmat* A, iCSRmat* S, INT* flags,
                                          INT* groups)
{
#if MULTI_COLOR_ORDER
    INT  k, i, j, pre, group;
    INT  igold, iend, iavg;
    INT  icount;
    INT  front, rear;
    INT  n  = A->row;
    INT* IA = S->IA;
    INT* JA = S->JA;

    INT* cq   = (INT*)malloc(sizeof(INT) * (n + 1));
    INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));

#ifdef _OPENMP
#pragma omp parallel for private(k)
#endif
    for (k = 0; k < n; k++) {
        cq[k] = k;
    }
    group = 0;
    for (k = 0; k < n; k++) {
        if ((IA[k + 1] - IA[k]) > group) group = IA[k + 1] - IA[k];
    }
    *flags = group;
#if 1
    iavg  = IA[n] / n;
    igold = (INT)MAX(iavg, group * 0.618) + 1;
    igold = group;
#endif

    A->IC    = (INT*)malloc(sizeof(INT) * (group + 2));
    A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));

    front = n - 1;
    rear  = n - 1;

    memset(newr, -1, sizeof(INT) * (n + 1));
    memset(A->ICMAP, 0, sizeof(INT) * n);

    group    = 0;
    icount   = 0;
    A->IC[0] = 0;
    pre      = 0;

    do {
        // front = (front+1)%n;
        front++;
        if (front == n) front = 0; // front = front < n ?  front :  0 ;
        i = cq[front];

        if (i <= pre) {
            A->IC[group]     = icount;
            A->ICMAP[icount] = i;
            group++;
            icount++;
#if 0
            if ((IA[i+1]-IA[i]) > igold)
                iend = MIN(IA[i+1], (IA[i] + igold));
        else
#endif
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        } else if (newr[i] == group) {
            // rear = (rear +1)%n;
            rear++;
            if (rear == n) rear = 0;
            cq[rear] = i;
        } else {
            A->ICMAP[icount] = i;
            icount++;
#if 0
            if ((IA[i+1] - IA[i]) > igold)  iend =MIN(IA[i+1], (IA[i] + igold));
            else
#endif
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        }
        pre = i;

    } while (rear != front);

    A->IC[group] = icount;
    A->color     = group;

#if 0
    for(i=0; i < A->color; i++ ){
        for(j=A -> IC[i]; j < A-> IC[i+1];j++)
            printf("color %d  ICMAP[%d] = %d \n", i,j,A-> ICMAP[j]);
        printf( "A.color = %d A.row= %d %d\n",A -> color,A -> row,A-> IC[i+1] - A-> IC[i] );
        getchar();
    }
#endif
    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);
    free(cq);
    free(newr);
    *groups = group;
#endif
}

void dCSRmat_Multicoloring_Theta(dCSRmat* A, REAL theta, INT* rowmax, INT* groups)
{
#if MULTI_COLOR_ORDER
    INT k, i, j, pre, group;
    INT igold, iend, iavg;
    INT icount;
    INT front, rear;
    INT n = A->row;
    //-----------------------------------------------------------------------
    iCSRmat S;
    INT *   IA, *JA;
    if (theta > 0 && theta < 1.0) {
        generate_S_theta(A, &S, theta);
        IA = S.IA;
        JA = S.JA;
    } else if (theta == 1.0) {

        A->IC    = (INT*)malloc(sizeof(INT) * 2);
        A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));
        A->IC[0] = 0;
        A->IC[1] = n;
#ifdef _OPENMP
#pragma omp parallel for private(k)
#endif
        for (k = 0; k < n; k++) A->ICMAP[k] = k;

        A->color = 1;
        *groups  = 1;
        *rowmax  = 1;
        printf("Theta = %lf \n", theta);

        return;

    } else {
        IA = A->IA;
        JA = A->JA;
    }
    //---------------------------------------------------------------------------
    INT* cq   = (INT*)malloc(sizeof(INT) * (n + 1));
    INT* newr = (INT*)malloc(sizeof(INT) * (n + 1));

#ifdef _OPENMP
#pragma omp parallel for private(k)
#endif
    for (k = 0; k < n; k++) {
        cq[k] = k;
    }
    group = 0;
    for (k = 0; k < n; k++) {
        if ((A->IA[k + 1] - A->IA[k]) > group) group = A->IA[k + 1] - A->IA[k];
    }
    *rowmax = group;

#if 0
    iavg = IA[n]/n ;
    igold = (INT)MAX(iavg,group*0.618) +1;
    igold = group ;
#endif

    A->IC    = (INT*)malloc(sizeof(INT) * (group + 2));
    A->ICMAP = (INT*)malloc(sizeof(INT) * (n + 1));

    front = n - 1;
    rear  = n - 1;

    memset(newr, -1, sizeof(INT) * (n + 1));
    memset(A->ICMAP, 0, sizeof(INT) * n);

    group    = 0;
    icount   = 0;
    A->IC[0] = 0;
    pre      = 0;

    do {
        // front = (front+1)%n;
        front++;
        if (front == n) front = 0; // front = front < n ?  front :  0 ;
        i = cq[front];

        if (i <= pre) {
            A->IC[group]     = icount;
            A->ICMAP[icount] = i;
            group++;
            icount++;
#if 0
            if ((IA[i+1]-IA[i]) > igold)
                iend = MIN(IA[i+1], (IA[i] + igold));
        else
#endif
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        } else if (newr[i] == group) {
            // rear = (rear +1)%n;
            rear++;
            if (rear == n) rear = 0;
            cq[rear] = i;
        } else {
            A->ICMAP[icount] = i;
            icount++;
#if 0
            if ((IA[i+1] - IA[i]) > igold)  iend =MIN(IA[i+1], (IA[i] + igold));
            else
#endif
            iend = IA[i + 1];
            for (j = IA[i]; j < iend; j++) newr[JA[j]] = group;
        }
        pre = i;

        //    printf("pre = %d\n",pre);
    } while (rear != front);

    //    printf("group\n");
    A->IC[group] = icount;
    A->color     = group;

#if 0
    for(i=0; i < A->color; i++ ){
        for(j=A -> IC[i]; j < A-> IC[i+1];j++)
            printf("color %d  ICMAP[%d] = %d \n", i,j,A-> ICMAP[j]);
        printf( "A.color = %d A.row= %d %d\n",A -> color,A -> row,A-> IC[i+1] - A-> IC[i] );
        getchar();
    }
    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 );
#endif
    free(cq);
    free(newr);
    if (theta > 0) {
        fasp_mem_free(S.IA);
        fasp_mem_free(S.JA);
    }
    *groups = group;
#endif
    return;
}

/*
* TODO: Why it is not in ItrSmootherCSR.c?  Move?
* TODO: Add Doxygen!
*/
void fasp_smoother_dcsr_gs_multicolor(dvector* u, dCSRmat* A, dvector* b, INT L,
                                      const INT order)
{
#if MULTI_COLOR_ORDER
    const INT   nrow = A->row; // number of rows
    const INT * ia = A->IA, *ja = A->JA;
    const REAL *aj = A->val, *bval = b->val;
    REAL*       uval = u->val;

    INT  i, j, k, begin_row, end_row;
    REAL t, d = 0.0;

    INT  myid, mybegin, myend;
    INT  color = A->color;
    INT* IC    = A->IC;
    INT* ICMAP = A->ICMAP;
    INT  I;

    // From color to 0 order
    if (order == -1) {
        while (L--) {
            for (myid = color - 1; myid > -1; myid--) {
                mybegin = IC[myid];
                myend   = IC[myid + 1];
#ifdef _OPENMP
#pragma omp parallel for private(I, i, t, begin_row, end_row, k, j, d)
#endif
                for (I = mybegin; I < myend; I++) {
                    i         = ICMAP[I];
                    t         = bval[i];
                    begin_row = ia[i], end_row = ia[i + 1];
                    for (k = begin_row; k < end_row; k++) {
                        j = ja[k];
                        if (i != j)
                            t -= aj[k] * uval[j];
                        else
                            d = aj[k];
                    } // end for k
                    if (ABS(d) > SMALLREAL) uval[i] = t / d;
                } // end for I
            }      // end for myid
        }          // end while
    }
    // From 0 to color order
    else {
        while (L--) {
            for (myid = 0; myid < color; myid++) {
                mybegin = IC[myid];
                myend   = IC[myid + 1];
#ifdef _OPENMP
#pragma omp parallel for private(I, i, t, begin_row, end_row, k, j, d)
#endif
                for (I = mybegin; I < myend; I++) {
                    i         = ICMAP[I];
                    t         = bval[i];
                    begin_row = ia[i], end_row = ia[i + 1];
                    for (k = begin_row; k < end_row; k++) {
                        j = ja[k];
                        if (i != j)
                            t -= aj[k] * uval[j];
                        else
                            d = aj[k];
                    } // end for k
                    if (ABS(d) > SMALLREAL) uval[i] = t / d;
                } // end for I
            }      // end for myid
        }          // end while
    }              // end if order
#else
    printf("### ERROR: MULTI_COLOR_ORDER  has not been turn on!!!  \n");
#endif
    return;
}

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

9.81 BlaSparseCSRL.c File Reference

Sparse matrix operations for dCSRLmat matrices.

```
#include "fasp.h"
#include "fasp_functs.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-Crummey and John Garvin Optimizaing sparse matrix vector product computations using unroll and jam, Tech Report Rice Univ, Aug 2002.

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

num_rows	Number of rows
num_cols	Number of cols
num_nonzeros	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

A	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.

```
#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

dCSRLmat * fasp_dcsrl_create (const INT num_rows,
                              const INT num_cols,
                              const INT num_nonzeros)
{
    dCSRLmat *A   = (dCSRLmat *)fasp_mem_calloc(1, sizeof(dCSRLmat));

    A -> row      = num_rows;
    A -> col      = num_cols;
    A -> nnz      = num_nonzeros;
    A -> nz_diff  = NULL;
    A -> index    = NULL;
    A -> start    = NULL;
    A -> ja       = NULL;
    A -> val      = NULL;

    return A;
}

void fasp_dcsrl_free (dCSRLmat *A)
{
    if (A) {
        if (A -> nz_diff) free(A -> nz_diff);
        if (A -> index)   free(A -> index);
        if (A -> start)   free(A -> start);
        if (A -> ja)      free(A -> ja);
        if (A -> val)     free(A -> val);
        free(A);
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 memeory 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

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

nx	Number of grids in x direction
ny	Number of grids in y direction
nz	Number of grids in z direction
nxy	Number of grids in x-y plane
ngrid	Number of grids
nband	Number of off-diagonal bands
nc	Number of components
offsets	Shift from diagonal
A	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

A	Pointer to the dSTRmat matrix
B	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

nx	Number of grids in x direction
ny	Number of grids in y direction
nz	Number of grids in z direction
nc	Number of components
nband	Number of off-diagonal bands
offsets	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 dSTRmat 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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

dSTRmat fasp_dstr_create (const INT  nx,
                          const INT  ny,
                          const INT  nz,
                          const INT  nc,
                          const INT  nband,
                          INT       *offsets)
{
    dSTRmat A;

    INT i;

    A.nx=nx; A.ny=ny; A.nz=nz;
    A.nc=nc;
    A.nxy=A.nx*A.ny;
    A.ngrid=A.nxy*A.nz;
    A.nband=nband;

    A.offsets=(INT*)fasp_mem_calloc(nband, sizeof(INT));

    for (i=0;i<nband;++i) A.offsets[i]=offsets[i];

    A.diag=(REAL*)fasp_mem_calloc(A.ngrid*A.nc*A.nc, sizeof(REAL));

    A.offdiag=(REAL**)fasp_mem_calloc(nband, sizeof(REAL*));

    for (i=0;i<A.nband;++i) {
        A.offdiag[i]=(REAL*)fasp_mem_calloc((A.ngrid-ABS(A.offsets[i]))*A.nc*A.nc, sizeof(REAL));
    }

    return(A);
}

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)
{
    INT i;

    A->nx=nx;
    A->ny=ny;
    A->nz=nz;
    A->nxy=nxy;
    A->ngrid=ngrid;
    A->nband=nband;
    A->nc=nc;

    A->offsets=(INT*)fasp_mem_calloc(nband, sizeof(INT));

    for (i=0;i<nband;++i) A->offsets[i]=offsets[i];

    A->diag=(REAL*)fasp_mem_calloc(ngrid*nc*nc, sizeof(REAL));

    A->offdiag = (REAL **)fasp_mem_calloc(A->nband, sizeof(REAL*));

    for (i=0;i<nband;++i) {
        A->offdiag[i]=(REAL*)fasp_mem_calloc((ngrid-ABS(offsets[i]))*nc*nc, sizeof(REAL));
    }
}

void fasp_dstr_free (dSTRmat *A)
{
    INT i;

    fasp_mem_free(A->offsets); A->offsets = NULL;
    fasp_mem_free(A->diag);    A->diag    = NULL;

    for ( i = 0; i < A->nband; ++i ) {
        fasp_mem_free(A->offdiag[i]); A->offdiag[i] = NULL;
    }

    A->nx = A->ny = A->nz = A->nxy=0;
    A->ngrid = A->nband = A->nc=0;
}

void fasp_dstr_cp (const dSTRmat *A,
                   dSTRmat       *B)
{
    const INT nc2 = (A->nc)*(A->nc);

    INT i;
    B->nx    = A->nx;
    B->ny    = A->ny;
    B->nz    = A->nz;
    B->nxy   = A->nxy;
    B->ngrid = A->ngrid;
    B->nc    = A->nc;
    B->nband = A->nband;

    memcpy(B->offsets,A->offsets,(A->nband)*sizeof(INT));
    memcpy(B->diag,A->diag,(A->ngrid*nc2)*sizeof(REAL));
    for (i=0;i<A->nband;++i) {
        memcpy(B->offdiag[i],A->offdiag[i],
               ((A->ngrid - ABS(A->offsets[i]))*nc2)*sizeof(REAL));
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.85 BlaSparseUtil.c File Reference

Routines for sparse matrix operations.

```
#include <math.h>
#include <time.h>
#include "fasp.h"
#include "fasp_functs.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 nonzeroes.
- 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 nonzeroes.
- 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∗A∗P, if jac is not null. If jac is null only finds num of nonzeroes.
- void fasp_sparse_wtams_ (INT ∗jw, INT ∗ia, INT ∗ja, INT ∗nwp, INT ∗map, INT ∗jv, INT ∗nvp, INT ∗icp)

  Finds the nonzeroes 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 ∗nxp, 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∗A∗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

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

ia	array of row pointers (as usual in CSR)
ja	array of column indices
a	array of entries of teh input
na	number of rows of A
ma	number of cols of A
iat	array of row pointers in the result
jat	array of column indices
at	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

ia	array of row pointers 1st multiplicand
ja	array of column indices 1st multiplicand
a	entries of the 1st multiplicand
ib	array of row pointers 2nd multiplicand
jb	array of column indices 2nd multiplicand
b	entries of the 2nd multiplicand
ic	array of row pointers in c=a∗b
jc	array of column indices in c=a∗b
c	entries of the result: c= a∗b
nap	number of rows in the 1st multiplicand
map	number of columns in the 1st multiplicand
mbp	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 nonzeroes.

Parameters

ia	array of row pointers 1st multiplicand
ja	array of column indices 1st multiplicand
ib	array of row pointers 2nd multiplicand
jb	array of column indices 2nd multiplicand
nap	number of rows of A
map	number of cols of A
mbp	number of cols of b
ic	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)
jc	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 nonzeroes.

Parameters

ia	array of row pointers 1st summand
ja	array of column indices 1st summand
ib	array of row pointers 2nd summand
jb	array of column indices 2nd summand
nab	number of rows
mab	number of cols
ic	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)
jc	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

ia	array of row pointers 1st summand
ja	array of column indices 1st summand
a	entries of the 1st summand
ib	array of row pointers 2nd summand
jb	array of column indices 2nd summand
b	entries of the 2nd summand

Parameters

nab	number of rows
mab	number of cols
ic	array of row pointers in c=a+b
jc	array of column indices in c=a+b
c	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

ia	array of row pointers (as usual in CSR)
ja	array of column indices
na	number of rows
ma	number of cols
iat	array of row pointers in the result
jat	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

A	pointer to the matrix

Note

Only use the sparsity of A, index starts from 1 (fortran)!!

Definition at line 907 of file BlaSparseUtil.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∗A∗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

ir	:I: array of row pointers for R
jr	:I: array of column indices for R
r	:I: entries of R
ia	:I: array of row pointers for A
ja	:I: array of column indices for A
a	:I: entries of A
ipt	:I: array of row pointers for P
jpt	:I: array of column indices for P
pt	:I: entries of P
nin	:I: number of rows in R
ncin	:I: number of rows in
iac	:O: array of row pointers for P
jac	:O: array of column indices for P
ac	:O: entries of P
idummy	not changed

Note

Compute R∗A∗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∗A∗P, if jac is not null. If jac is null only finds num of nonzeroes.

Note

:I: is input :O: is output :IO: is both

Parameters

ir	:I: array of row pointers for R
jr	:I: array of column indices for R
ia	:I: array of row pointers for A
ja	:I: array of column indices for A
ip	:I: array of row pointers for P
jp	:I: array of column indices for P
nin	:I: number of rows in R
ncin	:I: number of columns in R
iac	:O: array of row pointers for Ac
jac	:O: array of column indices for Ac
maxrout	:O: the maximum nonzeroes per row for R

Note

Computes the sparsity pattern of R∗A∗P. maxrout is output and is the maximum nonzeroes per row for r. On output we also have is 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

jw	:I: indices such that w[jw] is nonzero
w	:I: the values of w
ia	:I: array of row pointers for A
ja	:I: array of column indices for A
a	:I: entries of A
nwp	:I: number of nonzeroes in w (the length of w)
map	:I: number of columns in A
jv	:O: indices such that v[jv] is nonzero
v	:O: the result v^t=w^t A
nvp	:I: number of nonzeroes 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 nonzeroes 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

jw	:I: indices such that w[jw] is nonzero
ia	:I: array of row pointers for A
ja	:I: array of column indices for A

Parameters

nwp	:I: number of nonzeroes in w (the length of w)
map	:I: number of columns in A
jv	:O: indices such that v[jv] is nonzero
nvp	:I: number of nonzeroes in v
icp	: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

:I: is input :O: is output :IO: is both

Parameters

jy	:I: indices such that y[jy] is nonzero
y	:I: is a sparse vector.
nyp	:I: number of nonzeroes in y
jx	:I: indices such that x[jx] is nonzero
x	:I: is a sparse vector.
nxp	:I: number of nonzeroes in x
icp	???
s	: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

jy	:I: indices such that y[jy] is nonzero
y	:I: is a sparse vector
nyp	:I: number of nonzeroes in v
x	: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
s	:O: s = y^t x

Definition at line 699 of file BlaSparseUtil.c.

9.86 BlaSparseUtil.c

Go to the documentation of this file.

```

#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/
void fasp_sparse_abybms_ (INT *ia,
                          INT *ja,
                          INT *ib,
                          INT *jb,
                          INT *nap,
                          INT *map,
                          INT *mbp,
                          INT *ic,
                          INT *jc)
{
    /*  FORM ic when jc is null and both when jc is not null for
the ic and jc are for c=a*b, a and b sparse  */
    /*  na = number of rows of a    */
    /*  mb = number of columns of b */
    unsigned int jcform=0;
    INT na,mb,icpp,iastrt,ibstrt,iaend,ibend,i,j,k,jia,jib;
    INT *icp;
    if (jc) jcform=1;
    na=*nap;
    mb=*mbp;
    icpp = 1;
    icp=(INT *) calloc(mb,sizeof(INT));

    for (i = 0; i < mb; ++i) icp[i] = 0;

    for (i = 0; i < na; ++i) {
        ic[i] = icpp;
        iastrt = ia[i]-1;
        iaend = ia[i+1]-1;
        if (iaend > iastrt) {
            for (jia = iastrt; jia < iaend; ++jia) {
                j = ja[jia]-1;
                ibstrt = ib[j]-1;
                ibend = ib[j+1]-1;
                if (ibend > ibstrt) {
                    for (jib = ibstrt; jib< ibend; ++jib) {
                        k = jb[jib]-1;
                        if (icp[k] != i+1) {
                            if (jcform) jc[icpp-1] = k+1;
                            ++icpp;
                            icp[k] = i+1;
                        } //if
                    } //for
                } //if
            } //for
        } //if
    } //for (i...
    ic[na] = icpp;

    if (icp) free(icp);

    return;
}

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)
{
    INT na,mb,iastrt,ibstrt,iaend,ibend,icstrt,icend,i,j,k,ji,jia,jib;
    REAL *x;
    REAL x0;
    /*
C---------------------------------------------------------------------
C...    C = A*B
C---------------------------------------------------------------------
*/
    na=*nap;
    mb=*mbp;
    x=(REAL *)calloc(mb,sizeof(REAL));
    for (i = 0; i < na; ++i) {
        icstrt = ic[i]-1;
        icend = ic[i+1]-1;
        if (icend > icstrt) {
            for (ji = icstrt;ji < icend;++ji) {
                k=jc[ji]-1;
                x[k] = 0e+0;
            }
            iastrt = ia[i]-1;
            iaend = ia[i+1]-1;
            if (iaend > iastrt) {
                for (jia = iastrt; jia < iaend ; ++jia) {
                    j = ja[jia]-1;
                    x0 = a[jia];
                    ibstrt = ib[j]-1;
                    ibend = ib[j+1]-1;
                    if (ibend > ibstrt) {
                        for (jib = ibstrt; jib < ibend; ++jib) {
                            k = jb[jib]-1;
                            x[k] += x0*b[jib];
                        }
                    }  // end if
                } //  end for
            }
            for (ji = icstrt; ji < icend; ++ji) {
                k=jc[ji]-1;
                c[ji]=x[k];
            } // end for
        } // end if
    }//end do
    if (x) free(x);
    return;
}

void fasp_sparse_iit_ (INT *ia,
                       INT *ja,
                       INT *na,
                       INT *ma,
                       INT *iat,
                       INT *jat)
{
    /*C====================================================================*/
    INT i,j,jp,n,m,mh,nh,iaa,iab,k;
    /*
C-------------------------------------------------------------------
C...    Transposition of a graph (or the matrix) symbolically.
C...
C...    Input:
C...        IA, JA   - given graph (or matrix).
C...        N        - number of rows of the matrix.
C...        M        - number of columns of the matrix.
C...
C...    Output:
C...        IAT, JAT - transposed graph (or matrix).
C...
C...    Note:
C...        N+1 is the dimension of IA.
C...        M+1 is the dimension of IAT.
C-------------------------------------------------------------------
*/
    n=*na;
    m=*ma;
    mh = m + 1;
    nh = n + 1;
    for (i = 1; i < mh; ++i) {
        iat[i] = 0;
    }
    iab = ia[nh-1] - 1;
    for (i = 1; i <= iab; ++i) {
        j = ja[i-1] + 2;
        if (j <= mh)
            iat[j-1] = iat[j-1] + 1;
    }
    iat[0] = 1;
    iat[1] = 1;
    if (m != 1) {
        for (i= 2; i< mh; ++i) {
            iat[i] = iat[i] + iat[i-1];
        }
    }
    for (i = 1; i <= n; ++i) {
        iaa = ia[i-1];
        iab = ia[i] - 1;
        if (iab >= iaa) {
            for (jp = iaa; jp <= iab; ++jp) {
                j = ja[jp-1] + 1;
                k = iat[j-1];
                jat[k-1] = i;
                iat[j-1] = k + 1;
            }
        }
    }
    return;
}

void fasp_sparse_aat_ (INT  *ia,
                       INT  *ja,
                       REAL *a,
                       INT  *na,
                       INT  *ma,
                       INT  *iat,
                       INT  *jat,
                       REAL *at)
{
    /*C=====================================================================*/
    INT i,j,jp,n,m,mh,nh,iaa,iab,k;
    /*
C-------------------------------------------------------------------
C...    Transposition of a matrix.
C...
C...    Input:
C...        IA, JA   - given graph (or matrix).
C...        N        - number of rows of the matrix.
C...        M        - number of columns of the matrix.
C...
C...    Output:
C...        IAT, JAT, AT - transposed matrix
C...
C...    Note:
C...        N+1 is the dimension of IA.
C...        M+1 is the dimension of IAT.
C-------------------------------------------------------------------
*/
```

```
    n=*na;
    m=*ma;
    mh = m + 1;
    nh = n + 1;

    for (i = 1; i < mh; ++i) {
        iat[i] = 0;
    }
    iab = ia[nh-1] - 1; /* Size of ja */
    for (i = 1;i<=iab; ++i) {
        j = ja[i-1] + 2;
        if (j <= mh) {
            iat[j-1] = iat[j-1] + 1;
        }
    }
    iat[0] = 1;
    iat[1] = 1;
    if (m != 1) {
        for (i= 2; i< mh; ++i) {
            iat[i] = iat[i] + iat[i-1];
        }
    }

    for (i=1; i<=n; ++i) {
        iaa = ia[i-1];
        iab = ia[i] - 1;
        if (iab >= iaa) {
            for (jp = iaa; jp <= iab; ++jp) {
                j = ja[jp-1] + 1;
                k = iat[j-1];
                jat[k-1] = i;
                at[k-1] = a[jp-1];
                iat[j-1] = k + 1;
            }
        }
    }

    return;
}

void fasp_sparse_aplbms_ (INT *ia,
                          INT *ja,
                          INT *ib,
                          INT *jb,
                          INT *nab,
                          INT *mab,
                          INT *ic,
                          INT *jc)
{
    unsigned int jcform=0;
    INT icpp,i1,i,j,jp,n,m,iastrt,iaend,ibstrt,ibend;
    INT *icp;
    /*
c...    addition of two general sparse matricies (symbolic part) :
c= a + b.
*/
    if (jc) jcform=1;
    n=*nab;
    m=*mab;
    icp=(INT *) calloc(m,sizeof(INT));
    for (i=0; i< m; ++i) icp[i] = 0;
    icpp = 1;
    for (i=0; i< n; ++i) {
        ic[i] = icpp;
        i1=i+1;
        iastrt = ia[i]-1;
        iaend = ia[i1]-1;
        if (iaend > iastrt) {
            for (jp = iastrt; jp < iaend; ++jp) {
                j = ja[jp];
                if (jcform) jc[icpp-1] = j;
                ++icpp;
                icp[j-1] = i1;
            }
        }
        ibstrt = ib[i] - 1;
        ibend = ib[i1] - 1;
        if (ibend > ibstrt) {
            for (jp = ibstrt; jp < ibend; ++jp) {
                j = jb[jp];
                if (icp[j-1] != i1) {
                    if (jcform) jc[icpp-1] = j;
                    ++icpp;
                }
            }
        }
    }  // // loop i=0; i< n
    ic[n] = icpp;
    if (icp) free(icp);
    return;
}

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)
{
    INT n,m,icpp,i1,i,j,iastrt,iaend,ibstrt,ibend,icstrt,icend;
    REAL *x;
    /*
c...    addition of two general sparse matricies (numerical part) :
c= a + b
*/
    n=*nab;
    m=*mab;
    x=(REAL *)calloc(m,sizeof(REAL));
    for (i=0;i<n;++i) {
        i1=i+1;
        icstrt = ic[i]-1;
        icend = ic[i1]-1;
        if (icend > icstrt) {
            for (icpp = icstrt;icpp<icend;++icpp) {
                j=jc[icpp]-1;
                x[j] = 0e+00;
            }
            iastrt = ia[i]-1;
            iaend = ia[i1]-1;
            if (iaend > iastrt) {
                for (icpp = iastrt;icpp<iaend;++icpp) {
                    j=ja[icpp]-1;
                    x[j] = a[icpp];
                }
            }
            ibstrt = ib[i]-1;
            ibend = ib[i1]-1;
            if (ibend > ibstrt) {
                for (icpp = ibstrt;icpp<ibend;++icpp) {
                    j = jb[icpp]-1;
                    x[j] = x[j] + b[icpp];
                }
            }
            for (icpp = icstrt;icpp<icend;++icpp) {
                j=jc[icpp]-1;
                c[icpp] = x[j];
            }
        } // if (icstrt > icend)...
    } // loop i=0; i< n
    if (x) free(x);
    return;
}

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)
{
    INT i,jk,jak,jpk,ic,jc,nc,icp1,ira,irb,ipa,ipb;
    INT maxri,maxr,iaa,iab,iacp,if1,jf1,jacform=0;
    INT *ix;

    nc = *ncin;
    ix=(INT *) calloc(nc,sizeof(INT));
    if (jac) jacform=1;
    maxr = 0;
    for (i =0;i<nc; ++i) {
        ix[i]=0;
        ira=ir[i];
        irb=ir[i+1];
        maxri=irb-ira;
        if (maxr < maxri) maxr=maxri;
    }
    iac[0] = 1;
    iacp = iac[0]-1;
    for (ic = 0;ic<nc;ic++) {
        ira=ir[ic]-1;
        icp1=ic+1;
        irb=ir[icp1]-1;
        for (jk = ira;jk<irb;jk++) {
            if1 = jr[jk]-1;
            iaa = ia[if1]-1;
            iab = ia[if1+1]-1;
            for (jak = iaa;jak < iab;jak++) {
                jf1 = ja[jak]-1;
                ipa = ip[jf1]-1;
                ipb = ip[jf1+1]-1;
                for (jpk = ipa;jpk < ipb;jpk++) {
                    jc = jp[jpk]-1;
                    if (ix[jc] != icp1) {
                        ix[jc]=icp1;
                        if (jacform) jac[iacp] = jc+1;
                        iacp++;
                    }
                }
            }
        }
        iac[icp1] = iacp+1;
    }
    *maxrout=maxr;
    if (ix) free(ix);
    return;
}

void fasp_sparse_wtams_ (INT *jw,
                         INT *ia,
                         INT *ja,
                         INT *nwp,
                         INT *map,
                         INT *jv,
                         INT *nvp,
                         INT *icp)
{
    INT nw,nv,iastrt,iaend,j,k,jiw,jia;
    if (*nwp<=0) {*nvp=0; return;}
    nw=*nwp;
    nv = 0;
    for (jiw = 0;jiw < nw; ++jiw) {
        j = jw[jiw]-1;
        iastrt = ia[j]-1;
        iaend  = ia[j+1]-1;
        if (iaend > iastrt) {
            for (jia = iastrt ;jia< iaend;jia++) {
                k = ja[jia]-1;
                if (!icp[k]) {
                    jv[nv] = k+1;
                    nv++;
                    icp[k] = nv;
                }
            }
        }
    }
    *nvp=nv;
    return;
}

void fasp_sparse_wta_ (INT  *jw,
                       REAL *w,
                       INT  *ia,
                       INT  *ja,
                       REAL *a,
                        INT  *nwp,
                        INT  *map,
                        INT  *jv,
                        REAL *v,
                        INT  *nvp)
{
    INT nw,nv,iastrt,iaend,j,k,ji,jiw,jia;
    REAL v0;

    if (*nwp<=0) {*nvp=-1; return;}
    nw=*nwp;
    nv=*nvp;
    for (ji = 0;ji < nv;++ji) {
        k=jv[ji]-1;
        v[k] = 0e+0;
    }
    for (jiw = 0;jiw<nw; ++jiw) {
        j = jw[jiw]-1;
        v0 = w[jiw];
        iastrt = ia[j]-1;
        iaend  = ia[j+1]-1;
        if (iaend > iastrt) {
            for (jia = iastrt;jia < iaend;jia++) {
                k = ja[jia]-1;
                v[k] += v0*a[jia];
            }
        }  // end if
    } //  end for
    return;
}

void fasp_sparse_ytxbig_ (INT  *jy,
                          REAL *y,
                          INT  *nyp,
                          REAL *x,
                          REAL *s)
{
    INT i,ii;
    *s=0e+00;
    if (*nyp > 0) {
        for (i = 0;i< *nyp; ++i) {
            ii = jy[i]-1;
            *s += y[i]*x[ii];
        }
    }
    return;
}

void fasp_sparse_ytx_ (INT  *jy,
                       REAL *y,
                       INT  *jx,
                       REAL *x,
                       INT  *nyp,
                       INT  *nxp,
                       INT  *icp,
                       REAL *s)
{// not tested
    INT i,j,i0,ii;
    *s=0e+00;
    if ((*nyp > 0) && (*nxp > 0)) {
        for (i = 0;i< *nyp; ++i) {
            j = jy[i]-1;
            i0=icp[j];
            if (i0) {
                ii=jx[i0]-1;
                *s += y[i]*x[ii];
            }
        }
    }
    return;
}

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)
{
    INT i,j,k,n,nc,nv,nw,nptjc,iacst,iacen,ic,jc,is,js,jkc,iastrt,iaend,ji,jia;
    REAL aij,v0;
    INT *icp=NULL, *jv=NULL,*jris=NULL, *jptjs=NULL;
    REAL *v=NULL, *ris=NULL, *ptjs=NULL;
    n=*nin;
    nc=*ncin;

    v  = (REAL *) calloc(n,sizeof(REAL));
    icp = (INT *) calloc(n,sizeof(INT));
    jv  = (INT *) calloc(n,sizeof(INT));
    if (!(icp && v && jv)) {
        fprintf(stderr,"### ERROR: Could not allocate memory!\n");
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }
    for (i=0;i<n;++i) {
        icp[i] = 0;
        jv[i] = 0;
        v[i]=0e+00;
    }
    for (ic = 0;ic<nc;ic++) {
        nw = ir[ic+1]-ir[ic];
        if (nw<=0) continue;
        is = ir[ic]-1;
        jris=jr+is;
        //    wtams_(jris,ia, ja, &nw,&n,jv, &nv, icp);
        //    void wtams_(INT *jw,INT *ia, INT *ja, INT *nwp,INT *map,
        //        INT *jv, INT *nvp, INT *icp)
        //    INT nw,ma,nv,iastrt,iaend,i,j,k,ji,jia;
        nv = 0;
        for (ji = 0;ji < nw; ++ji) {
            j = *(jris+ji)-1;
            iastrt = ia[j]-1;
            iaend  = ia[j+1]-1;
            if (iaend > iastrt) {
                for (jia = iastrt ;jia< iaend;jia++) {
                    k = ja[jia]-1;
                    if (!icp[k]) {
                        *(jv+nv) = k+1;
                        nv++;
                        icp[k] = nv;
                    } //end if
                } //end for
            } //end if
        } //end for loop for forming the nonz struct of (r_i)^t*A
        ris=r+is;
        //    wta_(jris, ris,ia, ja, a,&nw, &n, jv, v, &nv);
        for (ji = 0;ji < nv;++ji) {
            k=jv[ji]-1;
            v[k] = 0e+0;
        }
        for (ji = 0;ji<nw ; ++ji) {
            j = *(jris+ji)-1;
            v0 = *(ris+ji);
            iastrt = ia[j]-1;
            iaend  = ia[j+1]-1;
            if (iaend > iastrt) {
                for (jia = iastrt;jia < iaend;jia++) {
                    k = ja[jia]-1;
                    v[k] += v0*a[jia];
                }
            }  // end if
        } //end for loop for calculating the product (r_i)^t*A
        iacst=iac[ic]-1;
        iacen=iac[ic+1]-1;
        for (jkc = iacst; jkc<iacen;jkc++) {
            jc = jac[jkc]-1;
            nptjc = ipt[jc+1]-ipt[jc];
            js = ipt[jc]-1;
            jptjs = jpt+js;
            ptjs = pt+js;
            //      ytxbig_(jptjs,ptjs,&nptjc,v,&aij);
            aij=0e+00;
            if (nptjc > 0) {
                for (i = 0;i< nptjc; ++i) {
                    j = *(jptjs+i)-1;
                    aij += (*(ptjs+i))*(*(v+j));
                } //end for
            } //end if
            ac[jkc] = aij;
        } //end for
        // set nos the values of v and icp back to 0;
        for (i=0; i < nv; ++i) {
            j=jv[i]-1;
            icp[j]=0;
            v[j]=0e+00;
        } //end for
    } //end for

    if (v) free(v);
    if (icp) free(icp);
    if (jv) free(jv);

    return;
}

ivector fasp_sparse_mis (dCSRmat *A)
{
    // information of A
    INT n = A->row;
    INT *IA = A->IA;
    INT *JA = A->JA;

    // local variables
    INT i,j;
    INT row_begin, row_end;
    INT count=0;
    INT *flag;
    flag = (INT *)fasp_mem_calloc(n, sizeof(INT));
    //for (i=0;i<n;i++) flag[i]=0;
    memset(flag, 0, sizeof(INT)*n);

    // work space
    INT *work = (INT*)fasp_mem_calloc(n,sizeof(INT));

    // return vector
    ivector MIS;

    // main loop
    for (i=0;i<n;i++) {
        if (flag[i] == 0) {
            flag[i] = 1;
            row_begin = IA[i] - 1; row_end = IA[i+1] - 1;
            for (j = row_begin; j<row_end; j++) {
                if (flag[JA[j]-1] > 0) {
                    flag[i] = -1;
                    break;
                }
            }
            if (flag[i]) {
                work[count] = i; count++;
                for (j = row_begin; j<row_end; j++) {
                    flag[JA[j]-1] = -1;
                }
            }
        } // end if
    }// end for

    // form MIS
    MIS.row = count;
    work = (INT *)fasp_mem_realloc(work, count*sizeof(INT));
    MIS.val = work;

    // clean
    fasp_mem_free(flag); flag = NULL;

    //return
    return MIS;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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_aAxpy (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_aAxpy (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

Definition in file BlaSpmvBLC.c.

9.87.2 Function Documentation

9.87.2.1 fasp_blas_dblc_aAxpy()

```
void fasp_blas_dblc_aAxpy (
            const REAL alpha,
            const dBLCmat * A,
            const REAL * x,
            REAL * y )
```

Matrix-vector multiplication y = alpha∗A∗x + y.

Parameters

alpha	REAL factor a
A	Pointer to dBLCmat matrix A
x	Pointer to array x
y	Pointer to array y

Author

Xiaozhe Hu

Date

06/04/2010

Definition at line 38 of file BlaSpmvBLC.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

A	Pointer to dBLCmat matrix A
x	Pointer to array x
y	Pointer to array y

Author

Chensong Zhang

Date

04/27/2013

Definition at line 164 of file BlaSpmvBLC.c.

9.87.2.3 fasp_blas_ldblc_aAxpy()

```
void fasp_blas_ldblc_aAxpy (
            const REAL alpha,
            const dBLCmat * A,
            const LONGREAL * x,
            REAL * y )
```

Matrix-vector multiplication y = alpha∗A∗x + y.

Parameters

alpha	REAL factor a
A	Pointer to dBLCmat matrix A
x	Pointer to array x
y	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.

```
#include <time.h>

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_dblc_aAxpy (const REAL      alpha,
                           const dBLCmat  *A,
                           const REAL     *x,
                           REAL           *y)
{
    // information of A
    const INT brow = A->brow;

    // local variables
    register dCSRmat *A11, *A12, *A21, *A22;
    register dCSRmat *A13, *A23, *A31, *A32, *A33;

    INT row1, col1;
    INT row2, col2;

    const register REAL *x1, *x2, *x3;
    register REAL       *y1, *y2, *y3;

    INT i,j;
    INT start_row, start_col;

    switch (brow) {

        case 2:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A21 = A->blocks[2];
            A22 = A->blocks[3];

            row1 = A11->row;
            col1 = A11->col;

            x1 = x;
            x2 = &(x[col1]);
            y1 = y;
            y2 = &(y[row1]);

            // y1 = alpha*A11*x1 + alpha*A12*x2 + y1
            if (A11) fasp_blas_dcsr_aAxpy(alpha, A11, x1, y1);
            if (A12) fasp_blas_dcsr_aAxpy(alpha, A12, x2, y1);

            // y2 = alpha*A21*x1 + alpha*A22*x2 + y2
            if (A21) fasp_blas_dcsr_aAxpy(alpha, A21, x1, y2);
            if (A22) fasp_blas_dcsr_aAxpy(alpha, A22, x2, y2);

            break;

        case 3:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A13 = A->blocks[2];
            A21 = A->blocks[3];
            A22 = A->blocks[4];
            A23 = A->blocks[5];
            A31 = A->blocks[6];
            A32 = A->blocks[7];
            A33 = A->blocks[8];

            row1 = A11->row;
            col1 = A11->col;
            row2 = A22->row;
            col2 = A22->col;

            x1 = x;
            x2 = &(x[col1]);
            x3 = &(x[col1+col2]);
            y1 = y;
            y2 = &(y[row1]);
            y3 = &(y[row1+row2]);

            // y1 = alpha*A11*x1 + alpha*A12*x2 + alpha*A13*x3 + y1
            if (A11) fasp_blas_dcsr_aAxpy(alpha, A11, x1, y1);
            if (A12) fasp_blas_dcsr_aAxpy(alpha, A12, x2, y1);
            if (A13) fasp_blas_dcsr_aAxpy(alpha, A13, x3, y1);

            // y2 = alpha*A21*x1 + alpha*A22*x2 + alpha*A23*x3 + y2
            if (A21) fasp_blas_dcsr_aAxpy(alpha, A21, x1, y2);
            if (A22) fasp_blas_dcsr_aAxpy(alpha, A22, x2, y2);
            if (A23) fasp_blas_dcsr_aAxpy(alpha, A23, x3, y2);

            // y3 = alpha*A31*x1 + alpha*A32*x2 + alpha*A33*x3 + y2
            if (A31) fasp_blas_dcsr_aAxpy(alpha, A31, x1, y3);
            if (A32) fasp_blas_dcsr_aAxpy(alpha, A32, x2, y3);
            if (A33) fasp_blas_dcsr_aAxpy(alpha, A33, x3, y3);

            break;

        default:

            start_row = 0;
            start_col = 0;

            for (i=0; i<brow; i++) {

                for (j=0; j<brow; j++) {

                    if (A->blocks[i*brow+j]) {
                        fasp_blas_dcsr_aAxpy(alpha, A->blocks[i*brow+j],
                                             &(x[start_col]), &(y[start_row]));
                    }
                    start_col = start_col + A->blocks[j*brow+j]->col;

                }

                start_row = start_row + A->blocks[i*brow+i]->row;
                start_col = 0;
            }

            break;

    } // end of switch

}

void fasp_blas_dblc_mxv (const dBLCmat  *A,
                         const REAL     *x,
                         REAL           *y)
{
    // information of A
    const INT brow = A->brow;

    // local variables
    register dCSRmat *A11, *A12, *A21, *A22;
    register dCSRmat *A13, *A23, *A31, *A32, *A33;

    INT row1, col1;
    INT row2, col2;

    const register REAL *x1, *x2, *x3;
    register REAL       *y1, *y2, *y3;

    INT i,j;
    INT start_row, start_col;

    switch (brow) {

        case 2:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A21 = A->blocks[2];
            A22 = A->blocks[3];

            row1 = A11->row;
            col1 = A11->col;

            x1 = x;
            x2 = &(x[col1]);
            y1 = y;
            y2 = &(y[row1]);

            // y1 = A11*x1 + A12*x2
            if (A11) fasp_blas_dcsr_mxv(A11, x1, y1);
            if (A12) fasp_blas_dcsr_aAxpy(1.0, A12, x2, y1);

            // y2 = A21*x1 + A22*x2
            if (A21) fasp_blas_dcsr_mxv(A21, x1, y2);
            if (A22) fasp_blas_dcsr_aAxpy(1.0, A22, x2, y2);

            break;

        case 3:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A13 = A->blocks[2];
            A21 = A->blocks[3];
            A22 = A->blocks[4];
            A23 = A->blocks[5];
            A31 = A->blocks[6];
            A32 = A->blocks[7];
            A33 = A->blocks[8];

            row1 = A11->row;
            col1 = A11->col;
            row2 = A22->row;
            col2 = A22->col;

            x1 = x;
            x2 = &(x[col1]);
            x3 = &(x[col1+col2]);
            y1 = y;
            y2 = &(y[row1]);
            y3 = &(y[row1+row2]);

            // y1 = A11*x1 + A12*x2 + A13*x3 + y1
            if (A11) fasp_blas_dcsr_mxv(A11, x1, y1);
            if (A12) fasp_blas_dcsr_aAxpy(1.0, A12, x2, y1);
            if (A13) fasp_blas_dcsr_aAxpy(1.0, A13, x3, y1);

            // y2 = A21*x1 + A22*x2 + A23*x3 + y2
            if (A21) fasp_blas_dcsr_mxv(A21, x1, y2);
            if (A22) fasp_blas_dcsr_aAxpy(1.0, A22, x2, y2);
            if (A23) fasp_blas_dcsr_aAxpy(1.0, A23, x3, y2);

            // y3 = A31*x1 + A32*x2 + A33*x3 + y2
            if (A31) fasp_blas_dcsr_mxv(A31, x1, y3);
            if (A32) fasp_blas_dcsr_aAxpy(1.0, A32, x2, y3);
            if (A33) fasp_blas_dcsr_aAxpy(1.0, A33, x3, y3);

            break;

        default:

            start_row = 0;
            start_col = 0;

            for (i=0; i<brow; i++) {

                for (j=0; j<brow; j++){

                    if (j==0) {
                        if (A->blocks[i*brow+j]){
                            fasp_blas_dcsr_mxv(A->blocks[i*brow+j], &(x[start_col]), &(y[start_row]));
                        }
                    }
                    else {
                        if (A->blocks[i*brow+j]){
                            fasp_blas_dcsr_aAxpy(1.0, A->blocks[i*brow+j], &(x[start_col]), &(y[start_row]));
                        }
                    }
                    start_col = start_col + A->blocks[j*brow+j]->col;
                }

                start_row = start_row + A->blocks[i*brow+i]->row;
                start_col = 0;
            }

            break;

    } // end of switch

}

void fasp_blas_ldblc_aAxpy (const REAL      alpha,
                           const dBLCmat    *A,
                           const LONGREAL   *x,
                           REAL             *y)
{
   // information of A
    const INT brow = A->brow;

    // local variables
    register dCSRmat *A11, *A12, *A21, *A22;
    register dCSRmat *A13, *A23, *A31, *A32, *A33;

    INT row1, col1;
    INT row2, col2;

    const register LONGREAL *x1, *x2, *x3;
    register REAL       *y1, *y2, *y3;

    INT i,j;
    INT start_row, start_col;

    switch (brow) {

        case 2:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A21 = A->blocks[2];
            A22 = A->blocks[3];

            row1 = A11->row;
            col1 = A11->col;

            x1 = x;
            x2 = &(x[col1]);
            y1 = y;
            y2 = &(y[row1]);

            // y1 = alpha*A11*x1 + alpha*A12*x2 + y1
            if (A11) fasp_blas_ldcsr_aAxpy(alpha, A11, x1, y1);
            if (A12) fasp_blas_ldcsr_aAxpy(alpha, A12, x2, y1);

            // y2 = alpha*A21*x1 + alpha*A22*x2 + y2
            if (A21) fasp_blas_ldcsr_aAxpy(alpha, A21, x1, y2);
            if (A22) fasp_blas_ldcsr_aAxpy(alpha, A22, x2, y2);

            break;

        case 3:
            A11 = A->blocks[0];
            A12 = A->blocks[1];
            A13 = A->blocks[2];
            A21 = A->blocks[3];
            A22 = A->blocks[4];
            A23 = A->blocks[5];
            A31 = A->blocks[6];
            A32 = A->blocks[7];
            A33 = A->blocks[8];

            row1 = A11->row;
            col1 = A11->col;
            row2 = A22->row;
            col2 = A22->col;
            x1 = x;
            x2 = &(x[col1]);
            x3 = &(x[col1+col2]);
            y1 = y;
            y2 = &(y[row1]);
            y3 = &(y[row1+row2]);

            // y1 = alpha*A11*x1 + alpha*A12*x2 + alpha*A13*x3 + y1
            if (A11) fasp_blas_ldcsr_aAxpy(alpha, A11, x1, y1);
            if (A12) fasp_blas_ldcsr_aAxpy(alpha, A12, x2, y1);
            if (A13) fasp_blas_ldcsr_aAxpy(alpha, A13, x3, y1);

            // y2 = alpha*A21*x1 + alpha*A22*x2 + alpha*A23*x3 + y2
            if (A21) fasp_blas_ldcsr_aAxpy(alpha, A21, x1, y2);
            if (A22) fasp_blas_ldcsr_aAxpy(alpha, A22, x2, y2);
            if (A23) fasp_blas_ldcsr_aAxpy(alpha, A23, x3, y2);

            // y3 = alpha*A31*x1 + alpha*A32*x2 + alpha*A33*x3 + y2
            if (A31) fasp_blas_ldcsr_aAxpy(alpha, A31, x1, y3);
            if (A32) fasp_blas_ldcsr_aAxpy(alpha, A32, x2, y3);
            if (A33) fasp_blas_ldcsr_aAxpy(alpha, A33, x3, y3);

            break;

        default:

            start_row = 0;
            start_col = 0;

            for (i=0; i<brow; i++) {

                for (j=0; j<brow; j++) {

                    if (A->blocks[i*brow+j]) {
                        fasp_blas_ldcsr_aAxpy(alpha, A->blocks[i*brow+j],
                                              &(x[start_col]), &(y[start_row]));
                    }
                    start_col = start_col + A->blocks[j*brow+j]->col;

                }

                start_row = start_row + A->blocks[i*brow+i]->row;
                start_col = 0;
            }

            break;

    } // end of switch
}

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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_aAxpy (const REAL alpha, const dBSRmat ∗A, const REAL ∗x, REAL ∗y)

Compute y := alpha∗A∗x + y.

- void fasp_blas_dbsr_aAxpy_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

Definition in file BlaSpmvBSR.c.

9.89.2 Function Documentation

9.89.2.1 fasp_blas_dbsr_aAxpby()

```
void fasp_blas_dbsr_aAxpby (
            const REAL alpha,
            dBSRmat * A,
            REAL * x,
            const REAL beta,
            REAL * y )
```

Compute y := alpha∗A∗x + beta∗y.

Parameters

alpha	REAL factor alpha
A	Pointer to the dBSRmat matrix
x	Pointer to the array x
beta	REAL factor beta
y	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_aAxpy()

```
void fasp_blas_dbsr_aAxpy (
             const REAL alpha,
             const dBSRmat * A,
             const REAL * x,
             REAL * y )
```

Compute y := alpha∗A∗x + y.

Parameters

alpha	REAL factor alpha
A	Pointer to the dBSRmat matrix
x	Pointer to the array x
y	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_aAxpy_agg()

```
void fasp_blas_dbsr_aAxpy_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

alpha	REAL factor alpha
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 624 of file BlaSpmvBSR.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

A	Pointer to dBSRmat matrix A
alpha	REAL factor alpha

Author

Xiaozhe Hu

Date

05/26/2014

Definition at line 38 of file BlaSpmvBSR.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

A	Pointer to the dBSRmat matrix A
B	Pointer to the dBSRmat matrix B
C	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_mxv()

```
void fasp_blas_dbsr_mxv (
             const dBSRmat * A,
             const REAL * x,
             REAL * y )
```

Compute y := A∗x.

Parameters

A	Pointer to the dBSRmat matrix
x	Pointer to the array x
y	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_mxv_agg()

```
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.

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

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

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

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

Date

10/24/2012

Definition at line 5227 of file BlaSpmvBSR.c.

9.90 BlaSpmvBSR.c

Go to the documentation of this file.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_dbsr_axm (dBSRmat     *A,
                         const REAL   alpha)
{
    const INT nnz = A->NNZ;
    const INT nb  = A->nb;

    // A direct calculation can be written as:
    fasp_blas_darray_ax(nnz*nb*nb, alpha, A->val);
}

void fasp_blas_dbsr_aAxpby (const REAL   alpha,
                            dBSRmat     *A,
                            REAL        *x,
                            const REAL   beta,
                            REAL        *y )
{
    /* members of A */
    INT  ROW  = A->ROW;
    INT  nb   = A->nb;
    INT  *IA  = A->IA;
    INT  *JA  = A->JA;
    REAL *val = A->val;

    /* local variables */
    INT     size = ROW*nb;
    INT     jump = nb*nb;
    INT     i,j,k,iend;
    REAL    temp;
    REAL   *pA  = NULL;
    REAL   *px0 = NULL;
    REAL   *py0 = NULL;
    REAL   *py  = NULL;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    //----------------------------------------------
    //   Treat (alpha == 0.0) computation
    //----------------------------------------------

    if (alpha == 0.0) {
        fasp_blas_darray_ax(size, beta, y);
        return;
    }

    //-------------------------------------------------
    //   y = (beta/alpha)*y
    //-------------------------------------------------

    temp = beta / alpha;
    if (temp != 1.0) {
        if (temp == 0.0) {
            memset(y, 0X0, size*sizeof(REAL));
        }
        else {
            //for (i = size; i--; ) y[i] *= temp; // modified by Xiaozhe, 03/11/2011
            fasp_blas_darray_ax(size, temp, y);
        }
    }

    //-----------------------------------------------------------------
    //   y += A*x (Core Computation)
    //   each non-zero block elements are stored in row-major order
    //-----------------------------------------------------------------

    switch (nb)
    {
        case 2:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*2];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*4; // &val[k*jump];
                            px0 = x+j*2; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc2( pA, px0, py );
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*2];
                    iend = IA[i+1];
                    for (k = IA[i]; k < iend; ++k) {
                        j = JA[k];
                        pA = val+k*4; // &val[k*jump];
                        px0 = x+j*2; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc2( pA, px0, py );
                    }
                }
            }
        }
            break;

        case 3:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*3];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*3];
                    iend = IA[i+1];
                    for (k = IA[i]; k < iend; ++k) {
                        j = JA[k];
                        pA = val+k*9; // &val[k*jump];
                        px0 = x+j*3; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc3( pA, px0, py );
                    }
                }
            }
        }
            break;

        case 5:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*5];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*5];
                    iend = IA[i+1];
                    for (k = IA[i]; k < iend; ++k) {
                        j = JA[k];
                        pA = val+k*25; // &val[k*jump];
                        px0 = x+j*5; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc5( pA, px0, py );
                    }
                }
            }
        }
            break;

        case 7:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*7];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb]��
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*7];
                    iend = IA[i+1];
                    for (k = IA[i]; k < iend; ++k) {
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
```

```
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );
                        }
                    }
                }
            }
                break;

            default:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
#endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*nb];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*nb];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );
                        }
                    }
                }
            }
                break;
        }

        //-----------------------------------------
        //   y = alpha*y
        //-----------------------------------------

        if (alpha != 1.0) {
            fasp_blas_darray_ax(size, alpha, y);
        }
}

void fasp_blas_dbsr_aAxpy (const REAL      alpha,
                           const dBSRmat  *A,
                           const REAL     *x,
                           REAL           *y)
{
    /* members of A */
    const INT    ROW = A->ROW;
    const INT    nb  = A->nb;
    const INT   *IA  = A->IA;
    const INT   *JA  = A->JA;
    const REAL  *val = A->val;

    /* local variables */
    const REAL *pA   = NULL;
    const REAL *px0  = NULL;
    REAL *py0        = NULL;
    REAL *py         = NULL;

    REAL  temp = 0.0;
    INT   size = ROW*nb;
    INT   jump = nb*nb;
    INT   i, j, k, iend;

    SHORT nthreads = 1, use_openmp = FALSE;

```

```
#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    //----------------------------------------------
    //   Treat (alpha == 0.0) computation
    //----------------------------------------------

    if (alpha == 0.0){
        return; // Nothing to compute
    }

    //-------------------------------------------------
    //   y = (1.0/alpha)*y
    //-------------------------------------------------

    if (alpha != 1.0){
        temp = 1.0 / alpha;
        fasp_blas_darray_ax(size, temp, y);
    }

    //-----------------------------------------------------------------
    //   y += A*x (Core Computation)
    //   each non-zero block elements are stored in row-major order
    //-----------------------------------------------------------------

    switch (nb)
    {
        case 2:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*2];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*4; // &val[k*jump];
                            px0 = x+j*2; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc2( pA, px0, py );
                        }
                    }
                }
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*2];
                    iend = IA[i+1];
                    for (k = IA[i]; k < iend; ++k) {
                        j = JA[k];
                        pA = val+k*4; // &val[k*jump];
                        px0 = x+j*2; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc2( pA, px0, py );
                    }
                }
            }
        }
            break;

        case 3:
        {
            if (use_openmp) {
                INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                for (myid =0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*3];
                        iend = IA[i+1];
```

```
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                pA = val+k*9; // &val[k*jump];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i){
                        py0 = &y[i*3];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );
                        }
                    }
                }
            }
                break;

            case 5:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*5];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i){
                        py0 = &y[i*5];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );
                        }
                    }
                }
            }
                break;

            case 7:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend )
#endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*7];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*7];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );
                        }

                    }
                }
            }
                break;

            default:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, pA, px0, py,iend)
#endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*nb];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );
                            }

                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*nb];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );
                        }

                    }
                }
            }
                break;
        }

        //-----------------------------------------
        //   y = alpha*y
        //-----------------------------------------

        if (alpha != 1.0){
            fasp_blas_darray_ax(size, alpha, y);
        }
        return;
}

void fasp_blas_dbsr_aAxpy_agg (const REAL      alpha,
                               const dBSRmat  *A,
                               const REAL     *x,
                               REAL           *y)
{
        /* members of A */
        const INT    ROW = A->ROW;
        const INT    nb  = A->nb;
        const INT  *IA  = A->IA;
        const INT  *JA  = A->JA;

        /* local variables */
        const REAL *px0 = NULL;
        REAL       *py0 = NULL, *py = NULL;
        SHORT       nthreads = 1, use_openmp = FALSE;

        INT         size = ROW*nb;
        INT         i, j, k, iend;
        REAL        temp = 0.0;

#ifdef _OPENMP
        if ( ROW > OPENMP_HOLDS ) {
            use_openmp = TRUE;
            nthreads = fasp_get_num_threads();
        }
#endif

        //-----------------------------------------------
        //   Treat (alpha == 0.0) computation
        //-----------------------------------------------

        if (alpha == 0.0){
            return; // Nothing to compute
        }

        //-------------------------------------------------
        //   y = (1.0/alpha)*y
        //-------------------------------------------------

        if (alpha != 1.0){
            temp = 1.0 / alpha;
            fasp_blas_darray_ax(size, temp, y);
        }

        //-----------------------------------------------------------------
        //   y += A*x (Core Computation)
        //   each non-zero block elements are stored in row-major order
        //-----------------------------------------------------------------

        switch (nb)
        {
            case 2:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, px0, py,iend)
#endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*2];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                px0 = x+j*2; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*2];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            px0 = x+j*2; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                        }
                    }
                }
            }
                break;

            case 3:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
    #ifdef _OPENMP
    #pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, px0, py,iend )
    #endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*3];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i){
                        py0 = &y[i*3];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                        }
                    }
                }
            }
                break;

            case 5:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
    #ifdef _OPENMP
    #pragma omp parallel for  private(myid, mybegin, myend, i, py0, k, j, px0, py,iend )
    #endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*5];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];
                                py[3] += px0[3];
                                py[4] += px0[4];
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i){
                        py0 = &y[i*5];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                        }
                    }
                }
            }
                break;

            case 7:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
    #ifdef _OPENMP
    #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py,iend )
    #endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*7];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];
                                py[3] += px0[3];
                                py[4] += px0[4];
                                py[5] += px0[5];
                                py[6] += px0[6];
                            }
                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*7];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];
                        }

                    }
                }
            }
                break;

            default:
            {
                if (use_openmp) {
                    INT myid, mybegin, myend;
    #ifdef _OPENMP
    #pragma omp parallel for private(myid, mybegin, myend, i, py0, k, j, px0, py,iend)
    #endif
                    for (myid =0; myid < nthreads; myid++) {
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i) {
                            py0 = &y[i*nb];
                            iend = IA[i+1];
                            for (k = IA[i]; k < iend; ++k) {
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy(nb, 1.0, px0, py);
                            }

                        }
                    }
                }
                else {
                    for (i = 0; i < ROW; ++i) {
                        py0 = &y[i*nb];
                        iend = IA[i+1];
                        for (k = IA[i]; k < iend; ++k) {
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
```

```
                            py = py0;
                            fasp_blas_darray_axpy(nb, 1.0, px0, py);
                        }

                    }
                }
            }
                break;
    }

    //-------------------------------------------
    //   y = alpha*y
    //-------------------------------------------

    if ( alpha != 1.0 ) fasp_blas_darray_ax(size, alpha, y);

    return;
}

void fasp_blas_dbsr_mxv (const dBSRmat  *A,
                         const REAL     *x,
                         REAL           *y)
{
    /* members of A */
    const INT   ROW = A->ROW;
    const INT   nb  = A->nb;
    const INT  *IA  = A->IA;
    const INT  *JA  = A->JA;
    const REAL *val = A->val;

    /* local variables */
    INT     size = ROW*nb;
    INT     jump = nb*nb;
    INT     i,j,k, num_nnz_row;

    const REAL *pA  = NULL;
    const REAL *px0 = NULL;
    REAL       *py0 = NULL;
    REAL        *py  = NULL;

    SHORT use_openmp = FALSE;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    //----------------------------------------------------------------
    //  zero out 'y'
    //----------------------------------------------------------------
    fasp_darray_set(size, y, 0.0);

    //----------------------------------------------------------------
    //   y = A*x (Core Computation)
    //   each non-zero block elements are stored in row-major order
    //----------------------------------------------------------------

    switch (nb)
    {
        case 3:
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i)
                    {
                        py0 = &y[i*3];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row)
                        {
                            case 3:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 4:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
```

```
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    break;

                case 6:
                    k = IA[i];
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    break;

                case 7:
                    k = IA[i];
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*9;
                    px0 = x+j*3;
                    py = py0;
                    fasp_blas_smat_ypAx_nc3( pA, px0, py );

```

```
                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*9;
                                    px0 = x+j*3;
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc3( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i)
                {
                    py0 = &y[i*3];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row)
                    {
                        case 3:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            break;

                        case 6:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            break;

                        case 7:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*9; // &val[k*jump];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc3( pA, px0, py );

                            break;

                        default:
                            for (k = IA[i]; k < IA[i+1]; ++k)
                            {
                                j = JA[k];
                                pA = val+k*9; // &val[k*jump];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );
```

```
                            }
                            break;
                        }
                    }
                }
            }
                break;

            case 5:
            {
                if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                    {
                        myid = omp_get_thread_num();
                        fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                        for (i=mybegin; i < myend; ++i)
                        {
                            py0 = &y[i*5];
                            num_nnz_row = IA[i+1] - IA[i];
                            switch(num_nnz_row)
                            {
                                case 3:
                                    k = IA[i];
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    break;

                                case 4:
                                    k = IA[i];
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    break;

                                case 5:
                                    k = IA[i];
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    break;

                                case 6:
                                    k = IA[i];
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    k ++;
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                    break;

                                case 7:
                                    k = IA[i];
```

```
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i)
                {
                    py0 = &y[i*5];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row)
                    {
                        case 3:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*25; // &val[k*jump];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc5( pA, px0, py );
```

```
                    break;

                case 6:
                    k = IA[i];
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    break;

                case 7:
                    k = IA[i];
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    fasp_blas_smat_ypAx_nc5( pA, px0, py );

                    k ++;
                    j = JA[k];
                    pA = val+k*25; // &val[k*jump];
                    px0 = x+j*5; // &x[j*nb];
```

```
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
            }
                break;

            case 7:
            {
                if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i)
                    {
                        py0 = &y[i*7];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row)
                        {
                            case 3:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                break;

                            case 4:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                break;

                            case 6:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
```

```
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                break;

                            case 7:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*49; // &val[k*jump];
                                    px0 = x+j*7; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc7( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i)
                {
                    py0 = &y[i*7];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row)
                    {
                        case 3:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            break;

                        case 6:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            break;

                        case 7:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
```

```
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            k ++;
                            j = JA[k];
                            pA = val+k*49; // &val[k*jump];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx_nc7( pA, px0, py );

                            break;

                        default:
                            for (k = IA[i]; k < IA[i+1]; ++k)
                            {
                                j = JA[k];
                                pA = val+k*49; // &val[k*jump];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc7( pA, px0, py );
                            }
                            break;
                    }
                }
            }
        }
            break;

        default:
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i)
                    {
                        py0 = &y[i*nb];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row)
                        {
                            case 3:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
```

```
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        break;

                                    case 4:
                                        k = IA[i];
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        break;

                                    case 5:
                                        k = IA[i];
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        break;

                                    case 6:
                                        k = IA[i];
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        break;

                                    case 7:
                                        k = IA[i];
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                                        k ++;
                                        j = JA[k];
                                        pA = val+k*jump; // &val[k*jump];
                                        px0 = x+j*nb; // &x[j*nb];
                                        py = py0;
                                        fasp_blas_smat_ypAx( pA, px0, py, nb );

```

```
                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*jump; // &val[k*jump];
                                    px0 = x+j*nb; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx( pA, px0, py, nb );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i)
                {
                    py0 = &y[i*nb];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row)
                    {
                        case 3:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );

                            k ++;
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );

                            k ++;
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );

                            k ++;
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
```

```
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                break;

                            case 6:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
                                pA = val+k*jump; // &val[k*jump];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx( pA, px0, py, nb );

                                k ++;
                                j = JA[k];
```

```
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        break;

                    case 7:
                        k = IA[i];
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        k ++;
                        j = JA[k];
                        pA = val+k*jump; // &val[k*jump];
                        px0 = x+j*nb; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx( pA, px0, py, nb );

                        break;

                    default:
                        for (k = IA[i]; k < IA[i+1]; ++k)
                        {
                            j = JA[k];
                            pA = val+k*jump; // &val[k*jump];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_smat_ypAx( pA, px0, py, nb );
                            }
                            break;
                        }
                    }
                }
            }
                break;
        }
}

void fasp_blas_dbsr_mxv_agg (const dBSRmat  *A,
                             const REAL     *x,
                             REAL           *y)
{
    /* members of A */
    const INT  ROW  = A->ROW;
    const INT  nb   = A->nb;
    const INT  size = ROW*nb;
    const INT *IA   = A->IA;
    const INT *JA   = A->JA;

    /* local variables */
    const REAL  *px0 = NULL;
    REAL        *py0 = NULL, *py = NULL;
    INT          i,j,k, num_nnz_row;
    SHORT        use_openmp = FALSE;

#ifdef _OPENMP
    const REAL  *val = A->val;
    const REAL  *pA;
    INT myid, mybegin, myend, nthreads;
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    //----------------------------------------------------------------
    //  zero out 'y'
    //----------------------------------------------------------------
    fasp_darray_set(size, y, 0.0);

    //----------------------------------------------------------------
    //   y = A*x (Core Computation)
    //   each non-zero block elements are stored in row-major order
    //----------------------------------------------------------------

    switch (nb)
    {
        case 3:
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*3];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row) {
                            case 3:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 4:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 6:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

```

```
                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            case 7:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*9;
                                px0 = x+j*3;
                                py = py0;
                                fasp_blas_smat_ypAx_nc3( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*9;
                                    px0 = x+j*3;
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc3( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*3];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row) {
                        case 3:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            break;

                        case 6:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];

                            k ++;
                            j = JA[k];
                            px0 = x+j*3; // &x[j*nb];
                            py = py0;
```

```
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                break;

                            case 7:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                k ++;
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];

                                break;

```

```
                        default:
                            for (k = IA[i]; k < IA[i+1]; ++k) {
                                j = JA[k];
                                px0 = x+j*3; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];
                            }
                            break;
                    }
                }
            }
        }
            break;

        case 5:
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*5];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row) {

                            case 3:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            case 4:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            case 6:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            case 7:
                                k = IA[i];
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                k ++;
                                j = JA[k];
                                pA = val+k*25; // &val[k*jump];
                                px0 = x+j*5; // &x[j*nb];
                                py = py0;
                                fasp_blas_smat_ypAx_nc5( pA, px0, py );

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k)
                                {
                                    j = JA[k];
                                    pA = val+k*25; // &val[k*jump];
                                    px0 = x+j*5; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc5( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*5];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row) {

                        case 3:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
```

```
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    break;

                case 6:
                    k = IA[i];
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
                    py[1] += px0[1];
                    py[2] += px0[2];
                    py[3] += px0[3];
                    py[4] += px0[4];

                    k ++;
                    j = JA[k];
                    px0 = x+j*5; // &x[j*nb];
                    py = py0;
                    py[0] += px0[0];
```

```
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            break;

                        case 7:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];

                            k ++;
                            j = JA[k];
                            px0 = x+j*5; // &x[j*nb];
```

```
                                  py = py0;
                                  py[0] += px0[0];
                                  py[1] += px0[1];
                                  py[2] += px0[2];
                                  py[3] += px0[3];
                                  py[4] += px0[4];

                                  k ++;
                                  j = JA[k];
                                  px0 = x+j*5; // &x[j*nb];
                                  py = py0;
                                  py[0] += px0[0];
                                  py[1] += px0[1];
                                  py[2] += px0[2];
                                  py[3] += px0[3];
                                  py[4] += px0[4];

                                  break;

                              default:
                                  for (k = IA[i]; k < IA[i+1]; ++k) {
                                      j = JA[k];
                                      px0 = x+j*5; // &x[j*nb];
                                      py = py0;
                                      py[0] += px0[0];
                                      py[1] += px0[1];
                                      py[2] += px0[2];
                                      py[3] += px0[3];
                                      py[4] += px0[4];
                                  }
                                  break;
                          }
                      }
                  }
              }
                  break;

              case 7:
              {
                  if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                      {
                          myid = omp_get_thread_num();
                          fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                          for (i=mybegin; i < myend; ++i) {
                              py0 = &y[i*7];
                              num_nnz_row = IA[i+1] - IA[i];
                              switch(num_nnz_row) {

                                  case 3:
                                      k = IA[i];
                                      j = JA[k];
                                      pA = val+k*49; // &val[k*jump];
                                      px0 = x+j*7; // &x[j*nb];
                                      py = py0;
                                      fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                      k ++;
                                      j = JA[k];
                                      pA = val+k*49; // &val[k*jump];
                                      px0 = x+j*7; // &x[j*nb];
                                      py = py0;
                                      fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                      k ++;
                                      j = JA[k];
                                      pA = val+k*49; // &val[k*jump];
                                      px0 = x+j*7; // &x[j*nb];
                                      py = py0;
                                      fasp_blas_smat_ypAx_nc7( pA, px0, py );

                                      break;

                                  case 4:
                                      k = IA[i];
                                      j = JA[k];
                                      pA = val+k*49; // &val[k*jump];
                                      px0 = x+j*7; // &x[j*nb];
                                      py = py0;
                                      fasp_blas_smat_ypAx_nc7( pA, px0, py );
```

```
                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        break;

                    case 5:
                        k = IA[i];
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        break;

                    case 6:
                        k = IA[i];
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        break;

                    case 7:
                        k = IA[i];
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        k ++;
                        j = JA[k];
                        pA = val+k*49; // &val[k*jump];
                        px0 = x+j*7; // &x[j*nb];
                        py = py0;
                        fasp_blas_smat_ypAx_nc7( pA, px0, py );

                        break;

                    default:
                        for (k = IA[i]; k < IA[i+1]; ++k) {
                            j = JA[k];
```

```
                                    pA = val+k*49; // &val[k*jump];
                                    px0 = x+j*7; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_smat_ypAx_nc7( pA, px0, py );
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*7];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row) {

                        case 3:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            break;

                        case 6:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

```

```
                            break;

                        case 7:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            k ++;
                            j = JA[k];
                            px0 = x+j*7; // &x[j*nb];
                            py = py0;
                            py[0] += px0[0];
                            py[1] += px0[1];
                            py[2] += px0[2];
                            py[3] += px0[3];
                            py[4] += px0[4];
                            py[5] += px0[5];
                            py[6] += px0[6];

                            break;

                        default:
                            for (k = IA[i]; k < IA[i+1]; ++k) {
                                j = JA[k];
                                px0 = x+j*7; // &x[j*nb];
                                py = py0;
                                py[0] += px0[0];
                                py[1] += px0[1];
                                py[2] += px0[2];
                                py[3] += px0[3];
                                py[4] += px0[4];
                                py[5] += px0[5];
                                py[6] += px0[6];
                            }
                            break;
                    }
                }
            }
        }
            break;

        default:
        {
            if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i, py0, num_nnz_row, k, j, pA, px0, py)
                {
                    myid = omp_get_thread_num();
                    fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                    for (i=mybegin; i < myend; ++i) {
                        py0 = &y[i*nb];
                        num_nnz_row = IA[i+1] - IA[i];
                        switch(num_nnz_row) {

                            case 3:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            case 4:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            case 5:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            case 6:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            case 7:
```

```
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k) {
                                    j = JA[k];
                                    px0 = x+j*nb; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_darray_axpy (nb, 1.0, px0, py);
                                }
                                break;
                        }
                    }
                }
#endif
            }
            else {
                for (i = 0; i < ROW; ++i) {
                    py0 = &y[i*nb];
                    num_nnz_row = IA[i+1] - IA[i];
                    switch(num_nnz_row) {

                        case 3:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            break;

                        case 4:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            break;

                        case 5:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            break;

                        case 6:
                            k = IA[i];
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);

                            k ++;
                            j = JA[k];
                            px0 = x+j*nb; // &x[j*nb];
                            py = py0;
                            fasp_blas_darray_axpy (nb, 1.0, px0, py);
```

```

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            case 7:
                                k = IA[i];
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                k ++;
                                j = JA[k];
                                px0 = x+j*nb; // &x[j*nb];
                                py = py0;
                                fasp_blas_darray_axpy (nb, 1.0, px0, py);

                                break;

                            default:
                                for (k = IA[i]; k < IA[i+1]; ++k) {
                                    j = JA[k];
                                    px0 = x+j*nb; // &x[j*nb];
                                    py = py0;
                                    fasp_blas_darray_axpy (nb, 1.0, px0, py);
                                }
                                break;
                        }
                    }
                }
            }
            break;
        }
}
```

```
void fasp_blas_dbsr_mxm (const dBSRmat  *A,
                         const dBSRmat  *B,
                         dBSRmat        *C)
{

    INT i,j,k,l,count;
    INT *JD = (INT *)fasp_mem_calloc(B->COL,sizeof(INT));

    const INT nb  = A->nb;
    const INT nb2 = nb*nb;

    // check A and B see if there are compatible for multiplication
    if ( (A->COL != B->ROW) && (A->nb != B->nb ) ) {
        printf("### ERROR: Matrix sizes do not match!\n");
        fasp_chkerr(ERROR_MAT_SIZE, __FUNCTION__);
    }

    C->ROW = A->ROW;
    C->COL = B->COL;
    C->nb  = A->nb;
    C->storage_manner = A->storage_manner;

    C->val = NULL;
    C->JA  = NULL;
    C->IA  = (INT*)fasp_mem_calloc(C->ROW+1,sizeof(INT));

    REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));

    for (i=0;i<B->COL;++i) JD[i]=-1;

    // step 1:  Find first the structure IA of C
    for (i=0;i<C->ROW;++i) {
        count=0;

        for (k=A->IA[i];k<A->IA[i+1];++k) {
            for (j=B->IA[A->JA[k]];j<B->IA[A->JA[k]+1];++j) {
                for (l=0;l<count;l++) {
                    if (JD[l]==B->JA[j]) break;
                }

                if (l==count) {
                    JD[count]=B->JA[j];
                    count++;
                }
            }
        }
        C->IA[i+1]=count;
        for (j=0;j<count;++j) {
            JD[j]=-1;
        }
    }

    for (i=0;i<C->ROW;++i) C->IA[i+1]+=C->IA[i];

    // step 2:  Find the structure JA of C
    INT countJD;

    C->JA=(INT*)fasp_mem_calloc(C->IA[C->ROW],sizeof(INT));

    for (i=0;i<C->ROW;++i) {
        countJD=0;
        count=C->IA[i];
        for (k=A->IA[i];k<A->IA[i+1];++k) {
            for (j=B->IA[A->JA[k]];j<B->IA[A->JA[k]+1];++j) {
                for (l=0;l<countJD;l++) {
                    if (JD[l]==B->JA[j]) break;
                }

                if (l==countJD) {
                    C->JA[count]=B->JA[j];
                    JD[countJD]=B->JA[j];
                    count++;
                    countJD++;
                }
            }
        }

        //for (j=0;j<countJD;++j) JD[j]=-1;
        fasp_iarray_set(countJD, JD, -1);
    }

    fasp_mem_free(JD); JD = NULL;

    // step 3:  Find the structure A of C
    C->val=(REAL*)fasp_mem_calloc((C->IA[C->ROW])*nb2,sizeof(REAL));

    for (i=0;i<C->ROW;++i) {
        for (j=C->IA[i];j<C->IA[i+1];++j) {

            fasp_darray_set(nb2, C->val+(j*nb2), 0x0);

            for (k=A->IA[i];k<A->IA[i+1];++k) {
                for (l=B->IA[A->JA[k]];l<B->IA[A->JA[k]+1];l++) {
                    if (B->JA[l]==C->JA[j]) {
                        fasp_blas_smat_mul (A->val+(k*nb2), B->val+(l*nb2), temp, nb);
                        fasp_blas_darray_axpy (nb2, 1.0, temp, C->val+(j*nb2));
                    } // end if
                } // end for l
            } // end for k
        } // end for j
    }    // end for i

    C->NNZ = C->IA[C->ROW]-C->IA[0];

    fasp_mem_free(temp); temp = NULL;

}

void fasp_blas_dbsr_rap1 (const dBSRmat  *R,
                          const dBSRmat  *A,
                          const dBSRmat  *P,
                          dBSRmat        *B)
{
    const INT   row=R->ROW, col=P->COL, nb=A->nb, nb2=A->nb*A->nb;

    const REAL *rj=R->val, *aj=A->val, *pj=P->val;
    const INT  *ir=R->IA,  *ia=A->IA,  *ip=P->IA;
    const INT  *jr=R->JA,  *ja=A->JA,  *jp=P->JA;

    REAL      *acj;
    INT       *iac, *jac;

    INT nB=A->NNZ;
    INT i,i1,j,jj,k,length;
    INT begin_row,end_row,begin_rowA,end_rowA,begin_rowR,end_rowR;
    INT istart,iistart,count;

    INT *index=(INT *)fasp_mem_calloc(A->COL,sizeof(INT));

    REAL *smat_tmp=(REAL *)fasp_mem_calloc(nb2,sizeof(REAL));

    INT *iindex=(INT *)fasp_mem_calloc(col,sizeof(INT));

    for (i=0; i<A->COL; ++i)  index[i] = -2;

    memcpy(iindex,index,col*sizeof(INT));

    jac=(INT*)fasp_mem_calloc(nB,sizeof(INT));

    iac=(INT*)fasp_mem_calloc(row+1,sizeof(INT));

    REAL *temp=(REAL*)fasp_mem_calloc(A->COL*nb2,sizeof(REAL));

    iac[0] = 0;

    // First loop:  form sparsity pattern of R*A*P
    for (i=0; i < row; ++i) {
        // reset istart and length at the beginning of each loop
        istart = -1; length = 0; i1 = i+1;

        // go across the rows in R
        begin_rowR=ir[i]; end_rowR=ir[i1];
        for (jj=begin_rowR; jj<end_rowR; ++jj) {
            j = jr[jj];
            // for each column in A
            begin_rowA=ia[j]; end_rowA=ia[j+1];
            for (k=begin_rowA; k<end_rowA; ++k) {
                if (index[ja[k]] == -2) {
                    index[ja[k]] = istart;
                    istart = ja[k];
                    ++length;
                }
            }
        }

        // book-keeping [resetting length and setting iistart]
        count = length; iistart = -1; length = 0;

        // use each column that would have resulted from R*A
        for (j=0; j < count; ++j) {
            jj = istart;
            istart = index[istart];
            index[jj] = -2;

            // go across the row of P
            begin_row=ip[jj]; end_row=ip[jj+1];
            for (k=begin_row; k<end_row; ++k) {
                // pull out the appropriate columns of P
                if (iindex[jp[k]] == -2){
                    iindex[jp[k]] = iistart;
                    iistart = jp[k];
                    ++length;
                }
            } // end for k
        } // end for j

        // set B->IA
        iac[i1]=iac[i]+length;

        if (iac[i1]>nB) {
            nB=nB*2;
            jac=(INT*)fasp_mem_realloc(jac, nB*sizeof(INT));
        }

        // put the correct columns of p into the column list of the products
        begin_row=iac[i]; end_row=iac[i1];
        for (j=begin_row; j<end_row; ++j) {
            // put the value in B->JA
            jac[j] = iistart;
            // set istart to the next value
            iistart = iindex[iistart];
            // set the iindex spot to 0
            iindex[jac[j]] = -2;
        } // end j
    } // end i:  First loop

    jac=(INT*)fasp_mem_realloc(jac,(iac[row])*sizeof(INT));

    acj=(REAL*)fasp_mem_calloc(iac[row]*nb2,sizeof(REAL));

    INT *BTindex=(INT*)fasp_mem_calloc(col,sizeof(INT));

    // Second loop:  compute entries of R*A*P
    for (i=0; i<row; ++i) {
        i1 = i+1;
        // each col of B
        begin_row=iac[i]; end_row=iac[i1];
        for (j=begin_row; j<end_row; ++j) {
            BTindex[jac[j]]=j;
        }
        // reset istart and length at the beginning of each loop
        istart = -1; length = 0;

        // go across the rows in R
        begin_rowR=ir[i]; end_rowR=ir[i1];
        for ( jj=begin_rowR; jj<end_rowR; ++jj ) {
            j = jr[jj];
            // for each column in A
            begin_rowA=ia[j]; end_rowA=ia[j+1];
            for (k=begin_rowA; k<end_rowA; ++k) {
                if (index[ja[k]] == -2) {
                    index[ja[k]] = istart;
                    istart = ja[k];
                    ++length;
                }
                fasp_blas_smat_mul(&rj[jj*nb2],&aj[k*nb2],smat_tmp,nb);
                //fasp_darray_xpy(nb2,&temp[ja[k]*nb2], smat_tmp );
                fasp_blas_darray_axpy (nb2, 1.0, smat_tmp, &temp[ja[k]*nb2]);

                //temp[ja[k]]+=rj[jj]*aj[k];
                // change to   X = X+Y*Z
            }
        }

        // book-keeping [resetting length and setting iistart]
        // use each column that would have resulted from R*A
        for (j=0; j<length; ++j) {
            jj = istart;
            istart = index[istart];
            index[jj] = -2;

            // go across the row of P
            begin_row=ip[jj]; end_row=ip[jj+1];
            for (k=begin_row; k<end_row; ++k) {
                // pull out the appropriate columns of P
                //acj[BTindex[jp[k]]]+=temp[jj]*pj[k];
                fasp_blas_smat_mul(&temp[jj*nb2],&pj[k*nb2],smat_tmp,nb);
                //fasp_darray_xpy(nb2,&acj[BTindex[jp[k]]*nb2], smat_tmp );
                fasp_blas_darray_axpy(nb2, 1.0, smat_tmp, &acj[BTindex[jp[k]]*nb2]);

                // change to   X = X+Y*Z
            }
            //temp[jj]=0.0; // change to   X[nb,nb] = 0;
            fasp_darray_set(nb2,&temp[jj*nb2],0.0);
        }
    } // end for i:  Second loop
    // setup coarse matrix B
    B->ROW=row; B->COL=col;
    B->IA=iac; B->JA=jac; B->val=acj;
    B->NNZ=B->IA[B->ROW]-B->IA[0];

    B->nb=A->nb;
    B->storage_manner = A->storage_manner;

    fasp_mem_free(temp);     temp     = NULL;
    fasp_mem_free(index);    index    = NULL;
    fasp_mem_free(iindex);   iindex   = NULL;
    fasp_mem_free(BTindex);  BTindex  = NULL;
    fasp_mem_free(smat_tmp); smat_tmp = NULL;
}

void fasp_blas_dbsr_rap (const dBSRmat  *R,
                         const dBSRmat  *A,
                         const dBSRmat  *P,
                         dBSRmat        *B)
{
    const INT row=R->ROW, col=P->COL, nb=A->nb, nb2=A->nb*A->nb;

    const REAL *rj=R->val, *aj=A->val, *pj=P->val;
    const INT  *ir=R->IA,  *ia=A->IA,  *ip=P->IA;
    const INT  *jr=R->JA,  *ja=A->JA,  *jp=P->JA;

    REAL       *acj;
    INT        *iac, *jac;

    INT *Ps_marker = NULL;
    INT *As_marker = NULL;

#ifdef _OPENMP
    INT *P_marker = NULL;
    INT *A_marker = NULL;
    REAL *smat_tmp = NULL;
#endif

    INT i, i1, i2, i3, jj1, jj2, jj3;
    INT counter, jj_row_begining;

    INT nthreads = 1;

#ifdef _OPENMP
    INT myid, mybegin, myend, Ctemp;
    nthreads = fasp_get_num_threads();
#endif

    INT n_coarse = row;
    INT n_fine   = A->ROW;
    INT coarse_mul_nthreads = n_coarse * nthreads;
    INT fine_mul_nthreads = n_fine * nthreads;
    INT coarse_add_nthreads = n_coarse + nthreads;
    INT minus_one_length = coarse_mul_nthreads + fine_mul_nthreads;
    INT total_calloc = minus_one_length + coarse_add_nthreads + nthreads;

    Ps_marker = (INT *)fasp_mem_calloc(total_calloc, sizeof(INT));
    As_marker = Ps_marker + coarse_mul_nthreads;

    /*------------------------------------------------------*
*  First Pass:  Determine size of B and set up B_i  *
*--------------------------------------------------------*/
    iac = (INT *)fasp_mem_calloc(n_coarse+1, sizeof(INT));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

    REAL *tmp=(REAL *)fasp_mem_calloc(2*nthreads*nb2, sizeof(REAL));

#ifdef _OPENMP
    INT * RAP_temp = As_marker + fine_mul_nthreads;
    INT * part_end = RAP_temp + coarse_add_nthreads;

    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker = Ps_marker + myid * n_coarse;
            A_marker = As_marker + myid * n_fine;
            counter  = 0;
            for (i = mybegin; i < myend; ++i) {
                P_marker[i] = counter;
                jj_row_begining = counter;
                counter ++;
                for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                    i1 = jr[jj1];
                    for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                        i2 = ja[jj2];
                        if (A_marker[i2] != i) {
                            A_marker[i2] = i;
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                                i3 = jp[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = counter;
                                    counter ++;
                                }
                            }
                        }
                    }
                }
                RAP_temp[i+myid] = jj_row_begining;
            }
            RAP_temp[myend+myid] = counter;
            part_end[myid] = myend + myid + 1;
        }
        fasp_iarray_cp(part_end[0], RAP_temp, iac);
        counter = part_end[0];
        Ctemp = 0;
        for (i1 = 1; i1 < nthreads; i1 ++) {
            Ctemp += RAP_temp[part_end[i1-1]-1];
            for (jj1 = part_end[i1-1]+1; jj1 < part_end[i1]; jj1 ++) {
                iac[counter] = RAP_temp[jj1] + Ctemp;
                counter ++;
            }
        }
    }
    else {
#endif
        counter = 0;
        for (i = 0; i < row; ++ i) {
            Ps_marker[i] = counter;
            jj_row_begining = counter;
            counter ++;

            for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                i1 = jr[jj1];
                for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                    i2 = ja[jj2];
                    if (As_marker[i2] != i) {
                        As_marker[i2] = i;
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                            i3 = jp[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = counter;
                                counter ++;
                            }
                        }
                    }
                }
                }
            }
            iac[i] = jj_row_begining;
        }
#ifdef _OPENMP
    }
#endif

    iac[row] = counter;

    jac=(INT*)fasp_mem_calloc(iac[row], sizeof(INT));

    acj=(REAL*)fasp_mem_calloc(iac[row]*nb2, sizeof(REAL));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

    /*-----------------------------------------------------*
* Second Pass:  compute entries of B=R*A*P             *
*-----------------------------------------------------*/
#ifdef _OPENMP
    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
counter, i, jj_row_begining, jj1, i1, jj2, i2,   \
jj3, i3, smat_tmp)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker = Ps_marker + myid * n_coarse;
            A_marker = As_marker + myid * n_fine;
            smat_tmp = tmp + myid*2*nb2;
            counter = iac[mybegin];
            for (i = mybegin; i < myend; ++i) {
                P_marker[i] = counter;
                jj_row_begining = counter;
                jac[counter] = i;
                fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
                counter ++;

                for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                    i1 = jr[jj1];
                    for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                        fasp_blas_smat_mul(&rj[jj1*nb2],&aj[jj2*nb2], smat_tmp, nb);
                        i2 = ja[jj2];
                        if (A_marker[i2] != i) {
                            A_marker[i2] = i;
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                                i3 = jp[jj3];
                                fasp_blas_smat_mul(smat_tmp, &pj[jj3*nb2], smat_tmp+nb2, nb);
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = counter;
                                    fasp_darray_cp(nb2, smat_tmp+nb2, &acj[counter*nb2]);
                                    jac[counter] = i3;
                                    counter ++;
                                }
                                else {
                                    fasp_blas_darray_axpy(nb2, 1.0, smat_tmp+nb2,
                                                          &acj[P_marker[i3]*nb2]);
                                }
                            }
                        }
                        else {
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++) {
                                i3 = jp[jj3];
                                fasp_blas_smat_mul(smat_tmp, &pj[jj3*nb2], smat_tmp+nb2, nb);
                                fasp_blas_darray_axpy(nb2, 1.0, smat_tmp+nb2,
                                                      &acj[P_marker[i3]*nb2]);
                            }
                        }
                    }
                }
            }
        }
    }
    else {
#endif
        counter = 0;
        for (i = 0; i < row; ++i) {
            Ps_marker[i] = counter;
            jj_row_begining = counter;
            jac[counter] = i;
            fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
            counter ++;

            for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                i1 = jr[jj1];
                for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                    fasp_blas_smat_mul(&rj[jj1*nb2],&aj[jj2*nb2], tmp, nb);
                    i2 = ja[jj2];
                    if (As_marker[i2] != i) {
                        As_marker[i2] = i;
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                            i3 = jp[jj3];
                            fasp_blas_smat_mul(tmp, &pj[jj3*nb2], tmp+nb2, nb);
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = counter;
                                fasp_darray_cp(nb2, tmp+nb2, &acj[counter*nb2]);
                                jac[counter] = i3;
                                counter ++;
                            }
                            else {
                                fasp_blas_darray_axpy(nb2, 1.0, tmp+nb2,
                                                      &acj[Ps_marker[i3]*nb2]);
                            }
                        }
                    }
                    else {
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++) {
                            i3 = jp[jj3];
                            fasp_blas_smat_mul(tmp, &pj[jj3*nb2], tmp+nb2, nb);
                            fasp_blas_darray_axpy(nb2, 1.0, tmp+nb2, &acj[Ps_marker[i3]*nb2]);
                        }
                    }
                }
            }
        }
#ifdef _OPENMP
    }
#endif
    // setup coarse matrix B
    B->ROW=row; B->COL=col;
    B->IA=iac; B->JA=jac; B->val=acj;
    B->NNZ=B->IA[B->ROW]-B->IA[0];
    B->nb=A->nb;
    B->storage_manner = A->storage_manner;

    fasp_mem_free(Ps_marker); Ps_marker = NULL;
    fasp_mem_free(tmp);       tmp       = NULL;
}

void fasp_blas_dbsr_rap_agg (const dBSRmat  *R,
                             const dBSRmat  *A,
                             const dBSRmat  *P,
                             dBSRmat        *B)
{
    const INT row=R->ROW, col=P->COL, nb2=A->nb*A->nb;

    const REAL *aj=A->val;
    const INT *ir=R->IA, *ia=A->IA, *ip=P->IA;
    const INT *jr=R->JA, *ja=A->JA, *jp=P->JA;

    INT  *iac, *jac;
    REAL *acj;
    INT  *Ps_marker = NULL;
    INT  *As_marker = NULL;

#ifdef _OPENMP
    INT  *P_marker = NULL;
    INT  *A_marker = NULL;
#endif

    INT i, i1, i2, i3, jj1, jj2, jj3;
    INT counter, jj_row_begining;

    INT nthreads = 1;

#ifdef _OPENMP
    INT myid, mybegin, myend, Ctemp;
    nthreads = fasp_get_num_threads();
#endif

    INT n_coarse = row;
    INT n_fine   = A->ROW;
    INT coarse_mul_nthreads = n_coarse * nthreads;
    INT fine_mul_nthreads = n_fine * nthreads;
    INT coarse_add_nthreads = n_coarse + nthreads;
    INT minus_one_length = coarse_mul_nthreads + fine_mul_nthreads;
    INT total_calloc = minus_one_length + coarse_add_nthreads + nthreads;

    Ps_marker = (INT *)fasp_mem_calloc(total_calloc, sizeof(INT));
    As_marker = Ps_marker + coarse_mul_nthreads;

    /*-----------------------------------------------------*
* First Pass:  Determine size of B and set up B_i  *
*-----------------------------------------------------*/
    iac = (INT *)fasp_mem_calloc(n_coarse+1, sizeof(INT));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

#ifdef _OPENMP
    INT * RAP_temp = As_marker + fine_mul_nthreads;
    INT * part_end = RAP_temp + coarse_add_nthreads;

    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker = Ps_marker + myid * n_coarse;
            A_marker = As_marker + myid * n_fine;
            counter  = 0;
            for (i = mybegin; i < myend; ++i) {
                P_marker[i] = counter;
                jj_row_begining = counter;
                counter ++;
                for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                    i1 = jr[jj1];
                    for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                        i2 = ja[jj2];
                        if (A_marker[i2] != i) {
                            A_marker[i2] = i;
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                                i3 = jp[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = counter;
                                    counter ++;
                                }
                            }
                        }
                    }
                }
                RAP_temp[i+myid] = jj_row_begining;
            }
            RAP_temp[myend+myid] = counter;
            part_end[myid] = myend + myid + 1;
        }
        fasp_iarray_cp(part_end[0], RAP_temp, iac);
        counter = part_end[0];
        Ctemp = 0;
        for (i1 = 1; i1 < nthreads; i1 ++) {
            Ctemp += RAP_temp[part_end[i1-1]-1];
            for (jj1 = part_end[i1-1]+1; jj1 < part_end[i1]; jj1 ++) {
                iac[counter] = RAP_temp[jj1] + Ctemp;
                counter ++;
            }
        }
    }
    else {
#endif
        counter = 0;
        for (i = 0; i < row; ++ i) {
            Ps_marker[i] = counter;
            jj_row_begining = counter;
            counter ++;

            for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                i1 = jr[jj1];
                for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {
                    i2 = ja[jj2];
                    if (As_marker[i2] != i) {
                        As_marker[i2] = i;
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                            i3 = jp[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = counter;
                                counter ++;
                            }
                        }
                    }
                }
            }
            iac[i] = jj_row_begining;
        }
#ifdef _OPENMP
    }
#endif

    iac[row] = counter;

    jac=(INT*)fasp_mem_calloc(iac[row], sizeof(INT));

    acj=(REAL*)fasp_mem_calloc(iac[row]*nb2, sizeof(REAL));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

    /*-------------------------------------------------------*
*  Second Pass:  compute entries of B=R*A*P             *
*-------------------------------------------------------*/
#ifdef _OPENMP
    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker, \
counter, i, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker = Ps_marker + myid * n_coarse;
            A_marker = As_marker + myid * n_fine;
            counter = iac[mybegin];
            for (i = mybegin; i < myend; ++i) {
                P_marker[i] = counter;
                jj_row_begining = counter;
                jac[counter] = i;
                fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
                counter ++;

                for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                    i1 = jr[jj1];
                    for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {

                        i2 = ja[jj2];
                        if (A_marker[i2] != i) {
                            A_marker[i2] = i;
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                                i3 = jp[jj3];

                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = counter;
                                    fasp_darray_cp(nb2, &aj[jj2*nb2], &acj[counter*nb2]);
                                    jac[counter] = i3;
                                    counter ++;
                                }
                                else {
                                    fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
                                                          &acj[P_marker[i3]*nb2]);
                                }
                            }
                        }
                        else {
                            for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++) {
                                i3 = jp[jj3];
                                fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
                                                      &acj[P_marker[i3]*nb2]);
                            }
                        }
                    }
                }
            }
        }
    }
    else {
#endif
        counter = 0;
        for (i = 0; i < row; ++i) {
            Ps_marker[i] = counter;
            jj_row_begining = counter;
            jac[counter] = i;
            fasp_darray_set(nb2, &acj[counter*nb2], 0x0);
            counter ++;

            for (jj1 = ir[i]; jj1 < ir[i+1]; ++jj1) {
                i1 = jr[jj1];
                for (jj2 = ia[i1]; jj2 < ia[i1+1]; ++jj2) {

                    i2 = ja[jj2];
                    if (As_marker[i2] != i) {
                        As_marker[i2] = i;
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; ++jj3) {
                            i3 = jp[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = counter;
                                fasp_darray_cp(nb2, &aj[jj2*nb2], &acj[counter*nb2]);
                                jac[counter] = i3;
                                counter ++;
                            }
                            else {
                                fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
                                                      &acj[Ps_marker[i3]*nb2]);
                            }
                        }
                    }
                    else {
                        for (jj3 = ip[i2]; jj3 < ip[i2+1]; jj3 ++) {
                            i3 = jp[jj3];
                            fasp_blas_darray_axpy(nb2, 1.0, &aj[jj2*nb2],
                                                  &acj[Ps_marker[i3]*nb2]);
                        }
                    }
                }
            }
        }
#ifdef _OPENMP
    }
#endif

    // setup coarse matrix B
    B->ROW=row; B->COL=col;
    B->IA=iac; B->JA=jac; B->val=acj;
    B->NNZ=B->IA[B->ROW]-B->IA[0];
    B->nb=A->nb;
    B->storage_manner = A->storage_manner;

    fasp_mem_free(Ps_marker); Ps_marker = NULL;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.91 BlaSpmvCSR.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_aAxpy (const REAL alpha, const dCSRmat ∗A, const REAL ∗x, REAL ∗y)

  Matrix-vector multiplication y = alpha∗A∗x + y.
- void fasp_blas_ldcsr_aAxpy (const REAL alpha, const dCSRmat ∗A, const LONGREAL ∗x, REAL ∗y)

  Matrix-vector multiplication y = alpha∗A∗x + y.
- void fasp_blas_dcsr_aAxpy_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 thy 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.

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_aAxpy()

```
void fasp_blas_dcsr_aAxpy (
             const REAL alpha,
             const dCSRmat * A,
             const REAL * x,
             REAL * y )
```

Matrix-vector multiplication y = alpha∗A∗x + y.

Parameters

alpha	REAL factor alpha
A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	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_aAxpy_agg()

```
void fasp_blas_dcsr_aAxpy_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

alpha	REAL factor alpha
A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	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 BlaSpmvCSR.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

A	Pointer to dCSRmat matrix
alpha	REAL factor alpha
B	Pointer to dCSRmat matrix
beta	REAL factor beta
C	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 BlaSpmvCSR.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

A	Pointer to dCSRmat matrix A
alpha	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

A	Pointer to the dCSRmat matrix A
B	Pointer to the dCSRmat matrix B
C	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

A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	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

A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	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

Pt	Pointer to the restriction matrix
A	Pointer to the fine coefficient matrix
P	Pointer to the prolongation matrix
Ac	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 ltz CSR format. In ltx 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_ltz[k] = ia_usual[k]+1, ja_ltz[k] = ja_usual[k]+1, a_ltz[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

R	Pointer to the dCSRmat matrix R
A	Pointer to the dCSRmat matrix A
P	Pointer to the dCSRmat matrix P
RAP	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∗A∗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 BlaSpmvCSR.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∗A∗P.

Parameters

R	pointer to the dCSRmat matrix
A	pointer to the dCSRmat matrix
P	pointer to the dCSRmat matrix
B	pointer to dCSRmat matrix equal to R∗A∗P
icor_ysk	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

R	Pointer to the dCSRmat matrix R
A	Pointer to the dCSRmat matrix A
P	Pointer to the dCSRmat matrix P
RAP	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

R	Pointer to the dCSRmat matrix R
A	Pointer to the dCSRmat matrix A
P	Pointer to the dCSRmat matrix P
B	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 BlaSpmvCSR.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 alpha = y'∗A∗x

Parameters

A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	Pointer to array y

Author

Chensong Zhang

Date

07/01/2009

Definition at line 834 of file BlaSpmvCSR.c.

9.91.2.15 fasp_blas_ldcsr_aAxpy()

```
void fasp_blas_ldcsr_aAxpy (
            const REAL alpha,
            const dCSRmat * A,
            const LONGREAL * x,
            REAL * y )
```

Matrix-vector multiplication y = alpha∗A∗x + y.

Parameters

alpha	REAL factor alpha
A	Pointer to dCSRmat matrix A
x	Pointer to array x
y	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.

```

#include <math.h>
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

extern unsigned long total_alloc_mem;
extern unsigned long total_alloc_count;
/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_blas_dcsr_add(const dCSRmat* A, const REAL alpha, const dCSRmat* B,
                         const REAL beta, dCSRmat* C)
{
    INT i, j, k, l;
    INT count = 0, added, countrow;

    SHORT status = FASP_SUCCESS, use_openmp = FALSE;

#ifdef _OPENMP
    INT mybegin, myend, myid, nthreads;
    if (A->nnz > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (A->row != B->row || A->col != B->col) {
        printf("### ERROR: Matrix sizes do not match!\n");
        status = ERROR_MAT_SIZE;
        goto FINISHED;
    }

    if (A == NULL && B == NULL) {
        C->row = 0;
        C->col = 0;
        C->nnz = 0;
        status = FASP_SUCCESS;
        goto FINISHED;
    }

    if (A->nnz == 0 && B->nnz == 0) {
        C->row = A->row;
        C->col = A->col;
        C->nnz = A->nnz;
        status = FASP_SUCCESS;
        goto FINISHED;
    }

    // empty matrix A
    if (A->nnz == 0 || A == NULL) {
        fasp_dcsr_alloc(B->row, B->col, B->nnz, C);
        memcpy(C->IA, B->IA, (B->row + 1) * sizeof(INT));
        memcpy(C->JA, B->JA, (B->nnz) * sizeof(INT));

        if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid, mybegin, myend, i)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end(myid, nthreads, A->nnz, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) C->val[i] = B->val[i] * beta;
            }
#endif
        } else {
            for (i = 0; i < A->nnz; ++i) C->val[i] = B->val[i] * beta;
        }

        status = FASP_SUCCESS;
        goto FINISHED;
    }

    // empty matrix B
    if (B->nnz == 0 || B == NULL) {
        fasp_dcsr_alloc(A->row, A->col, A->nnz, C);
        memcpy(C->IA, A->IA, (A->row + 1) * sizeof(INT));
        memcpy(C->JA, A->JA, (A->nnz) * sizeof(INT));

        if (use_openmp) {
#ifdef _OPENMP
            INT mybegin, myend, myid;
#pragma omp parallel private(myid, mybegin, myend, i)
            {
                myid = omp_get_thread_num();
                fasp_get_start_end(myid, nthreads, A->nnz, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) C->val[i] = A->val[i] * alpha;
            }
#endif
        } else {
            for (i = 0; i < A->nnz; ++i) C->val[i] = A->val[i] * alpha;
        }

        status = FASP_SUCCESS;
        goto FINISHED;
    }

    C->row = A->row;
    C->col = A->col;

    C->IA = (INT*)fasp_mem_calloc(C->row + 1, sizeof(INT));

    // allocate work space for C->JA and C->val
    C->JA = (INT*)fasp_mem_calloc(A->nnz + B->nnz, sizeof(INT));

    C->val = (REAL*)fasp_mem_calloc(A->nnz + B->nnz, sizeof(REAL));

    // initial C->IA
    memset(C->IA, 0, sizeof(INT) * (C->row + 1));
    memset(C->JA, -1, sizeof(INT) * (A->nnz + B->nnz));

    for (i = 0; i < A->row; ++i) {
        countrow = 0;
        for (j = A->IA[i]; j < A->IA[i + 1]; ++j) {
            C->val[count] = alpha * A->val[j];
            C->JA[count]  = A->JA[j];
            C->IA[i + 1]++;
            count++;
            countrow++;
        } // end for js

        for (k = B->IA[i]; k < B->IA[i + 1]; ++k) {
            added = 0;

            for (l = C->IA[i]; l < C->IA[i] + countrow + 1; l++) {
                if (B->JA[k] == C->JA[l]) {
                    C->val[l] = C->val[l] + beta * B->val[k];
                    added     = 1;
                    break;
                }
            } // end for l

            if (added == 0) {
                C->val[count] = beta * B->val[k];
                C->JA[count]  = B->JA[k];
                C->IA[i + 1]++;
                count++;
            }

        } // end for k

        C->IA[i + 1] += C->IA[i];
    }

    C->nnz = count;
    C->JA  = (INT*)fasp_mem_realloc(C->JA, (count) * sizeof(INT));
    C->val = (REAL*)fasp_mem_realloc(C->val, (count) * sizeof(REAL));

#if MULTI_COLOR_ORDER
    C->color = 0;
    C->IC    = NULL;
    C->ICMAP = NULL;
#endif

FINISHED:
    return status;
}

void fasp_blas_dcsr_axm(dCSRmat* A, const REAL alpha)
{
    const INT nnz = A->nnz;

    // A direct calculation can be written as:
    fasp_blas_darray_ax(nnz, alpha, A->val);
}

void fasp_blas_dcsr_mxv(const dCSRmat* A, const REAL* x, REAL* y)
{
    const INT   m  = A->row;
    const INT * ia = A->IA, *ja = A->JA;
    const REAL* aj = A->val;

    INT           i, k, begin_row, end_row, nnz_row;
    register REAL temp;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
        INT myid, mybegin, myend;

#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row,    \
nnz_row, k)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                nnz_row   = end_row - begin_row;
                switch (nnz_row) {
                    case 3:
                        k = begin_row;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        break;
                    case 4:
                        k = begin_row;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        break;
                    case 5:
                        k = begin_row;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        break;
                    case 6:
                        k = begin_row;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        break;
                    case 7:
                        k = begin_row;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        k++;
                        temp += aj[k] * x[ja[k]];
                        break;
                    default:
                        for (k = begin_row; k < end_row; ++k) {
                            temp += aj[k] * x[ja[k]];
                        }
                        break;
                }
                y[i] = temp;
            }
        }
    }

    else {
        for (i = 0; i < m; ++i) {
            temp      = 0.0;
            begin_row = ia[i];
            end_row   = ia[i + 1];
            nnz_row   = end_row - begin_row;
            switch (nnz_row) {
                case 3:
                    k = begin_row;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    break;
                case 4:
                    k = begin_row;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    break;
                case 5:
                    k = begin_row;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    break;
                case 6:
                    k = begin_row;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    break;
                case 7:
                    k = begin_row;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    k++;
                    temp += aj[k] * x[ja[k]];
                    break;
                default:
                    for (k = begin_row; k < end_row; ++k) {
                        temp += aj[k] * x[ja[k]];
                    }
                    break;
            }
            y[i] = temp;
        }
    }
}

void fasp_blas_dcsr_mxv_agg(const dCSRmat* A, const REAL* x, REAL* y)
{
    const INT     m  = A->row;
    const INT *   ia = A->IA, *ja = A->JA;
    INT           i, k, begin_row, end_row;
    register REAL temp;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
```

```
#endif

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, i, mybegin, myend, temp, begin_row, end_row, k)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                y[i] = temp;
            }
        }
    } else {
#endif
        for (i = 0; i < m; ++i) {
            temp      = 0.0;
            begin_row = ia[i];
            end_row   = ia[i + 1];
            for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
            y[i] = temp;
        }
#ifdef _OPENMP
    }
#endif
}

void fasp_blas_dcsr_aAxpy(const REAL alpha, const dCSRmat* A, const REAL* x, REAL* y)
{
    const INT     m  = A->row;
    const INT *   ia = A->IA, *ja = A->JA;
    const REAL*   aj = A->val;
    INT           i, k, begin_row, end_row;
    register REAL temp;
    SHORT         nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif
    if (alpha == 1.0) {
        if (use_openmp) {
            INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
#endif
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] += temp;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] += temp;
            }
        }
    }

    else if (alpha == -1.0) {
        if (use_openmp) {
            INT myid, mybegin, myend;
            INT S = 0;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, temp, i, begin_row, end_row, k)
#endif

            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] -= temp;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] -= temp;
            }
        }
    }

    else {
        if (use_openmp) {
            INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
#endif
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] += temp * alpha;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] += temp * alpha;
            }
        }
    }
}

void fasp_blas_ldcsr_aAxpy(const REAL alpha, const dCSRmat* A, const LONGREAL* x,
                           REAL* y)
{
    const INT          m  = A->row;
    const INT *        ia = A->IA, *ja = A->JA;
    const REAL*        aj = A->val;
    INT                i, k, begin_row, end_row;
    register LONGREAL temp;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (alpha == 1.0) {
        if (use_openmp) {
            INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
#endif
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] += temp;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] += temp;
            }
        }
    }

    else if (alpha == -1.0) {
        if (use_openmp) {
            INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, temp, i, begin_row, end_row, k)
#endif
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] -= temp;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] -= temp;
            }
        }
    }

    else {
        if (use_openmp) {
            INT myid, mybegin, myend;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i, temp, begin_row, end_row, k)
#endif
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                    y[i] += temp * alpha;
                }
            }
        } else {
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
                y[i] += temp * alpha;
            }
        }
    }
}

void fasp_blas_dcsr_aAxpy_agg(const REAL alpha, const dCSRmat* A, const REAL* x,
                              REAL* y)
{
    const INT  m  = A->row;
    const INT *ia = A->IA, *ja = A->JA;

    INT           i, k, begin_row, end_row;
    register REAL temp;

    if (alpha == 1.0) {
#ifdef _OPENMP
        if (m > OPENMP_HOLDS) {
            INT myid, mybegin, myend;
            INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)
```

```
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                    y[i] += temp;
                }
            }
        } else {
#endif
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                y[i] += temp;
            }
#ifdef _OPENMP
        }
#endif
    } else if (alpha == -1.0) {
#ifdef _OPENMP
        if (m > OPENMP_HOLDS) {
            INT myid, mybegin, myend;
            INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                    y[i] -= temp;
                }
            }
        } else {
#endif
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                y[i] -= temp;
            }
#ifdef _OPENMP
        }
#endif
    }

    else {
#ifdef _OPENMP
        if (m > OPENMP_HOLDS) {
            INT myid, mybegin, myend;
            INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, i, mybegin, myend, begin_row, end_row, temp, k)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                for (i = mybegin; i < myend; ++i) {
                    temp      = 0.0;
                    begin_row = ia[i];
                    end_row   = ia[i + 1];
                    for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                    y[i] += temp * alpha;
                }
            }
        } else {
#endif
            for (i = 0; i < m; ++i) {
                temp      = 0.0;
                begin_row = ia[i];
                end_row   = ia[i + 1];
                for (k = begin_row; k < end_row; ++k) temp += x[ja[k]];
                y[i] += temp * alpha;
            }
#ifdef _OPENMP
        }
#endif
    }
```

```
}

REAL fasp_blas_dcsr_vmv(const dCSRmat* A, const REAL* x, const REAL* y)
{
    register REAL value = 0.0;
    const INT      m      = A->row;
    const INT *    ia = A->IA, *ja = A->JA;
    const REAL*    aj = A->val;
    INT            i, k, begin_row, end_row;
    register REAL temp;

    SHORT use_openmp = FALSE;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        use_openmp = TRUE;
    }
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+ :  value) private(i, temp, begin_row, end_row, k)
#endif
        for (i = 0; i < m; ++i) {
            temp      = 0.0;
            begin_row = ia[i];
            end_row   = ia[i + 1];
            for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
            value += y[i] * temp;
        }
    } else {
        for (i = 0; i < m; ++i) {
            temp      = 0.0;
            begin_row = ia[i];
            end_row   = ia[i + 1];
            for (k = begin_row; k < end_row; ++k) temp += aj[k] * x[ja[k]];
            value += y[i] * temp;
        }
    }
    return value;
}

void fasp_blas_dcsr_mxm(const dCSRmat* A, const dCSRmat* B, dCSRmat* C)
{
    INT i, j, k, l, count;

    INT* JD = (INT*)fasp_mem_calloc(B->col, sizeof(INT));

    C->row = A->row;
    C->col = B->col;
    C->val = NULL;
    C->JA  = NULL;
    C->IA  = (INT*)fasp_mem_calloc(C->row + 1, sizeof(INT));

    for (i = 0; i < B->col; ++i) JD[i] = -1;

    // step 1:  Find first the structure IA of C
    for (i = 0; i < C->row; ++i) {
        count = 0;

        for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
            for (j = B->IA[A->JA[k]]; j < B->IA[A->JA[k] + 1]; ++j) {
                for (l = 0; l < count; l++) {
                    if (JD[l] == B->JA[j]) break;
                }

                if (l == count) {
                    JD[count] = B->JA[j];
                    count++;
                }
            }
        }
        C->IA[i + 1] = count;
        for (j = 0; j < count; ++j) {
            JD[j] = -1;
        }
    }

    for (i = 0; i < C->row; ++i) C->IA[i + 1] += C->IA[i];

    // step 2:  Find the structure JA of C
    INT countJD;

    C->JA = (INT*)fasp_mem_calloc(C->IA[C->row], sizeof(INT));

    for (i = 0; i < C->row; ++i) {
        countJD = 0;
        count   = C->IA[i];
        for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
            for (j = B->IA[A->JA[k]]; j < B->IA[A->JA[k] + 1]; ++j) {
                for (l = 0; l < countJD; l++) {
                    if (JD[l] == B->JA[j]) break;
                }

                if (l == countJD) {
                    C->JA[count] = B->JA[j];
                    JD[countJD]  = B->JA[j];
                    count++;
                    countJD++;
                }
            }
        }

        // for (j=0;j<countJD;++j) JD[j]=-1;
        fasp_iarray_set(countJD, JD, -1);
    }

    fasp_mem_free(JD);
    JD = NULL;

    // step 3:  Find the structure A of C
    C->val = (REAL*)fasp_mem_calloc(C->IA[C->row], sizeof(REAL));

    for (i = 0; i < C->row; ++i) {
        for (j = C->IA[i]; j < C->IA[i + 1]; ++j) {
            C->val[j] = 0;
            for (k = A->IA[i]; k < A->IA[i + 1]; ++k) {
                for (l = B->IA[A->JA[k]]; l < B->IA[A->JA[k] + 1]; l++) {
                    if (B->JA[l] == C->JA[j]) {
                        C->val[j] += A->val[k] * B->val[l];
                    } // end if
                }     // end for l
            }         // end for k
        }             // end for j
    }                 // end for i

    C->nnz = C->IA[C->row] - C->IA[0];
}

void fasp_blas_dcsr_rap(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
                        dCSRmat* RAP)
{
    const INT   n_coarse = R->row;
    const INT*  R_i      = R->IA;
    const INT*  R_j      = R->JA;
    const REAL* R_data   = R->val;

    const INT   n_fine = A->row;
    const INT*  A_i    = A->IA;
    const INT*  A_j    = A->JA;
    const REAL* A_data = A->val;

    const INT*  P_i    = P->IA;
    const INT*  P_j    = P->JA;
    const REAL* P_data = P->val;

    INT   RAP_size;
    INT*  RAP_i    = NULL;
    INT*  RAP_j    = NULL;
    REAL* RAP_data = NULL;

#ifdef _OPENMP
    INT* P_marker = NULL;
    INT* A_marker = NULL;
#endif

    INT* Ps_marker = NULL;
    INT* As_marker = NULL;

    INT  ic, i1, i2, i3, jj1, jj2, jj3;
    INT  jj_counter, jj_row_begining;
    REAL r_entry, r_a_product, r_a_p_product;

    INT nthreads = 1;

#ifdef _OPENMP
    INT myid, mybegin, myend, Ctemp;
    nthreads = fasp_get_num_threads();
#endif

    INT coarse_mul_nthreads = n_coarse * nthreads;
    INT fine_mul_nthreads   = n_fine * nthreads;
    INT coarse_add_nthreads = n_coarse + nthreads;
    INT minus_one_length    = coarse_mul_nthreads + fine_mul_nthreads;
    INT total_calloc        = minus_one_length + coarse_add_nthreads + nthreads;

    Ps_marker = (INT*)fasp_mem_calloc(total_calloc, sizeof(INT));
    As_marker = Ps_marker + coarse_mul_nthreads;

    /*------------------------------------------------------*
*  First Pass:  Determine size of RAP and set up RAP_i  *
*------------------------------------------------------*/
    RAP_i = (INT*)fasp_mem_calloc(n_coarse + 1, sizeof(INT));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

#ifdef _OPENMP
    INT* RAP_temp = As_marker + fine_mul_nthreads;
    INT* part_end = RAP_temp + coarse_add_nthreads;

    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker,      \
jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2,    \
jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker   = Ps_marker + myid * n_coarse;
            A_marker   = As_marker + myid * n_fine;
            jj_counter = 0;
            for (ic = mybegin; ic < myend; ic++) {
                P_marker[ic]    = jj_counter;
                jj_row_begining = jj_counter;
                jj_counter++;

                for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                    i1 = R_j[jj1];
                    for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                        i2 = A_j[jj2];
                        if (A_marker[i2] != ic) {
                            A_marker[i2] = ic;
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                                i3 = P_j[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = jj_counter;
                                    jj_counter++;
                                }
                            }
                        }
                    }
                }

                RAP_temp[ic + myid] = jj_row_begining;
            }
            RAP_temp[myend + myid] = jj_counter;

            part_end[myid] = myend + myid + 1;
        }
        fasp_iarray_cp(part_end[0], RAP_temp, RAP_i);
        jj_counter = part_end[0];
        Ctemp      = 0;
        for (i1 = 1; i1 < nthreads; i1++) {
            Ctemp += RAP_temp[part_end[i1 - 1] - 1];
            for (jj1 = part_end[i1 - 1] + 1; jj1 < part_end[i1]; jj1++) {
                RAP_i[jj_counter] = RAP_temp[jj1] + Ctemp;
                jj_counter++;
            }
        }
        RAP_size = RAP_i[n_coarse];
    }

    else {
#endif
        jj_counter = 0;
        for (ic = 0; ic < n_coarse; ic++) {
            Ps_marker[ic]   = jj_counter;
            jj_row_begining = jj_counter;
            jj_counter++;

            for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                i1 = R_j[jj1];

                for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                    i2 = A_j[jj2];
                    if (As_marker[i2] != ic) {
                        As_marker[i2] = ic;
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            i3 = P_j[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = jj_counter;
                                jj_counter++;
                            }
                        }
                    }
                }
            }

            RAP_i[ic] = jj_row_begining;
        }

        RAP_i[n_coarse] = jj_counter;
        RAP_size        = jj_counter;
#ifdef _OPENMP
    }
#endif

    RAP_j    = (INT*)fasp_mem_calloc(RAP_size, sizeof(INT));
    RAP_data = (REAL*)fasp_mem_calloc(RAP_size, sizeof(REAL));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

#ifdef _OPENMP
    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, P_marker, A_marker, jj_counter, \
ic, jj_row_begining, jj1, r_entry, i1, jj2,            \
r_a_product, i2, jj3, r_a_p_product, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker   = Ps_marker + myid * n_coarse;
            A_marker   = As_marker + myid * n_fine;
            jj_counter = RAP_i[mybegin];
            for (ic = mybegin; ic < myend; ic++) {
                P_marker[ic]         = jj_counter;
                jj_row_begining      = jj_counter;
                RAP_j[jj_counter]    = ic;
                RAP_data[jj_counter] = 0.0;
                jj_counter++;
                for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                    r_entry = R_data[jj1];

                    i1 = R_j[jj1];
                    for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                        r_a_product = r_entry * A_data[jj2];

                        i2 = A_j[jj2];
                        if (A_marker[i2] != ic) {
                            A_marker[i2] = ic;
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                                r_a_p_product = r_a_product * P_data[jj3];

                                i3 = P_j[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3]         = jj_counter;
                                    RAP_data[jj_counter] = r_a_p_product;
                                    RAP_j[jj_counter]    = i3;
                                    jj_counter++;
                                } else {
                                    RAP_data[P_marker[i3]] += r_a_p_product;
                                }
                            }
                        } else {
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                                i3            = P_j[jj3];
                                r_a_p_product = r_a_product * P_data[jj3];
                                RAP_data[P_marker[i3]] += r_a_p_product;
                            }
                        }
                    }
                }
            }
        }
    } else {
#endif
        jj_counter = 0;
        for (ic = 0; ic < n_coarse; ic++) {
            Ps_marker[ic]        = jj_counter;
            jj_row_begining      = jj_counter;
            RAP_j[jj_counter]    = ic;
            RAP_data[jj_counter] = 0.0;
            jj_counter++;

            for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                r_entry = R_data[jj1];

                i1 = R_j[jj1];
                for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                    r_a_product = r_entry * A_data[jj2];

                    i2 = A_j[jj2];
                    if (As_marker[i2] != ic) {
                        As_marker[i2] = ic;
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            r_a_p_product = r_a_product * P_data[jj3];

                            i3 = P_j[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3]        = jj_counter;
                                RAP_data[jj_counter] = r_a_p_product;
                                RAP_j[jj_counter]    = i3;
                                jj_counter++;
                            } else {
                                RAP_data[Ps_marker[i3]] += r_a_p_product;
                            }
                        }
                    } else {
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            i3            = P_j[jj3];
                            r_a_p_product = r_a_product * P_data[jj3];
                            RAP_data[Ps_marker[i3]] += r_a_p_product;
                        }
                    }
                }
            }
        }
#ifdef _OPENMP
    }
#endif

    RAP->row = n_coarse;
    RAP->col = n_coarse;
    RAP->nnz = RAP_size;
    RAP->IA  = RAP_i;
    RAP->JA  = RAP_j;
    RAP->val = RAP_data;

    fasp_mem_free(Ps_marker);
    Ps_marker = NULL;
}

void fasp_blas_dcsr_rap_agg(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
                            dCSRmat* RAP)
{
    const INT  n_coarse = R->row;
    const INT* R_i      = R->IA;
    const INT* R_j      = R->JA;

    const INT   n_fine = A->row;
    const INT*  A_i    = A->IA;
    const INT*  A_j    = A->JA;
    const REAL* A_data = A->val;

    const INT* P_i = P->IA;
    const INT* P_j = P->JA;

    INT   RAP_size;
    INT*  RAP_i    = NULL;
    INT*  RAP_j    = NULL;
    REAL* RAP_data = NULL;

#ifdef _OPENMP
    INT* P_marker = NULL;
    INT* A_marker = NULL;
#endif

    INT* Ps_marker = NULL;
    INT* As_marker = NULL;

    INT ic, i1, i2, i3, jj1, jj2, jj3;
    INT jj_counter, jj_row_begining;

    INT nthreads = 1;

#ifdef _OPENMP
    INT myid, mybegin, myend, Ctemp;
    nthreads = fasp_get_num_threads();
#endif

    INT coarse_mul_nthreads = n_coarse * nthreads;
    INT fine_mul_nthreads   = n_fine * nthreads;
    INT coarse_add_nthreads = n_coarse + nthreads;
    INT minus_one_length    = coarse_mul_nthreads + fine_mul_nthreads;
    INT total_calloc        = minus_one_length + coarse_add_nthreads + nthreads;

    Ps_marker = (INT*)fasp_mem_calloc(total_calloc, sizeof(INT));
    As_marker = Ps_marker + coarse_mul_nthreads;

    /*------------------------------------------------------*
*  First Pass:  Determine size of RAP and set up RAP_i  *
*------------------------------------------------------*/
    RAP_i = (INT*)fasp_mem_calloc(n_coarse + 1, sizeof(INT));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

#ifdef _OPENMP
    INT* RAP_temp = As_marker + fine_mul_nthreads;
    INT* part_end = RAP_temp + coarse_add_nthreads;

    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, Ctemp, P_marker, A_marker,       \
jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2,    \
jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker   = Ps_marker + myid * n_coarse;
            A_marker   = As_marker + myid * n_fine;
            jj_counter = 0;
            for (ic = mybegin; ic < myend; ic++) {
                P_marker[ic]    = jj_counter;
                jj_row_begining = jj_counter;
                jj_counter++;

                for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                    i1 = R_j[jj1];
                    for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                        i2 = A_j[jj2];
                        if (A_marker[i2] != ic) {
                            A_marker[i2] = ic;
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                                i3 = P_j[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3] = jj_counter;
                                    jj_counter++;
                                }
                            }
                        }
                    }
                }

                RAP_temp[ic + myid] = jj_row_begining;
            }
            RAP_temp[myend + myid] = jj_counter;

            part_end[myid] = myend + myid + 1;
        }
        fasp_iarray_cp(part_end[0], RAP_temp, RAP_i);
        jj_counter = part_end[0];
        Ctemp      = 0;
        for (i1 = 1; i1 < nthreads; i1++) {
            Ctemp += RAP_temp[part_end[i1 - 1] - 1];
            for (jj1 = part_end[i1 - 1] + 1; jj1 < part_end[i1]; jj1++) {
                RAP_i[jj_counter] = RAP_temp[jj1] + Ctemp;
                jj_counter++;
            }
        }
        RAP_size = RAP_i[n_coarse];
    }

    else {
#endif
        jj_counter = 0;
        for (ic = 0; ic < n_coarse; ic++) {
            Ps_marker[ic]    = jj_counter;
            jj_row_begining = jj_counter;
            jj_counter++;

            for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                i1 = R_j[jj1];

                for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                    i2 = A_j[jj2];
                    if (As_marker[i2] != ic) {
                        As_marker[i2] = ic;
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            i3 = P_j[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3] = jj_counter;
                                jj_counter++;
                            }
                        }
                    }
                }
            }

            RAP_i[ic] = jj_row_begining;
        }

        RAP_i[n_coarse] = jj_counter;
        RAP_size         = jj_counter;
#ifdef _OPENMP
    }
#endif

    RAP_j    = (INT*)fasp_mem_calloc(RAP_size, sizeof(INT));
    RAP_data = (REAL*)fasp_mem_calloc(RAP_size, sizeof(REAL));

    fasp_iarray_set(minus_one_length, Ps_marker, -1);

#ifdef _OPENMP
    if (n_coarse > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, P_marker, A_marker, jj_counter, \
ic, jj_row_begining, jj1, i1, jj2, i2, jj3, i3)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n_coarse, &mybegin, &myend);
            P_marker    = Ps_marker + myid * n_coarse;
            A_marker    = As_marker + myid * n_fine;
            jj_counter = RAP_i[mybegin];
            for (ic = mybegin; ic < myend; ic++) {
                P_marker[ic]          = jj_counter;
                jj_row_begining       = jj_counter;
                RAP_j[jj_counter]     = ic;
                RAP_data[jj_counter] = 0.0;
                jj_counter++;
                for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {

                    i1 = R_j[jj1];
                    for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {

                        i2 = A_j[jj2];
                        if (A_marker[i2] != ic) {
                            A_marker[i2] = ic;
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {

                                i3 = P_j[jj3];
                                if (P_marker[i3] < jj_row_begining) {
                                    P_marker[i3]          = jj_counter;
                                    RAP_data[jj_counter] = A_data[jj2];
                                    RAP_j[jj_counter]     = i3;
                                    jj_counter++;
                                } else {
                                    RAP_data[P_marker[i3]] += A_data[jj2];
                                }
                            }
                        } else {
                            for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                                i3 = P_j[jj3];
                                RAP_data[P_marker[i3]] += A_data[jj2];
                            }
                        }
                    }
                }
            }
        }
    } else {
#endif
        jj_counter = 0;
        for (ic = 0; ic < n_coarse; ic++) {
            Ps_marker[ic]         = jj_counter;
            jj_row_begining       = jj_counter;
            RAP_j[jj_counter]     = ic;
            RAP_data[jj_counter] = 0.0;
            jj_counter++;

            for (jj1 = R_i[ic]; jj1 < R_i[ic + 1]; jj1++) {
                i1 = R_j[jj1];
                for (jj2 = A_i[i1]; jj2 < A_i[i1 + 1]; jj2++) {
                    i2 = A_j[jj2];
                    if (As_marker[i2] != ic) {
                        As_marker[i2] = ic;
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            i3 = P_j[jj3];
                            if (Ps_marker[i3] < jj_row_begining) {
                                Ps_marker[i3]         = jj_counter;
                                RAP_data[jj_counter] = A_data[jj2];
                                RAP_j[jj_counter]     = i3;
                                jj_counter++;
                            } else {
                                RAP_data[Ps_marker[i3]] += A_data[jj2];
                            }
                        }
                    } else {
                        for (jj3 = P_i[i2]; jj3 < P_i[i2 + 1]; jj3++) {
                            i3 = P_j[jj3];
                            RAP_data[Ps_marker[i3]] += A_data[jj2];
                        }
                    }
                }
            }
        }
#ifdef _OPENMP
    }
#endif

    RAP->row = n_coarse;
    RAP->col = n_coarse;
    RAP->nnz = RAP_size;
    RAP->IA  = RAP_i;
    RAP->JA  = RAP_j;
    RAP->val = RAP_data;

    fasp_mem_free(Ps_marker);
    Ps_marker = NULL;
}

void fasp_blas_dcsr_rap_agg1(const dCSRmat* R, const dCSRmat* A, const dCSRmat* P,
                             dCSRmat* B)
{
    const INT   row = R->row, col = P->col;
    const INT * ir = R->IA, *ia = A->IA, *ip = P->IA;
    const INT * jr = R->JA, *ja = A->JA, *jp = P->JA;
    const REAL* aj = A->val;

    INT * iac, *jac;
    REAL* acj;

    INT* index  = (INT*)fasp_mem_calloc(A->col, sizeof(INT));
    INT* iindex = (INT*)fasp_mem_calloc(col, sizeof(INT));

    INT nB = A->nnz;
    INT i, i1, j, jj, k, length;
    INT begin_row, end_row, begin_rowA, end_rowA, begin_rowR, end_rowR;
    INT istart, iistart, count;

    // for (i=0; i<A->col; ++i) index[i] = -2;
    fasp_iarray_set(A->col, index, -2);

    // memcpy(iindex,index,col*sizeof(INT));
    fasp_iarray_cp(col, index, iindex);

    jac = (INT*)fasp_mem_calloc(nB, sizeof(INT));

    iac = (INT*)fasp_mem_calloc(row + 1, sizeof(INT));

    REAL* temp = (REAL*)fasp_mem_calloc(A->col, sizeof(REAL));

    iac[0] = 0;

    // First loop:  form sparsity partern of R*A*P
    for (i = 0; i < row; ++i) {
        // reset istart and length at the begining of each loop
        istart = -1;
        length = 0;
        i1     = i + 1;

        // go across the rows in R
        begin_rowR = ir[i];
        end_rowR   = ir[i1];
        for (jj = begin_rowR; jj < end_rowR; ++jj) {
            j = jr[jj];
            // for each column in A
            begin_rowA = ia[j];
            end_rowA   = ia[j + 1];
            for (k = begin_rowA; k < end_rowA; ++k) {
                if (index[ja[k]] == -2) {
                    index[ja[k]] = istart;
                    istart       = ja[k];
                    ++length;
                }
            }
        }

        // book-keeping [reseting length and setting iistart]
        count   = length;
        iistart = -1;
        length  = 0;

        // use each column that would have resulted from R*A
        for (j = 0; j < count; ++j) {
            jj         = istart;
            istart     = index[istart];
            index[jj] = -2;

            // go across the row of P
            begin_row = ip[jj];
            end_row   = ip[jj + 1];
            for (k = begin_row; k < end_row; ++k) {
                // pull out the appropriate columns of P
                if (iindex[jp[k]] == -2) {
                    iindex[jp[k]] = iistart;
                    iistart       = jp[k];
                    ++length;
                }
            } // end for k
        }     // end for j

        // set B->IA
        iac[i1] = iac[i] + length;

        if (iac[i1] > nB) { // Memory not enough!!!
            nB  = nB * 2;
            jac = (INT*)fasp_mem_realloc(jac, nB * sizeof(INT));
        }

        // put the correct columns of p into the column list of the products
        begin_row = iac[i];
        end_row   = iac[i1];
        for (j = begin_row; j < end_row; ++j) {
            // put the value in B->JA
            jac[j] = iistart;
            // set istart to the next value
            iistart = iindex[iistart];
            // set the iindex spot to 0
            iindex[jac[j]] = -2;
        } // end j

    } // end i:  First loop

    jac = (INT*)fasp_mem_realloc(jac, (iac[row]) * sizeof(INT));

    acj = (REAL*)fasp_mem_calloc(iac[row], sizeof(REAL));

    INT* BTindex = (INT*)fasp_mem_calloc(col, sizeof(INT));

    // Second loop:  compute entries of R*A*P
    for (i = 0; i < row; ++i) {
        i1 = i + 1;

        // each col of B
        begin_row = iac[i];
        end_row   = iac[i1];
        for (j = begin_row; j < end_row; ++j) {
            BTindex[jac[j]] = j;
        }

        // reset istart and length at the beginning of each loop
        istart = -1;
        length = 0;

        // go across the rows in R
        begin_rowR = ir[i];
        end_rowR   = ir[i1];
        for (jj = begin_rowR; jj < end_rowR; ++jj) {
            j = jr[jj];

            // for each column in A
            begin_rowA = ia[j];
            end_rowA   = ia[j + 1];
            for (k = begin_rowA; k < end_rowA; ++k) {
                if (index[ja[k]] == -2) {
                    index[ja[k]] = istart;
                    istart       = ja[k];
                    ++length;
                }
                temp[ja[k]] += aj[k];
            }
        }

        // book-keeping [resetting length and setting iistart]
        // use each column that would have resulted from R*A
        for (j = 0; j < length; ++j) {
            jj        = istart;
            istart    = index[istart];
            index[jj] = -2;

            // go across the row of P
            begin_row = ip[jj];
            end_row   = ip[jj + 1];
            for (k = begin_row; k < end_row; ++k) {
                // pull out the appropriate columns of P
                acj[BTindex[jp[k]]] += temp[jj];
            }
            temp[jj] = 0.0;
        }

    } // end for i:  Second loop

    // setup coarse matrix B
    B->row = row;
    B->col = col;
    B->IA  = iac;
    B->JA  = jac;
    B->val = acj;
    B->nnz = B->IA[B->row] - B->IA[0];

    fasp_mem_free(temp);
    temp = NULL;
    fasp_mem_free(index);
    index = NULL;
    fasp_mem_free(iindex);
    iindex = NULL;
    fasp_mem_free(BTindex);
    BTindex = NULL;
```

```
}

void fasp_blas_dcsr_ptap(const dCSRmat* Pt, const dCSRmat* A, const dCSRmat* P,
                         dCSRmat* Ac)
{
    const INT nc = Pt->row, n = Pt->col, nnzP = P->nnz, nnzA = A->nnz;
    INT       i, maxrpout;

    // shift A from usual to ltz format
#ifdef _OPENMP
#pragma omp parallel for if (n > OPENMP_HOLDS)
#endif
    for (i = 0; i <= n; ++i) {
        A->IA[i]++;
        P->IA[i]++;
    }

#ifdef _OPENMP
#pragma omp parallel for if (nnzA > OPENMP_HOLDS)
#endif
    for (i = 0; i < nnzA; ++i) {
        A->JA[i]++;
    }

#ifdef _OPENMP
#pragma omp parallel for if (nc > OPENMP_HOLDS)
#endif
    for (i = 0; i <= nc; ++i) {
        Pt->IA[i]++;
    }

#ifdef _OPENMP
#pragma omp parallel for if (nnzP > OPENMP_HOLDS)
#endif
    for (i = 0; i < nnzP; ++i) {
        P->JA[i]++;
        Pt->JA[i]++;
    }

    // compute P' A P
    dCSRmat PtAP =
        fasp_blas_dcsr_rap2(Pt->IA, Pt->JA, Pt->val, A->IA, A->JA, A->val, Pt->IA,
                            Pt->JA, Pt->val, n, nc, &maxrpout, P->IA, P->JA);

    Ac->row = PtAP.row;
    Ac->col = PtAP.col;
    Ac->nnz = PtAP.nnz;
    Ac->IA  = PtAP.IA;
    Ac->JA  = PtAP.JA;
    Ac->val = PtAP.val;

    // shift A back from ltz format
#ifdef _OPENMP
#pragma omp parallel for if (Ac->row > OPENMP_HOLDS)
#endif
    for (i = 0; i <= Ac->row; ++i) Ac->IA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (Ac->nnz > OPENMP_HOLDS)
#endif
    for (i = 0; i < Ac->nnz; ++i) Ac->JA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (n > OPENMP_HOLDS)
#endif
    for (i = 0; i <= n; ++i) A->IA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (nnzA > OPENMP_HOLDS)
#endif
    for (i = 0; i < nnzA; ++i) A->JA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (n > OPENMP_HOLDS)
#endif
    for (i = 0; i <= n; ++i) P->IA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (nnzP > OPENMP_HOLDS)
#endif
    for (i = 0; i < nnzP; ++i) P->JA[i]--;
```

```

#ifdef _OPENMP
#pragma omp parallel for if (nc > OPENMP_HOLDS)
#endif
    for (i = 0; i <= nc; ++i) Pt->IA[i]--;

#ifdef _OPENMP
#pragma omp parallel for if (nnzP > OPENMP_HOLDS)
#endif
    for (i = 0; i < nnzP; ++i) Pt->JA[i]--;

    return;
}

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)
{
    dCSRmat ac;
    INT     n1, nc1, nnzp, maxrp;
    INT *   ip = NULL, *jp = NULL;

    /*
if ipin is null, this
means that we need to do the transpose of p here; otherwise,
these are considered to be input
*/
    maxrp = 0;
    nnzp  = ipt[nc] - 1;
    n1    = n + 1;

    if (!ipin) {
        ip = (INT*)calloc(n1, sizeof(INT));
        jp = (INT*)calloc(nnzp, sizeof(INT));
        /* these must be null anyway, so no need to assign null
ipin=NULL;
jpin=NULL;
*/
    } else {
        ip = ipin;
        jp = jpin;
    }

    fasp_sparse_iit_(ipt, jpt, &nc, &n, ip, jp);

    /* triple matrix product:  R * A * transpose(P^T)=R*A*P.*/
    /* A is square n by n*/
    /* Note:  to compute R*A* P the input are R, A and P^T */
    /* we need to transpose now the structure of P, because the input is P^T */
    /* end of transpose of the boolean corresponding to P */
    /* ic are the addresses of the rows of the output */
    nc1   = nc + 1;
    ac.IA = (INT*)calloc(nc1, sizeof(INT));

    /*
First call is with jc=null so that we find the number of
nonzeros in the result
*/
    ac.JA = NULL;
    fasp_sparse_rapms_(ir, jr, ia, ja, ip, jp, &n, &nc, ac.IA, ac.JA, &maxrp);
    ac.nnz = ac.IA[nc] - 1;
    ac.JA  = (INT*)calloc(ac.nnz, sizeof(INT));

    /*
second call is to fill the column indexes array jc.
*/
    fasp_sparse_rapms_(ir, jr, ia, ja, ip, jp, &n, &nc, ac.IA, ac.JA, &maxrp);
    if (!ipin) {
        if (ip) free(ip);
        if (jp) free(jp);
    }
    ac.val = (REAL*)calloc(ac.nnz, sizeof(REAL));
    /* this is the compute with the entries */
    fasp_sparse_rapcmp_(ir, jr, r, ia, ja, a, ipt, jpt, pt, &n, &nc, ac.IA, ac.JA,
                        ac.val, &maxrp);
    ac.row = nc;
    ac.col = nc;

    /*=========================================================*/
    *maxrpout = maxrp;

    return ac;
}

void fasp_blas_dcsr_rap4(dCSRmat* R, dCSRmat* A, dCSRmat* P, dCSRmat* B, INT* icor_ysk)
{
    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if (R->row > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads   = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
        const INT row = R->row, col = P->col;
        INT *     ir = R->IA, *ia = A->IA, *ip = P->IA;
        INT *     jr = R->JA, *ja = A->JA, *jp = P->JA;
        REAL *    rj = R->val, *aj = A->val, *pj = P->val;
        INT       istart, iistart;
        INT       end_row, end_rowA, end_rowR;
        INT       i, j, jj, k, length, myid, mybegin, myend;
        INT       jj_counter, ic, jj_row_begining, jj1, i1, jj2, i2, jj3, i3;
        INT*      index   = NULL;
        INT*      iindex  = NULL;
        INT*      BTindex = NULL;
        REAL*     temp    = NULL;
        INT       FiveMyid, min_A, min_P, A_pos, P_pos, FiveIc;
        INT       minus_one_length_A = icor_ysk[5 * nthreads];
        INT       minus_one_length_P = icor_ysk[5 * nthreads + 1];
        INT       minus_one_length   = minus_one_length_A + minus_one_length_P;

        INT* iindexs =
            (INT*)fasp_mem_calloc(minus_one_length + minus_one_length_P, sizeof(INT));

#if DEBUG_MODE > 1
        total_alloc_mem += minus_one_length * sizeof(INT);
#endif
        INT* indexs   = iindexs + minus_one_length_P;
        INT* BTindexs = indexs + minus_one_length_A;

        INT* iac = (INT*)fasp_mem_calloc(row + 1, sizeof(INT));

#if DEBUG_MODE > 1
        total_alloc_mem += (row + 1) * sizeof(INT);
#endif

        INT* part_end = (INT*)fasp_mem_calloc(2 * nthreads + row, sizeof(INT));

#if DEBUG_MODE > 1
        total_alloc_mem += (2 * nthreads + row) * sizeof(INT);
#endif

        INT*  iac_temp     = part_end + nthreads;
        INT** iindex_array = (INT**)fasp_mem_calloc(nthreads, sizeof(INT*));
        INT** index_array  = (INT**)fasp_mem_calloc(nthreads, sizeof(INT*));

        fasp_iarray_set(minus_one_length, iindexs, -2);

#ifdef _OPENMP
#pragma omp parallel for private(myid, FiveMyid, mybegin, myend, min_A, min_P, index,  \
iindex, A_pos, P_pos, ic, FiveIc, jj_counter,         \
jj_row_begining, end_rowR, jj1, i1, end_rowA, jj2,    \
i2, end_row, jj3, i3)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            FiveMyid = myid * 5;
            mybegin  = icor_ysk[FiveMyid];
            if (myid == nthreads - 1) {
                myend = row;
            } else {
                myend = icor_ysk[FiveMyid + 5];
            }
            min_A = icor_ysk[FiveMyid + 2];
            min_P = icor_ysk[FiveMyid + 4];
            A_pos = 0;
            P_pos = 0;
            for (ic = myid - 1; ic >= 0; ic--) {
                FiveIc = ic * 5;
                A_pos += icor_ysk[FiveIc + 1];
                P_pos += icor_ysk[FiveIc + 3];
            }
            iindex_array[myid] = iindex = iindexs + P_pos - min_P;
            index_array[myid] = index = indexs + A_pos - min_A;
            jj_counter               = 0;
            for (ic = mybegin; ic < myend; ic++) {
                iindex[ic]      = jj_counter;
                jj_row_begining = jj_counter;
                jj_counter++;
                end_rowR = ir[ic + 1];
                for (jj1 = ir[ic]; jj1 < end_rowR; jj1++) {
                    i1       = jr[jj1];
                    end_rowA = ia[i1 + 1];
                    for (jj2 = ia[i1]; jj2 < end_rowA; jj2++) {
                        i2 = ja[jj2];
                        if (index[i2] != ic) {
                            index[i2] = ic;
                            end_row   = ip[i2 + 1];
                            for (jj3 = ip[i2]; jj3 < end_row; jj3++) {
                                i3 = jp[jj3];
                                if (iindex[i3] < jj_row_begining) {
                                    iindex[i3] = jj_counter;
                                    jj_counter++;
                                }
                            }
                        }
                    }
                }
                iac_temp[ic + myid] = jj_row_begining;
            }
            iac_temp[myend + myid] = jj_counter;
            part_end[myid]         = myend + myid + 1;
        }
        fasp_iarray_cp(part_end[0], iac_temp, iac);
        jj_counter = part_end[0];
        INT Ctemp  = 0;
        for (i1 = 1; i1 < nthreads; i1++) {
            Ctemp += iac_temp[part_end[i1 - 1] - 1];
            for (jj1 = part_end[i1 - 1] + 1; jj1 < part_end[i1]; jj1++) {
                iac[jj_counter] = iac_temp[jj1] + Ctemp;
                jj_counter++;
            }
        }
        INT* jac = (INT*)fasp_mem_calloc(iac[row], sizeof(INT));
#if DEBUG_MODE > 1
        total_alloc_mem += iac[row] * sizeof(INT);
#endif
        fasp_iarray_set(minus_one_length, iindexs, -2);
#ifdef _OPENMP
#pragma omp parallel for private(myid, index, iindex, FiveMyid, mybegin, myend, i,      \
istart, length, i1, end_rowR, jj, j, end_rowA, k,      \
iistart, end_row)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            iindex   = iindex_array[myid];
            index    = index_array[myid];
            FiveMyid = myid * 5;
            mybegin  = icor_ysk[FiveMyid];
            if (myid == nthreads - 1) {
                myend = row;
            } else {
                myend = icor_ysk[FiveMyid + 5];
            }
            for (i = mybegin; i < myend; ++i) {
                istart = -1;
                length = 0;
                i1     = i + 1;
                // go across the rows in R
                end_rowR = ir[i1];
                for (jj = ir[i]; jj < end_rowR; ++jj) {
                    j = jr[jj];
                    // for each column in A
                    end_rowA = ia[j + 1];
                    for (k = ia[j]; k < end_rowA; ++k) {
                        if (index[ja[k]] == -2) {
                            index[ja[k]] = istart;
                            istart       = ja[k];
                            ++length;
                        }
                    }
                }
                // book-keeping [resetting length and setting iistart]
                // count = length;
                iistart = -1;
                // length = 0;
                //  use each column that would have resulted from R*A
                // for (j = 0; j < count; ++ j) {
                for (j = 0; j < length; ++j) {
                    jj        = istart;
                    istart    = index[istart];
                    index[jj] = -2;
                    // go across the row of P
                    end_row = ip[jj + 1];
                    for (k = ip[jj]; k < end_row; ++k) {
                        // pull out the appropriate columns of P
                        if (iindex[jp[k]] == -2) {
                            iindex[jp[k]] = iistart;
                            iistart       = jp[k];
                            //++length;
                        }
                    } // end for k
                }     // end for j
                // put the correct columns of p into the column list of the products
                end_row = iac[i1];
                for (j = iac[i]; j < end_row; ++j) {
                    // put the value in B->JA
                    jac[j] = iistart;
                    // set istart to the next value
                    iistart = iindex[iistart];
                    // set the iindex spot to 0
                    iindex[jac[j]] = -2;
                } // end j
            }
        }
        // Third loop:  compute entries of R*A*P
        REAL* acj = (REAL*)fasp_mem_calloc(iac[row], sizeof(REAL));
#if DEBUG_MODE > 1
        total_alloc_mem += iac[row] * sizeof(REAL);
#endif
        REAL* temps = (REAL*)fasp_mem_calloc(minus_one_length_A, sizeof(REAL));
#if DEBUG_MODE > 1
        total_alloc_mem += minus_one_length_A * sizeof(REAL);
#endif

#ifdef _OPENMP
#pragma omp parallel for private(                                               \
myid, index, FiveMyid, mybegin, myend, min_A, min_P, A_pos, P_pos, ic, FiveIc,     \
BTindex, temp, i, i1, end_row, j, istart, length, end_rowR, jj, end_rowA, k)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            index    = index_array[myid];
            FiveMyid = myid * 5;
            mybegin  = icor_ysk[FiveMyid];
            if (myid == nthreads - 1) {
                myend = row;
            } else {
                myend = icor_ysk[FiveMyid + 5];
            }
            min_A = icor_ysk[FiveMyid + 2];
            min_P = icor_ysk[FiveMyid + 4];
            A_pos = 0;
            P_pos = 0;
            for (ic = myid - 1; ic >= 0; ic--) {
                FiveIc = ic * 5;
                A_pos += icor_ysk[FiveIc + 1];
                P_pos += icor_ysk[FiveIc + 3];
            }
            BTindex = BTindexs + P_pos - min_P;
            temp    = temps + A_pos - min_A;
            for (i = mybegin; i < myend; ++i) {
                i1 = i + 1;
                // each col of B
                end_row = iac[i1];
                for (j = iac[i]; j < end_row; ++j) {
                    BTindex[jac[j]] = j;
                }
                // reset istart and length at the beginning of each loop
                istart = -1;
                length = 0;
                // go across the rows in R
                end_rowR = ir[i1];
                for (jj = ir[i]; jj < end_rowR; ++jj) {
                    j = jr[jj];
                    // for each column in A
                    end_rowA = ia[j + 1];
                    for (k = ia[j]; k < end_rowA; ++k) {
                        if (index[ja[k]] == -2) {
                            index[ja[k]] = istart;
                            istart       = ja[k];
                            ++length;
                        }
                        temp[ja[k]] += rj[jj] * aj[k];
                    }
                }
                // book-keeping [resetting length and setting iistart]
                // use each column that would have resulted from R*A
                for (j = 0; j < length; ++j) {
                    jj        = istart;
                    istart    = index[istart];
                    index[jj] = -2;
                    // go across the row of P
                    end_row = ip[jj + 1];
                    for (k = ip[jj]; k < end_row; ++k) {
                        // pull out the appropriate columns of P
                        acj[BTindex[jp[k]]] += temp[jj] * pj[k];
                    }
                    temp[jj] = 0.0;
                }
            }
        }
        // setup coarse matrix B
        B->row = row;
        B->col = col;
        B->IA  = iac;
        B->JA  = jac;
        B->val = acj;
        B->nnz = B->IA[B->row] - B->IA[0];

        fasp_mem_free(temps);
        temps = NULL;
        fasp_mem_free(iindexs);
        iindexs = NULL;
        fasp_mem_free(part_end);
        part_end = NULL;
        fasp_mem_free(iindex_array);
        iindex_array = NULL;
        fasp_mem_free(index_array);
        index_array = NULL;
    } else {
        fasp_blas_dcsr_rap(R, A, P, B);
    }
}

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

9.93 BlaSpmvCSRL.c File Reference

Linear algebraic operations for dCSRLmat matrices.

```
#include "fasp.h"
```

Functions

- void fasp_blas_dcsrl_mxv (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.

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∗x for a sparse matrix in CSRL format.

Parameters

A	Pointer to dCSRLmat matrix A
x	Pointer to REAL array of vector x
y	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.

```
#include "fasp.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_dcsrl_mxv (const dCSRLmat  *A,
                          const REAL      *x,
                          REAL            *y)
{
    const INT    dif       = A -> dif;
    const INT   *nz_diff   = A -> nz_diff;
    const INT   *rowindex  = A -> index;
    const INT   *rowstart  = A -> start;
    const INT   *ja        = A -> ja;
    const REAL  *a         = A -> val;

    INT i;
    INT row, col=0;
    INT len, rowlen;
    INT firstrow, lastrow;

    REAL val0, val1;

    for (len = 0; len < dif; len ++) {
        firstrow = rowstart[len];
        lastrow  = rowstart[len+1] - 1;
        rowlen   = nz_diff[len];

        if (lastrow > firstrow ) {
            //----------------------------------------------------------
            // Fully-unrolled code for special case (i.g.,rowlen = 5)
            // Note:  you can also set other special case
            //----------------------------------------------------------
            if (rowlen == 5) {
                for (row = firstrow; row < lastrow; row += 2) {
                    val0 = a[col]*x[ja[col]];
                    val1 = a[col+5]*x[ja[col+5]];
                    col ++;

                    val0 += a[col]*x[ja[col]];
                    val1 += a[col+5]*x[ja[col+5]];
                    col ++;

                    val0 += a[col]*x[ja[col]];
                    val1 += a[col+5]*x[ja[col+5]];
                    col ++;

                    val0 += a[col]*x[ja[col]];
                    val1 += a[col+5]*x[ja[col+5]];
                    col ++;

                    val0 += a[col]*x[ja[col]];
                    val1 += a[col+5]*x[ja[col+5]];
                    col ++;

                    y[rowindex[row]] = val0;
                    y[rowindex[row+1]] = val1;

                    col += 5;
                }
            }
            else {
                //------------------------------------------------------------------
                // Unroll-and-jammed code for handling two rows at a time
                //------------------------------------------------------------------

                for (row = firstrow; row < lastrow; row += 2) {
                    val0 = 0.0;
                    val1 = 0.0;
                    for (i = 0; i < rowlen; i ++) {
                        val0 += a[col]*x[ja[col]];
                        val1 += a[col+rowlen]*x[ja[col+rowlen]];
                        col ++;
                    }
                    y[rowindex[row]] = val0;
                    y[rowindex[row+1]] = val1;
                    col += rowlen;
                }
            }
            firstrow = row;
        }

        //------------------------------------------------------------
        // Handle leftover rows that can't be handled in bundles
        // in the unroll-and -jammed loop
        //------------------------------------------------------------

        for (row = firstrow; row <= lastrow; row ++) {
            val0 = 0.0;
            for (i = 0; i < rowlen; i ++) {
                val0 += a[col]*x[ja[col]];
                col ++;
            }
            y[rowindex[row]] = val0;
        }

    }
```

```

}

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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_aAxpy (const REAL alpha, const dSTRmat ∗A, const REAL ∗x, REAL ∗y)

  Matrix-vector multiplication y = alpha∗A∗x + y.
- void fasp_blas_dstr_mxv (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

Definition in file BlaSpmvSTR.c.

9.95.2 Function Documentation

9.95.2.1 fasp_blas_dstr_aAxpy()

```
void fasp_blas_dstr_aAxpy (
              const REAL alpha,
              const dSTRmat * A,
              const REAL * x,
              REAL * y )
```

Matrix-vector multiplication y = alpha∗A∗x + y.

Parameters

alpha	REAL factor alpha
A	Pointer to dSTRmat matrix

Parameters

x	Pointer to REAL array
y	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

A	Pointer to a 'dSTRmat' type matrix A
B	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_mxv()

```
void fasp_blas_dstr_mxv (
             const dSTRmat * A,
             const REAL * x,
             REAL * y )
```

Matrix-vector multiplication y = A∗x.

Parameters

A	Pointer to dSTRmat matrix
x	Pointer to REAL array
y	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.

```c

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static inline void smat_amxv_nc3(const REAL, const REAL *, const REAL *, REAL *);
static inline void smat_amxv_nc5(const REAL, const REAL *, const REAL *, REAL *);
static inline void smat_amxv(const REAL, const REAL *, const REAL *, const INT, REAL *);
static inline void str_spaAxpy_2D_nc1(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_2D_nc3(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_2D_nc5(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_2D_blk(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_3D_nc1(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_3D_nc3(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_3D_nc5(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy_3D_blk(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void str_spaAxpy(const REAL, const dSTRmat *, const REAL *, REAL *);
static inline void blkcontr_str(const INT, const INT, const INT, const INT,
                                const REAL *, const REAL *, REAL *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_dstr_aAxpy (const REAL      alpha,
                           const dSTRmat  *A,
                           const REAL     *x,
                           REAL           *y)
{

    switch (A->nband) {

        case 4:

            switch (A->nc) {
                case 1:
                    str_spaAxpy_2D_nc1(alpha, A, x, y);
                    break;

                case 3:
                    str_spaAxpy_2D_nc3(alpha, A, x, y);
                    break;

                case 5:
                    str_spaAxpy_2D_nc5(alpha, A, x, y);
                    break;

                default:
                    str_spaAxpy_2D_blk(alpha, A, x, y);
                    break;
            }

            break;

        case 6:

            switch (A->nc) {
                case 1:
                    str_spaAxpy_3D_nc1(alpha, A, x, y);
                    break;

                case 3:
                    str_spaAxpy_3D_nc3(alpha, A, x, y);
                    break;

                case 5:
                    str_spaAxpy_3D_nc5(alpha, A, x, y);
                    break;

                default:
                    str_spaAxpy_3D_blk(alpha, A, x, y);
                    break;
            }
            break;

        default:
            str_spaAxpy(alpha, A, x, y);
            break;
    }

}

void fasp_blas_dstr_mxv (const dSTRmat  *A,
                         const REAL     *x,
                         REAL           *y)
{
    int n = (A->ngrid)*(A->nc)*(A->nc);

    memset(y, 0, n*sizeof(REAL));

    fasp_blas_dstr_aAxpy(1.0, A, x, y);
}

INT fasp_blas_dstr_diagscale (const dSTRmat  *A,
                              dSTRmat        *B)
{
    const INT ngrid=A->ngrid, nc=A->nc, nband=A->nband;
    const INT nc2=nc*nc, size=ngrid*nc2;
    INT i,j,ic2,nb,nb1;

#ifdef _OPENMP
    //variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    REAL *diag=(REAL *)fasp_mem_calloc(size,sizeof(REAL));

    fasp_darray_cp(size,A->diag,diag);

    fasp_dstr_alloc(A->nx, A->ny, A->nz,A->nxy,ngrid, nband,nc,A->offsets, B);

    //compute diagnal elements of B
#ifdef _OPENMP
    if (ngrid > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, ic2, j)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ngrid, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                ic2=i*nc2;
                for (j=0; j<nc2; j++) {
                    if (j/nc == j%nc) B->diag[ic2+j]=1;
                    else B->diag[ic2+j]=0;
                }
            }
        }
    }
    else {
#endif
        for (i=0;i<ngrid;++i) {
            ic2=i*nc2;
            for (j=0;j<nc2;++j) {
                if (j/nc == j%nc) B->diag[ic2+j]=1;
                else B->diag[ic2+j]=0;
            }
        }
#ifdef _OPENMP
    }
#endif

    for (i=0;i<ngrid;++i) fasp_smat_inv(&(diag[i*nc2]),nc);

    for (i=0;i<nband;++i) {
        nb=A->offsets[i];
        nb1=abs(nb);
        if (nb<0) {
            for (j=0;j<ngrid-nb1;++j)
fasp_blas_smat_mul(&(diag[(j+nb1)*nc2]),&(A->offdiag[i][j*nc2]),&(B->offdiag[i][j*nc2]),nc);
        }
        else {
            for (j=0;j<ngrid-nb1;++j)
                fasp_blas_smat_mul(&(diag[j*nc2]),&(A->offdiag[i][j*nc2]),&(B->offdiag[i][j*nc2]),nc);
        }

    }

    fasp_mem_free(diag); diag = NULL;

    return (0);
}

/*----------------------------------*/
/*--      Private Functions       --*/
/*----------------------------------*/

static inline void smat_amxv_nc3 (const REAL   alpha,
                                  const REAL  *a,
                                  const REAL  *b,
                                  REAL        *c)
{
    c[0] += alpha*(a[0]*b[0] + a[1]*b[1] + a[2]*b[2]);
    c[1] += alpha*(a[3]*b[0] + a[4]*b[1] + a[5]*b[2]);
    c[2] += alpha*(a[6]*b[0] + a[7]*b[1] + a[8]*b[2]);
}

static inline void smat_amxv_nc5 (const REAL   alpha,
                                  const REAL  *a,
                                  const REAL  *b,
                                  REAL        *c)
{
    c[0] += alpha*(a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + a[3] * b[3] + a[4] * b[4]);
    c[1] += alpha*(a[5]*b[0] + a[6]*b[1] + a[7]*b[2] + a[8] * b[3] + a[9] * b[4]);
    c[2] += alpha*(a[10]*b[0] + a[11]*b[1] + a[12]*b[2] + a[13] * b[3] + a[14] * b[4]);
    c[3] += alpha*(a[15]*b[0] + a[16]*b[1] + a[17]*b[2] + a[18] * b[3] + a[19] * b[4]);
    c[4] += alpha*(a[20]*b[0] + a[21]*b[1] + a[22]*b[2] + a[23] * b[3] + a[24] * b[4]);
}

static inline void smat_amxv (const REAL   alpha,
                              const REAL  *a,
                              const REAL  *b,
                              const INT    n,
                              REAL        *c)
{
    INT i,j;
    INT in;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

#ifdef _OPENMP
    if (n > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, in, j)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                in = i*n;
                for (j=0; j<n; j++)
                    c[i] += alpha*a[in+j]*b[j];
            }
        }
    }
    else {
```

```
#endif
        for (i=0;i<n;++i) {
            in = i*n;
            for (j=0;j<n;++j)
                c[i] += alpha*a[in+j]*b[j];
        }
#ifdef _OPENMP
    }
#endif
    return;
}

static inline void blkcontr_str (const INT   start_data,
                                 const INT   start_vecx,
                                 const INT   start_vecy,
                                 const INT   nc,
                                 const REAL *data,
                                 const REAL *x,
                                 REAL *y)
{
    INT i,j,k,m;

#ifdef _OPENMP
    //variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

#ifdef _OPENMP
    if (nc > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, k, m, j)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, nc, &mybegin, &myend);
            for (i = mybegin; i < myend; i ++) {
                k = start_data + i*nc;
                m = start_vecy + i;
                for (j = 0; j < nc; j ++) {
                    y[m] += data[k+j]*x[start_vecx+j];
                }
            }
        }
    }
    else {
#endif
        for (i = 0; i < nc; i ++) {
            k = start_data + i*nc;
            m = start_vecy + i;
            for (j = 0; j < nc; j ++) {
                y[m] += data[k+j]*x[start_vecx+j];
            }
        }
#ifdef _OPENMP
    }
#endif
}

static inline void str_spaAxpy_2D_nc1 (const REAL     alpha,
                                       const dSTRmat *A,
                                       const REAL    *x,
                                       REAL          *y)
{
    INT i;
    INT idx1, idx2;
    INT end1, end2;
    INT nline;

#ifdef _OPENMP
    //variables for OpenMP
    INT myid, mybegin, myend, idx;
    INT nthreads = fasp_get_num_threads();
#endif

    // information of A
    INT nx = A->nx;
    INT ngrid = A->ngrid;  // number of grids
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;

    if (nx == 1) {
        nline = A->ny;
    }
    else {
        nline = nx;
    }

    for (i=0; i<nband; ++i) {
        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nline) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nline) {
            offdiag3 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nline;

    y[0] += alpha*(diag[0]*x[0] + offdiag1[0]*x[1] + offdiag3[0]*x[nline]);

#ifdef _OPENMP
    if (nline-1 > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, idx1, idx)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, nline-1, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx1 = i;
                idx  = i+1;
                y[idx] += alpha*(offdiag0[idx1]*x[idx1] + diag[idx]*x[idx] +
                                 offdiag1[idx]*x[idx+1] + offdiag3[idx]*x[idx+nline]);
            }
        }
    }
    else {
#endif
        for (i=1; i<nline; ++i) {
            idx1 = i-1;
            y[i] += alpha*(offdiag0[idx1]*x[idx1] + diag[i]*x[i] +
                           offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nline]);
        }
#ifdef _OPENMP
    }
#endif

#ifdef _OPENMP
    if (end2-nline > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, i, mybegin, myend, idx1, idx2, idx)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, end2-nline, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
                idx  = i+nline;
                idx1 = idx-1; //idx1 = i-1+nline;
                idx2 = i;
                y[idx] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
                                 diag[idx]*x[idx] + offdiag1[idx]*x[idx+1] +
                                 offdiag3[idx]*x[idx+nline]);
            }
        }
    }
    else {
#endif
        for (i=nline; i<end2; ++i) {
            idx1 = i-1;
            idx2 = i-nline;
            y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
                           diag[i]*x[i] + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nline]);
        }
#ifdef _OPENMP
    }
#endif
```

```

#ifdef _OPENMP
    if (end1-end2 > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, i, mybegin, myend, idx1, idx2, idx)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, end1-end2, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
                idx  = i+end2;
                idx1 = idx-1;     //idx1 = i-1+end2;
                idx2 = idx-nline; //idx2 = i-nline+end2;
                y[idx] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
                                 diag[idx]*x[idx] + offdiag1[idx]*x[idx+1]);
            }
        }
    }
    else {
#endif
        for (i=end2; i<end1; ++i) {
            idx1 = i-1;
            idx2 = i-nline;
            y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] +
                           diag[i]*x[i] + offdiag1[i]*x[i+1]);
        }
#ifdef _OPENMP
    }
#endif

    idx1 = end1-1;
    idx2 = end1-nline;
    y[end1] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1] + diag[end1]*x[end1]);

    return;

}

static inline void str_spaAxpy_2D_nc3 (const REAL     alpha,
                                       const dSTRmat *A,
                                       const REAL    *x,
                                       REAL          *y)
{
    INT i;
    INT idx,idx1,idx2;
    INT matidx, matidx1, matidx2;
    INT end1, end2;
    INT nline, nlinenc;

    // information of A
    INT nx = A->nx;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nband = A->nband;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend, up;
    INT nthreads = fasp_get_num_threads();
#endif

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;

    if (nx == 1) {
        nline = A->ny;
    }
    else {
        nline = nx;
    }
    nlinenc = nline*nc;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nline) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nline) {
            offdiag3 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }

    }

    end1 = ngrid-1;
    end2 = ngrid-nline;

    smat_amxv_nc3(alpha, diag, x, y);
    smat_amxv_nc3(alpha, offdiag1, x+nc, y);
    smat_amxv_nc3(alpha, offdiag3, x+nlinenc, y);

#ifdef _OPENMP
    up = nline - 1;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, idx, matidx, idx1, matidx1)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+1)*nc;
                matidx = idx*nc;
                idx1 = i*nc;
                matidx1 = idx1*nc;
                smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
                smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
                smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=1; i<nline; ++i) {
            idx = i*nc;
            matidx = idx*nc;
            idx1 = idx - nc;
            matidx1 = idx1*nc;
            smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
        }
#ifdef _OPENMP
    }
#endif

#ifdef _OPENMP
    up = end2 - nx;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+nx)*nc;
                idx1 = idx-nc;
                idx2 = idx-nlinenc;
                matidx = idx*nc;
                matidx1 = idx1*nc;
                matidx2 = idx2*nc;
                smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
                smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc3(alpha, diag+matidx,      x+idx,  y+idx);
                smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
                smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=nx; i<end2; ++i) {
            idx = i*nc;
            idx1 = idx-nc;
            idx2 = idx-nlinenc;
            matidx = idx*nc;
            matidx1 = idx1*nc;
            matidx2 = idx2*nc;
            smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
            smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
        }
#ifdef _OPENMP
    }
#endif

#ifdef _OPENMP
    up = end1 - end2;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+end2)*nc;
                idx1 = idx-nc;
                idx2 = idx-nlinenc;
                matidx = idx*nc;
                matidx1 = idx1*nc;
                matidx2 = idx2*nc;
                smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
                smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
                smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=end2; i<end1; ++i) {
            idx = i*nc;
            idx1 = idx-nc;
            idx2 = idx-nlinenc;
            matidx = idx*nc;
            matidx1 = idx1*nc;
            matidx2 = idx2*nc;
            smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
            smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        }
#ifdef _OPENMP
    }
#endif
    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nlinenc;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
    smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
    smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);

    return;
}

static inline void str_spaAxpy_2D_nc5 (const REAL      alpha,
                                       const dSTRmat  *A,
                                       const REAL     *x,
                                       REAL           *y)
{
    INT i;
    INT idx,idx1,idx2;
    INT matidx, matidx1, matidx2;
    INT end1, end2;
    INT nline, nlinenc;

    // information of A
    INT nx = A->nx;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nband = A->nband;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend, up;
    INT nthreads = fasp_get_num_threads();
#endif

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;

    if (nx == 1) {
        nline = A->ny;
    }
    else {
        nline = nx;
    }
    nlinenc = nline*nc;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nline) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nline) {
            offdiag3 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nline;

    smat_amxv_nc5(alpha, diag, x, y);
    smat_amxv_nc5(alpha, offdiag1, x+nc, y);
    smat_amxv_nc5(alpha, offdiag3, x+nlinenc, y);

#ifdef _OPENMP
    up = nline - 1;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, idx, matidx, idx1, matidx1)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+1)*nc;
                matidx = idx*nc;
                idx1 = i*nc; // idx1 = idx - nc;
                matidx1 = idx1*nc;
                smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
                smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
                smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=1; i<nline; ++i) {
            idx = i*nc;
            matidx = idx*nc;
            idx1 = idx - nc;
            matidx1 = idx1*nc;
            smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
        }
#ifdef _OPENMP
    }
#endif

#ifdef _OPENMP
    up = end2 - nx;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+nx)*nc;
                idx1 = idx-nc;
                idx2 = idx-nlinenc;
                matidx = idx*nc;
                matidx1 = idx1*nc;
                matidx2 = idx2*nc;
                smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
                smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc5(alpha, diag+matidx,      x+idx,   y+idx);
                smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
                smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=nx; i<end2; ++i) {
            idx = i*nc;
            idx1 = idx-nc;
            idx2 = idx-nlinenc;
            matidx = idx*nc;
            matidx1 = idx1*nc;
            matidx2 = idx2*nc;
            smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
            smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nlinenc, y+idx);
        }
#ifdef _OPENMP
    }
#endif

#ifdef _OPENMP
    up = end1 - end2;
    if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, idx, idx1, idx2, matidx, matidx1, matidx2)
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                idx = (i+end2)*nc;
                idx1 = idx-nc;
                idx2 = idx-nlinenc;
                matidx = idx*nc;
                matidx1 = idx1*nc;
                matidx2 = idx2*nc;
                smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
                smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
                smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
                smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
            }
        }
    }
    else {
#endif
        for (i=end2; i<end1; ++i) {
            idx = i*nc;
            idx1 = idx-nc;
            idx2 = idx-nlinenc;
            matidx = idx*nc;
            matidx1 = idx1*nc;
            matidx2 = idx2*nc;
            smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
            smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
            smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
            smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        }
#ifdef _OPENMP
    }
#endif

    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nlinenc;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
    smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
    smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);

    return;

}

static inline void str_spaAxpy_2D_blk (const REAL      alpha,
                                       const dSTRmat  *A,
                                       const REAL     *x,
                                       REAL           *y)
{
    INT i;
    INT idx,idx1,idx2;
    INT matidx, matidx1, matidx2;
    INT end1, end2;
    INT nline, nlinenc;

    // information of A
    INT nx = A->nx;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL, *offdiag3=NULL;

    if (nx == 1) {
        nline = A->ny;
    }
    else {
        nline = nx;
    }
    nlinenc = nline*nc;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nline) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nline) {
            offdiag3 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }

    }

    end1 = ngrid-1;
    end2 = ngrid-nline;

    smat_amxv(alpha, diag, x, nc, y);
    smat_amxv(alpha, offdiag1, x+nc, nc, y);
    smat_amxv(alpha, offdiag3, x+nlinenc, nc, y);

    for (i=1; i<nline; ++i) {
        idx = i*nc;
        matidx = idx*nc;
        idx1 = idx - nc;
        matidx1 = idx1*nc;
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nlinenc, nc, y+idx);
    }

    for (i=nx; i<end2; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nlinenc;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nlinenc, nc, y+idx);
    }

    for (i=end2; i<end1; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nlinenc;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
    }

    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nlinenc;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
    smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
    smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);

    return;
}

static inline void str_spaAxpy_3D_nc1 (const REAL      alpha,
                                       const dSTRmat   *A,
                                       const REAL      *x,
                                       REAL            *y)
{
    INT i;
    INT idx1,idx2,idx3;
    INT end1, end2, end3;
    // information of A
    INT nx = A->nx;
    INT nxy = A->nxy;
    INT ngrid = A->ngrid;  // number of grids
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
    *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nx) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nx) {
            offdiag3 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nxy) {
            offdiag4 = A->offdiag[i];
        }
        else if (A->offsets[i] == nxy) {
            offdiag5 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 3D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nx;
    end3 = ngrid-nxy;

    y[0] += alpha*(diag[0]*x[0] + offdiag1[0]*x[1] + offdiag3[0]*x[nx] + offdiag5[0]*x[nxy]);

    for (i=1; i<nx; ++i) {
        idx1 = i-1;
        y[i] += alpha*(offdiag0[idx1]*x[idx1] + diag[i]*x[i] + offdiag1[i]*x[i+1] +
                       offdiag3[i]*x[i+nx] + offdiag5[i]*x[i+nxy]);
    }

    for (i=nx; i<nxy; ++i) {
        idx1 = i-1;
        idx2 = i-nx;
        y[i] += alpha*(offdiag2[idx2]*x[idx2] + offdiag0[idx1]*x[idx1]
                       + diag[i]*x[i] + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nx]
                       + offdiag5[i]*x[i+nxy]);
    }

    for (i=nxy; i<end3; ++i) {
        idx1 = i-1;
        idx2 = i-nx;
        idx3 = i-nxy;
        y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
                       + offdiag0[idx1]*x[idx1] + diag[i]*x[i] + offdiag1[i]*x[i+1]
                       + offdiag3[i]*x[i+nx] + offdiag5[i]*x[i+nxy]);
    }

    for (i=end3; i<end2; ++i) {
        idx1 = i-1;
        idx2 = i-nx;
        idx3 = i-nxy;
        y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
                       + offdiag0[idx1]*x[idx1] + diag[i]*x[i]
                       + offdiag1[i]*x[i+1] + offdiag3[i]*x[i+nx]);
    }

    for (i=end2; i<end1; ++i) {
        idx1 = i-1;
        idx2 = i-nx;
        idx3 = i-nxy;
        y[i] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2]
                       + offdiag0[idx1]*x[idx1] + diag[i]*x[i]
                       + offdiag1[i]*x[i+1]);
    }

    idx1 = end1-1;
    idx2 = end1-nx;
    idx3 = end1-nxy;
    y[end1] += alpha*(offdiag4[idx3]*x[idx3] + offdiag2[idx2]*x[idx2] +
                      offdiag0[idx1]*x[idx1] + diag[end1]*x[end1]);

    return;
}

static inline void str_spaAxpy_3D_nc3 (const REAL      alpha,
                                       const dSTRmat   *A,
                                       const REAL      *x,
                                       REAL            *y)
{
    INT i;
    INT idx,idx1,idx2,idx3;
    INT matidx, matidx1, matidx2, matidx3;
    INT end1, end2, end3;
    // information of A
    INT nx = A->nx;
    INT nxy = A->nxy;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nxnc = nx*nc;
    INT nxync = nxy*nc;
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
    *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nx) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nx) {
            offdiag3 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nxy) {
            offdiag4 = A->offdiag[i];
        }
        else if (A->offsets[i] == nxy) {
            offdiag5 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nx;
    end3 = ngrid-nxy;

    smat_amxv_nc3(alpha, diag, x, y);
    smat_amxv_nc3(alpha, offdiag1, x+nc, y);
    smat_amxv_nc3(alpha, offdiag3, x+nxnc, y);
    smat_amxv_nc3(alpha, offdiag5, x+nxync, y);

    for (i=1; i<nx; ++i) {
        idx = i*nc;
        matidx = idx*nc;
        idx1 = idx - nc;
        matidx1 = idx1*nc;
        smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
    }

    for (i=nx; i<nxy; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);

    }

    for (i=nxy; i<end3; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc3(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
    }

    for (i=end3; i<end2; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc3(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
    }

    for (i=end2; i<end1; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc3(alpha, offdiag1+matidx, x+idx+nc, y+idx);
    }

    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nxnc;
    idx3 = idx-nxync;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    matidx3 = idx3*nc;
    smat_amxv_nc3(alpha, offdiag4+matidx3, x+idx3, y+idx);
    smat_amxv_nc3(alpha, offdiag2+matidx2, x+idx2, y+idx);
    smat_amxv_nc3(alpha, offdiag0+matidx1, x+idx1, y+idx);
    smat_amxv_nc3(alpha, diag+matidx, x+idx, y+idx);

    return;

}

static inline void str_spaAxpy_3D_nc5 (const REAL      alpha,
                                       const dSTRmat  *A,
                                       const REAL     *x,
                                       REAL           *y)
{
    INT i;
    INT idx,idx1,idx2,idx3;
    INT matidx, matidx1, matidx2, matidx3;
    INT end1, end2, end3;
    // information of A
    INT nx = A->nx;
    INT nxy = A->nxy;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nxnc = nx*nc;
    INT nxync = nxy*nc;
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
    *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nx) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nx) {
            offdiag3 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nxy) {
            offdiag4 = A->offdiag[i];
        }
        else if (A->offsets[i] == nxy) {
            offdiag5 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nx;
    end3 = ngrid-nxy;

    smat_amxv_nc5(alpha, diag, x, y);
    smat_amxv_nc5(alpha, offdiag1, x+nc, y);
    smat_amxv_nc5(alpha, offdiag3, x+nxnc, y);
    smat_amxv_nc5(alpha, offdiag5, x+nxync, y);

    for (i=1; i<nx; ++i) {
        idx = i*nc;
        matidx = idx*nc;
        idx1 = idx - nc;
        matidx1 = idx1*nc;
        smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
    }

    for (i=nx; i<nxy; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);

    }

    for (i=nxy; i<end3; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
        smat_amxv_nc5(alpha, offdiag5+matidx, x+idx+nxync, y+idx);
    }

    for (i=end3; i<end2; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
        smat_amxv_nc5(alpha, offdiag3+matidx, x+idx+nxnc, y+idx);
    }

    for (i=end2; i<end1; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
        smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
        smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
        smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);
        smat_amxv_nc5(alpha, offdiag1+matidx, x+idx+nc, y+idx);
    }

    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nxnc;
    idx3 = idx-nxync;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    matidx3 = idx3*nc;
    smat_amxv_nc5(alpha, offdiag4+matidx3, x+idx3, y+idx);
    smat_amxv_nc5(alpha, offdiag2+matidx2, x+idx2, y+idx);
    smat_amxv_nc5(alpha, offdiag0+matidx1, x+idx1, y+idx);
    smat_amxv_nc5(alpha, diag+matidx, x+idx, y+idx);

    return;

}

static inline void str_spaAxpy_3D_blk (const REAL      alpha,
                                       const dSTRmat  *A,
                                       const REAL     *x,
                                       REAL           *y)
{
    INT i;
    INT idx,idx1,idx2,idx3;
    INT matidx, matidx1, matidx2, matidx3;
    INT end1, end2, end3;
    // information of A
    INT nx = A->nx;
    INT nxy = A->nxy;
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;
    INT nxnc = nx*nc;
    INT nxync = nxy*nc;
    INT nband = A->nband;

    REAL *diag = A->diag;
    REAL *offdiag0=NULL, *offdiag1=NULL, *offdiag2=NULL,
    *offdiag3=NULL, *offdiag4=NULL, *offdiag5=NULL;

    for (i=0; i<nband; ++i) {

        if (A->offsets[i] == -1) {
            offdiag0 = A->offdiag[i];
        }
        else if (A->offsets[i] == 1) {
            offdiag1 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nx) {
            offdiag2 = A->offdiag[i];
        }
        else if (A->offsets[i] == nx) {
            offdiag3 = A->offdiag[i];
        }
        else if (A->offsets[i] == -nxy) {
            offdiag4 = A->offdiag[i];
        }
        else if (A->offsets[i] == nxy) {
            offdiag5 = A->offdiag[i];
        }
        else {
            printf("### WARNING: offsets for 2D scalar is illegal!  %s\n", __FUNCTION__);
            str_spaAxpy(alpha, A, x, y);
            return;
        }
    }

    end1 = ngrid-1;
    end2 = ngrid-nx;
    end3 = ngrid-nxy;

    smat_amxv(alpha, diag, x, nc, y);
    smat_amxv(alpha, offdiag1, x+nc, nc, y);
    smat_amxv(alpha, offdiag3, x+nxnc, nc, y);
    smat_amxv(alpha, offdiag5, x+nxync, nc, y);

    for (i=1; i<nx; ++i) {
        idx = i*nc;
        matidx = idx*nc;
        idx1 = idx - nc;
        matidx1 = idx1*nc;
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
        smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);
    }

    for (i=nx; i<nxy; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
        smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);

    }

    for (i=nxy; i<end3; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
        smat_amxv(alpha, offdiag5+matidx, x+idx+nxync, nc, y+idx);
    }

    for (i=end3; i<end2; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
        smat_amxv(alpha, offdiag3+matidx, x+idx+nxnc, nc, y+idx);
    }

    for (i=end2; i<end1; ++i) {
        idx = i*nc;
        idx1 = idx-nc;
        idx2 = idx-nxnc;
        idx3 = idx-nxync;
        matidx = idx*nc;
        matidx1 = idx1*nc;
        matidx2 = idx2*nc;
        matidx3 = idx3*nc;
        smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
        smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
        smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
        smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);
        smat_amxv(alpha, offdiag1+matidx, x+idx+nc, nc, y+idx);
    }

    i=end1;
    idx = i*nc;
    idx1 = idx-nc;
    idx2 = idx-nxnc;
    idx3 = idx-nxync;
    matidx = idx*nc;
    matidx1 = idx1*nc;
    matidx2 = idx2*nc;
    matidx3 = idx3*nc;
    smat_amxv(alpha, offdiag4+matidx3, x+idx3, nc, y+idx);
    smat_amxv(alpha, offdiag2+matidx2, x+idx2, nc, y+idx);
    smat_amxv(alpha, offdiag0+matidx1, x+idx1, nc, y+idx);
    smat_amxv(alpha, diag+matidx, x+idx, nc, y+idx);

    return;

}

static inline void str_spaAxpy (const REAL      alpha,
                                const dSTRmat   *A,
                                const REAL      *x,
                                REAL            *y)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // local variables
    INT k;
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT size = nc*ngrid;
    INT nc2  = nc*nc;
    INT ncw  = 0;
    INT start_data = 0;
    INT start_vecx = 0;
    INT start_vecy = 0;
    INT start_vect = 0;
    REAL beta = 0.0;

    if (alpha == 0) {
        return; // nothing should be done
    }

    beta = 1.0/alpha;

    // y:  = beta*y
    for (k = 0; k < size; ++k) {
        y[k] *= beta;
    }

    // y:  = y + A*x
    if (nc > 1) {
        // Deal with the diagonal band
        for (block = 0; block < ngrid; ++block) {
            start_data = nc2*block;
            start_vect = nc*block;
            blkcontr_str(start_data,start_vect,start_vect,nc,diag,x,y);
        }

        // Deal with the off-diagonal bands
        for (band = 0; band < nband; band ++) {
            width = offsets[band];
            ncw   = nc*width;
            if (width < 0) {
                for (block = 0; block < ngrid+width; ++block) {
                    start_data = nc2*block;
                    start_vecx = nc*block;
                    start_vecy = start_vecx - ncw;
                    blkcontr_str(start_data,start_vecx,start_vecy,nc,offdiag[band],x,y);
                }
            }
            else {
                for (block = 0; block < ngrid-width; ++block) {
                    start_data = nc2*block;
                    start_vecy = nc*block;
                    start_vecx = start_vecy + ncw;
                    blkcontr_str(start_data,start_vecx,start_vecy,nc,offdiag[band],x,y);
                }
            }
        }
    }
    else if (nc == 1) {
        // Deal with the diagonal band
        for (point = 0; point < ngrid; point ++) {
            y[point] += diag[point]*x[point];
        }

        // Deal with the off-diagonal bands
        for (band = 0; band < nband; band ++) {
            width = offsets[band];
            if (width < 0) {
                for (point = 0; point < ngrid+width; point ++) {
                    y[point-width] += offdiag[band][point]*x[point];
                }
            }
            else {
                for (point = 0; point < ngrid-width; point ++) {
                    y[point] += offdiag[band][point]*x[point+width];
                }
            }
        }
    }
    else {
        printf("### WARNING: nc is illegal!  %s\n", __FUNCTION__);
        return;
    }

    // y:  = alpha*y
    for (k = 0; k < size; ++k) {
        y[k] *= alpha;
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.97 BlaVector.c File Reference

BLAS1 operations for vectors.

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

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

a	REAL factor a
x	Pointer to dvector x
y	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

a	REAL factor a
x	Pointer to dvector x
y	Pointer to dvector y
z	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

x	Pointer to dvector x
y	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

x	Pointer to dvector x

Returns

L_inf 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

x	Pointer to dvector x
y	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
```

```
#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_blas_dvec_axpy (const REAL     a,
                          const dvector *x,
                          dvector       *y)
{
    const INT   m    = x->row;
    const REAL *xpt = x->val;
    REAL       *ypt = y->val;

    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    INT myid, mybegin, myend, nthreads;
    if ( m > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if ( y->row != m ) {
        printf("### ERROR: Vectors have different dimensions!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel private(myid,mybegin,myend,i) num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            fasp_get_start_end (myid, nthreads, m, &mybegin, &myend);
            for ( i = mybegin; i < myend; ++i ) ypt[i] += a*xpt[i];
        }
#endif
    }
    else {
        for ( i = 0; i < m; ++i ) ypt[i] += a*xpt[i];
    }
}

void fasp_blas_dvec_axpyz (const REAL     a,
                           const dvector *x,
                           const dvector *y,
                           dvector       *z)
{
    const INT     m = x->row;
    const REAL *xpt = x->val, *ypt = y->val;
    REAL       *zpt = z->val;

    if ( y->row != m ) {
        printf("### ERROR: Vectors have different dimensions!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

    z->row = m;

    memcpy (zpt, ypt, m*sizeof(REAL));
    fasp_blas_darray_axpy (m, a, xpt, zpt);
}

REAL fasp_blas_dvec_norm1 (const dvector *x)
{
    const INT   length = x->row;
    const REAL *xpt = x->val;

    register REAL onenorm = 0.0;
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    if ( length > OPENMP_HOLDS ) {
        use_openmp = TRUE;
    }
#endif

```

```c
    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+:onenorm) private(i)
#endif
        for ( i = 0; i < length; ++i ) onenorm += ABS(xpt[i]);
    }
    else {
        for ( i = 0; i < length; ++i ) onenorm += ABS(xpt[i]);
    }

    return onenorm;
}

REAL fasp_blas_dvec_norm2 (const dvector *x)
{
    const INT   length = x->row;
    const REAL *xpt = x->val;

    register REAL twonorm = 0.0;
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    if ( length > OPENMP_HOLDS ) use_openmp = TRUE;
#endif

    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+:twonorm) private(i)
#endif
        for ( i = 0; i < length; ++i ) twonorm += xpt[i]*xpt[i];
    }
    else {
        for ( i = 0; i < length; ++i ) twonorm += xpt[i]*xpt[i];
    }

    return sqrt(twonorm);
}

REAL fasp_blas_dvec_norminf (const dvector *x)
{
    const INT      length=x->row;
    const REAL    *xpt=x->val;

    register REAL  infnorm = 0.0;
    register INT   i;

    for ( i = 0; i < length; ++i ) infnorm = MAX(infnorm, ABS(xpt[i]));

    return infnorm;
}

REAL fasp_blas_dvec_dotprod (const dvector *x,
                             const dvector *y)
{
    const INT   length = x->row;
    const REAL *xpt = x->val, *ypt = y->val;

    register REAL value = 0.0;
    SHORT use_openmp = FALSE;
    INT   i;

#ifdef _OPENMP
    if ( length > OPENMP_HOLDS ) use_openmp = TRUE;
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+:value) private(i)
#endif
        for ( i = 0; i < length; ++i ) value += xpt[i] * ypt[i];
    }
    else {
        for ( i = 0; i < length; ++i ) value += xpt[i] * ypt[i];
    }

    return value;
}

REAL fasp_blas_dvec_relerr (const dvector *x,
                            const dvector *y)
```

```
{
    const INT   length = x->row;
    const REAL *xpt = x->val, *ypt = y->val;

    SHORT use_openmp = FALSE;
    REAL diff = 0.0, temp = 0.0;
    INT   i;

    if ( length != y->row ) {
        printf("### ERROR: Vectors have different dimensions!\n");
        fasp_chkerr(ERROR_DATA_STRUCTURE, __FUNCTION__);
    }

#ifdef _OPENMP
    if ( length > OPENMP_HOLDS ) use_openmp = TRUE;
#endif

    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for reduction(+:temp,diff) private(i)
#endif
        for ( i = 0; i < length; ++i ) {
            temp += xpt[i]*xpt[i];
            diff += pow(xpt[i]-ypt[i],2);
        }
    }
    else {
        for ( i = 0; i < length; ++i ) {
            temp += xpt[i]*xpt[i];
            diff += pow(xpt[i]-ypt[i],2);
        }
    }

    return sqrt(diff/temp);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 (Itr) functions. It requires: AuxArray.c, AuxMemory.c, AuxMessage.c, AuxThreads.c, AuxTiming.c, BlaSmallMatInv.c, BlaSmallMat.c, BlaArray.c, BlaSpmvBSR.c, and PreBSR.c

// TODO: Need to optimize routines here! –Chensong
Definition in file ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
order	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
mark	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
order	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
mark	Pointer to NULL or to the user-defined ordering
diaginv	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 545 of file ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 582 of file ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 724 of file ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	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 ItrSmootherBSR.c.

9.99.2.7 fasp_smoother_dbsr_gs_order1()

```
void fasp_smoother_dbsr_gs_order1 (
             dBSRmat * A,
             dvector * b,
             dvector * u,
             REAL * diaginv,
             INT * mark )
```

Gauss-Seidel relaxation in the user-defined order.

Parameters

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A
mark	Pointer to the user-defined ordering

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 868 of file ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
mark	Pointer to the user-defined ordering
work	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
x	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
data	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 ItrSmootherBSR.c.

9.99.2.10 fasp_smoother_dbsr_jacobi()

```
void fasp_smoother_dbsr_jacobi (
            dBSRmat * A,
            dvector * b,
            dvector * u )
```

Jacobi relaxation.

Parameters

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
diaginv	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 ItrSmootherBSR.c.

9.99.2.12 fasp_smoother_dbsr_jacobi_setup()

```
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.

Parameters

A	Pointer to dBSRmat: the coefficient matrix
diaginv	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
order	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
mark	Pointer to NULL or to the user-defined ordering
weight	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 ItrSmootherBSR.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 * diaginv,
            REAL weight )
```

SOR relaxation.

Parameters

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
order	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
mark	Pointer to NULL or to the user-defined ordering
diaginv	Inverses for all the diagonal blocks of A
weight	Over-relaxation weight

Author

Zhiyang Zhou

Date

2010/10/25

Definition at line 1146 of file ItrSmootherBSR.c.

9.99.2.15 fasp_smoother_dbsr_sor_ascend()

```
void fasp_smoother_dbsr_sor_ascend (
            dBSRmat * A,
            dvector * b,
            dvector * u,
            REAL * diaginv,
            REAL weight )
```

SOR relaxation in the ascending order.

Parameters

A	Pointer to dBSRmat: the coefficient matrix

Parameters

b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A
weight	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial guess, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A
weight	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 ItrSmootherBSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
diaginv	Inverses for all the diagonal blocks of A
mark	Pointer to the user-defined ordering
weight	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 ItrSmootherBSR.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 ItrSmootherBSR.c.

9.100 ItrSmootherBSR.c

Go to the documentation of this file.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#ifdef _OPENMP

#if ILU_MC_OMP
static inline void perm (const INT, const INT, const REAL *, const INT *, REAL *);
static inline void invperm (const INT, const INT, const REAL *, const INT *, REAL *);
```

```
#endif

#endif

REAL ilu_solve_time = 0.0;
/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_smoother_dbsr_jacobi (dBSRmat *A,
                                dvector *b,
                                dvector *u)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    size = ROW*nb2;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    const REAL   *val = A->val;

    // local variables
    INT   i,k;
    SHORT nthreads = 1, use_openmp = FALSE;
    REAL *diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // get all the diagonal sub-blocks
    if (use_openmp) {
        INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,k)
#endif
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                for (k=IA[i]; k<IA[i+1]; ++k)
                    if (JA[k] == i)
                        memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }

    // compute the inverses of all the diagonal sub-blocks
    if (nb > 1) {
        if (use_openmp) {
            INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    fasp_smat_inv(diaginv+i*nb2, nb);
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv+i*nb2, nb);
            }
        }
    }
    else {
        if (use_openmp) {
            INT mybegin, myend, myid;
```

```
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    diaginv[i] = 1.0 / diaginv[i];
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv[i] = 1.0 / diaginv[i];
            }
        }
    }

    fasp_smoother_dbsr_jacobi1(A, b, u, diaginv);

    fasp_mem_free(diaginv); diaginv = NULL;
}

void fasp_smoother_dbsr_jacobi_setup (dBSRmat *A,
                                      REAL    *diaginv)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    const REAL   *val = A->val;

    // local variables
    INT i,k;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // get all the diagonal sub-blocks
    if (use_openmp) {
        INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,k)
#endif
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                for (k=IA[i]; k<IA[i+1]; ++k)
                    if (JA[k] == i)
                        memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }

    // compute the inverses of all the diagonal sub-blocks
    if (nb > 1) {
        if (use_openmp) {
            INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    fasp_smat_inv(diaginv+i*nb2, nb);
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv+i*nb2, nb);
            }
        }
    }
    else {
        if (use_openmp) {
            INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    diaginv[i] = 1.0 / diaginv[i];
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv[i] = 1.0 / diaginv[i];
            }
        }
    }

}

void fasp_smoother_dbsr_jacobi1 (dBSRmat *A,
                                 dvector *b,
                                 dvector *u,
                                 REAL    *diaginv)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    size = ROW*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // auxiliary array
    REAL *b_tmp = NULL;

    // local variables
    INT i,j,k;
    INT pb;

    // b_tmp = b_val
    b_tmp = (REAL *)fasp_mem_calloc(size, sizeof(REAL));
    memcpy(b_tmp, b_val, size*sizeof(REAL));

    // No need to assign the smoothing order since the result doesn't depend on it
    if (nb == 1) {
        if (use_openmp) {
            INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,j,k)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            b_tmp[i] -= val[k]*u_val[j];
                    }
                }
            }
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    u_val[i] = b_tmp[i]*diaginv[i];
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        b_tmp[i] -= val[k]*u_val[j];
                }
            }
            for (i = 0; i < ROW; ++i) {
                u_val[i] = b_tmp[i]*diaginv[i];
            }
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else if (nb > 1) {
        if (use_openmp) {
            INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,pb,k,j)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    pb = i*nb;
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+pb, nb);
                    }
                }
            }
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,pb)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    pb = i*nb;
                    fasp_blas_smat_mxv(diaginv+nb2*i, b_tmp+pb, u_val+pb, nb);
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                pb = i*nb;
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+pb, nb);
                }
            }

            for (i = 0; i < ROW; ++i) {
                pb = i*nb;
                fasp_blas_smat_mxv(diaginv+nb2*i, b_tmp+pb, u_val+pb, nb);
            }

        }
        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        printf("### ERROR: nb is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }
```

```

}

void fasp_smoother_dbsr_gs (dBSRmat *A,
                            dvector *b,
                            dvector *u,
                            INT      order,
                            INT     *mark )
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    size = ROW*nb2;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    const REAL   *val = A->val;

    // local variables
    INT    i,k;
    SHORT  nthreads = 1, use_openmp = FALSE;
    REAL  *diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // get all the diagonal sub-blocks
    if (use_openmp) {
        INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,k)
#endif
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                for (k=IA[i]; k<IA[i+1]; ++k)
                    if (JA[k] == i)
                        memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }

    // compute the inverses of all the diagonal sub-blocks
    if (nb > 1) {
        if (use_openmp) {
            INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    fasp_smat_inv(diaginv+i*nb2, nb);
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv+i*nb2, nb);
            }
        }
    }
    else {
        if (use_openmp) {
            INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    diaginv[i] = 1.0 / diaginv[i];
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv[i] = 1.0 / diaginv[i];
            }
        }
    }

    fasp_smoother_dbsr_gs1(A, b, u, order, mark, diaginv);

    fasp_mem_free(diaginv); diaginv = NULL;
}

void fasp_smoother_dbsr_gs1 (dBSRmat *A,
                             dvector *b,
                             dvector *u,
                             INT      order,
                             INT     *mark,
                             REAL    *diaginv)
{
    if (!mark) {
        if (order == ASCEND) // smooth ascendingly
        {
            fasp_smoother_dbsr_gs_ascend(A, b, u, diaginv);
        }
        else if (order == DESCEND) // smooth descendingly
        {
            fasp_smoother_dbsr_gs_descend(A, b, u, diaginv);
        }
    }
    // smooth according to the order 'mark' defined by user
    else {
        fasp_smoother_dbsr_gs_order1(A, b, u, diaginv, mark);
    }
}

void fasp_smoother_dbsr_gs_ascend (dBSRmat *A,
                                   dvector *b,
                                   dvector *u,
                                   REAL    *diaginv)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT   i,j,k;
    INT   pb;
    REAL  rhs = 0.0;

    if (nb == 1) {
        for (i = 0; i < ROW; ++i) {
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs*diaginv[i];
        }
    }
    else if (nb > 1) {
        REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

        for (i = 0; i < ROW; ++i) {
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            fasp_blas_smat_mxv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        printf("### ERROR: nb is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_gs_ascend1 (dBSRmat *A,
                                    dvector *b,
                                    dvector *u)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT   i,j,k;
    INT   pb;
    REAL  rhs = 0.0;

    if (nb == 1) {
        for (i = 0; i < ROW; ++i) {
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs;
        }
    }
    else if (nb > 1) {
        REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

        for (i = 0; i < ROW; ++i) {
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        printf("### ERROR: nb is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_gs_descend (dBSRmat *A,
                                    dvector *b,
                                    dvector *u,
                                    REAL    *diaginv )
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT i,j,k;
    INT pb;
    REAL rhs = 0.0;

    if (nb == 1) {
        for (i = ROW-1; i >= 0; i--) {
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs*diaginv[i];
        }
    }
    else if (nb > 1) {
        REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

        for (i = ROW-1; i >= 0; i--) {
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            fasp_blas_smat_mxv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        printf("### ERROR: nb is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_gs_descend1 (dBSRmat *A,
                                     dvector *b,
                                     dvector *u)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT i,j,k;
    INT pb;
    REAL rhs = 0.0;

    if (nb == 1) {
        for (i = ROW-1; i >= 0; i--) {
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs;
        }
    }
    else if (nb > 1) {
        REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

        for (i = ROW-1; i >= 0; i--) {
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        printf("### ERROR: nb is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_gs_order1 (dBSRmat *A,
                                   dvector *b,
                                   dvector *u,
                                   REAL    *diaginv,
                                   INT     *mark)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT i,j,k;
    INT I,pb;
    REAL rhs = 0.0;

    if (nb == 1) {
        for (I = 0; I < ROW; ++I) {
            i = mark[I];
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs*diaginv[i];
        }
    }
    else if (nb > 1) {
        REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));

        for (I = 0; I < ROW; ++I) {
            i = mark[I];
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            fasp_blas_smat_mxv(diaginv+nb2*i, b_tmp, u_val+pb, nb);
        }

        fasp_mem_free(b_tmp); b_tmp = NULL;
    }
    else {
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_gs_order2 (dBSRmat *A,
                                   dvector *b,
                                     dvector *u,
                                     INT     *mark,
                                     REAL    *work)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // auxiliary array
    REAL *b_tmp = work;

    // local variables
    INT i,j,k,I,pb;
    REAL rhs = 0.0;

    if (nb == 1) {
        for (I = 0; I < ROW; ++I) {
            i = mark[I];
            rhs = b_val[i];
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    rhs -= val[k]*u_val[j];
            }
            u_val[i] = rhs;
        }
    }
    else if (nb > 1) {
        for (I = 0; I < ROW; ++I) {
            i = mark[I];
            pb = i*nb;
            memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
            for (k = IA[i]; k < IA[i+1]; ++k) {
                j = JA[k];
                if (j != i)
                    fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
            }
            memcpy(u_val+pb, b_tmp, nb*sizeof(REAL));
        }
    }
    else {
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }
}

void fasp_smoother_dbsr_sor (dBSRmat *A,
                             dvector *b,
                             dvector *u,
                             INT      order,
                             INT     *mark,
                             REAL     weight)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    size = ROW*nb2;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    const REAL   *val = A->val;

    // local variables
    INT i,k;
    REAL *diaginv = NULL;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( ROW > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif
```

```

    // allocate memory
    diaginv = (REAL *)fasp_mem_calloc(size, sizeof(REAL));

    // get all the diagonal sub-blocks
    if (use_openmp) {
        INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,k)
#endif
        for (myid=0; myid<nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i=mybegin; i<myend; i++) {
                for (k=IA[i]; k<IA[i+1]; ++k)
                    if (JA[k] == i)
                        memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }
    else {
        for (i = 0; i < ROW; ++i) {
            for (k = IA[i]; k < IA[i+1]; ++k) {
                if (JA[k] == i)
                    memcpy(diaginv+i*nb2, val+k*nb2, nb2*sizeof(REAL));
            }
        }
    }

    // compute the inverses of all the diagonal sub-blocks
    if (nb > 1) {
        if (use_openmp) {
            INT mybegin,myend,myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    fasp_smat_inv(diaginv+i*nb2, nb);
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                fasp_smat_inv(diaginv+i*nb2, nb);
            }
        }
    }
    else {
        if (use_openmp) {
            INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i)
#endif
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    diaginv[i] = 1.0 / diaginv[i];
                }
            }
        }
        else {
            for (i = 0; i < ROW; ++i) {
                // zero-diagonal should be tested previously
                diaginv[i] = 1.0 / diaginv[i];
            }
        }
    }

    fasp_smoother_dbsr_sor1(A, b, u, order, mark, diaginv, weight);

    fasp_mem_free(diaginv); diaginv = NULL;
}

void fasp_smoother_dbsr_sor1 (dBSRmat *A,
                              dvector *b,
                              dvector *u,
                              INT      order,
                              INT     *mark,
                              REAL    *diaginv,
                              REAL     weight)
```

```
{
    if (!mark) {
        if (order == ASCEND)        // smooth ascendingly
        {
            fasp_smoother_dbsr_sor_ascend(A, b, u, diaginv, weight);
        }
        else if (order == DESCEND) // smooth descendingly
        {
            fasp_smoother_dbsr_sor_descend(A, b, u, diaginv, weight);
        }
    }
    // smooth according to the order 'mark' defined by user
    else {
        fasp_smoother_dbsr_sor_order(A, b, u, diaginv, mark, weight);
    }
}

void fasp_smoother_dbsr_sor_ascend (dBSRmat *A,
                                    dvector *b,
                                    dvector *u,
                                    REAL    *diaginv,
                                    REAL     weight)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    const REAL   *val = A->val;

    // values of dvector b and u
    const REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    const INT nb2 = nb*nb;
    INT i,j,k;
    INT pb;
    REAL rhs = 0.0;
    REAL one_minus_weight = 1.0 - weight;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    if (nb == 1) {
#ifdef _OPENMP
        if (ROW > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, rhs, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i = mybegin; i < myend; i++) {
                    rhs = b_val[i];
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            rhs -= val[k]*u_val[j];
                    }
                    u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
                }
            }
        }
        else {
#endif
            for (i = 0; i < ROW; ++i) {
                rhs = b_val[i];
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        rhs -= val[k]*u_val[j];
                }
                u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
            }
#ifdef _OPENMP
        }
#endif
    }
    else if (nb > 1) {
#ifdef _OPENMP
```

```
        if (ROW > OPENMP_HOLDS) {
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
#pragma omp parallel for private(myid, mybegin, myend, i, pb, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (i = mybegin; i < myend; i++) {
                    pb = i*nb;
                    memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
                    }
                    fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb, one_minus_weight,
u_val+pb, nb);
                }
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
        }
        else {
#endif
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
            for (i = 0; i < ROW; ++i) {
                pb = i*nb;
                memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
                }
                fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight, u_val+pb, nb);
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
#ifdef _OPENMP
        }
#endif
    }
    else {
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_sor_descend (dBSRmat *A,
                                     dvector *b,
                                     dvector *u,
                                     REAL    *diaginv,
                                     REAL     weight)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT     nb  = A->nb;
    const INT     nb2 = nb*nb;
    const INT    *IA  = A->IA;
    const INT    *JA  = A->JA;
    REAL         *val = A->val;
    const REAL    one_minus_weight = 1.0 - weight;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT i,j,k;
    INT pb;
    REAL rhs = 0.0;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    if (nb == 1) {
#ifdef _OPENMP
        if (ROW > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, rhs, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                mybegin = ROW-1-mybegin; myend = ROW-1-myend;
                for (i = mybegin; i > myend; i--) {
                    rhs = b_val[i];
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            rhs -= val[k]*u_val[j];
                    }
                    u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
                }
            }
        }
        else {
#endif
            for (i = ROW-1; i >= 0; i--) {
                rhs = b_val[i];
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        rhs -= val[k]*u_val[j];
                }
                u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
            }
#ifdef _OPENMP
        }
#endif
    }
    else if (nb > 1) {
#ifdef _OPENMP
        if (ROW > OPENMP_HOLDS) {
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
#pragma omp parallel for private(myid, mybegin, myend, i, pb, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                mybegin = ROW-1-mybegin; myend = ROW-1-myend;
                for (i = mybegin; i > myend; i--) {
                    pb = i*nb;
                    memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+myid*nb, nb);
                    }
                    fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb,
                                          one_minus_weight, u_val+pb, nb);
                }
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
        }
        else {
#endif
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
            for (i = ROW-1; i >= 0; i--) {
                pb = i*nb;
                memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
                }
                fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight,
                                      u_val+pb, nb);
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
#ifdef _OPENMP
        }
#endif
    }
    else {
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_sor_order (dBSRmat *A,
                                   dvector *b,
                                   dvector *u,
                                   REAL    *diaginv,
                                   INT     *mark,
                                   REAL     weight)
{
    // members of A
    const INT     ROW = A->ROW;
    const INT      nb   = A->nb;
    const INT      nb2 = nb*nb;
    const INT     *IA   = A->IA;
    const INT     *JA   = A->JA;
    REAL          *val = A->val;
    const REAL     one_minus_weight = 1.0 - weight;

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT i,j,k;
    INT I,pb;
    REAL rhs = 0.0;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    if (nb == 1) {
#ifdef _OPENMP
        if (ROW > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, I, i, rhs, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (I = mybegin; I < myend; ++I) {
                    i = mark[I];
                    rhs = b_val[i];
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            rhs -= val[k]*u_val[j];
                    }
                    u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
                }
            }
        }
        else {
#endif
            for (I = 0; I < ROW; ++I) {
                i = mark[I];
                rhs = b_val[i];
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        rhs -= val[k]*u_val[j];
                }
                u_val[i] = one_minus_weight*u_val[i] + weight*(rhs*diaginv[i]);
            }
#ifdef _OPENMP
        }
#endif
    }
    else if (nb > 1) {
#ifdef _OPENMP
        if (ROW > OPENMP_HOLDS) {
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb*nthreads, sizeof(REAL));
#pragma omp parallel for private(myid, mybegin, myend, I, i, pb, k, j)
            for (myid = 0; myid < nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
                for (I = mybegin; I < myend; ++I) {
                    i = mark[I];
                    pb = i*nb;
                    memcpy(b_tmp+myid*nb, b_val+pb, nb*sizeof(REAL));
                    for (k = IA[i]; k < IA[i+1]; ++k) {
                        j = JA[k];
                        if (j != i)
                            fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp+myid*nb, nb);
                    }
                    fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp+myid*nb,
                                          one_minus_weight, u_val+pb, nb);
                }
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
        }
        else {
#endif
            REAL *b_tmp = (REAL *)fasp_mem_calloc(nb, sizeof(REAL));
            for (I = 0; I < ROW; ++I) {
                i = mark[I];
                pb = i*nb;
                memcpy(b_tmp, b_val+pb, nb*sizeof(REAL));
                for (k = IA[i]; k < IA[i+1]; ++k) {
                    j = JA[k];
                    if (j != i)
                        fasp_blas_smat_ymAx(val+k*nb2, u_val+j*nb, b_tmp, nb);
                }
                fasp_blas_smat_aAxpby(weight, diaginv+nb2*i, b_tmp, one_minus_weight,
                                      u_val+pb, nb);
            }
            fasp_mem_free(b_tmp); b_tmp = NULL;
#ifdef _OPENMP
        }
#endif
    }
    else {
        fasp_chkerr(ERROR_NUM_BLOCKS, __FUNCTION__);
    }

}

void fasp_smoother_dbsr_ilu (dBSRmat *A,
                             dvector *b,
                             dvector *x,
                             void    *data)
{
    ILU_data   *iludata=(ILU_data *)data;
    const INT   nb=iludata->nb,m=A->ROW*nb, memneed=5*m;

    REAL *xval = x->val, *bval = b->val;
    REAL *zr = iludata->work + 3*m;
    REAL *z  = zr + m;

    double start, end;

    if (iludata->nwork<memneed) goto MEMERR;

    fasp_darray_cp(m,bval,zr); fasp_blas_dbsr_aAxpy(-1.0,A,xval,zr);

#ifdef __OPENMP

#if ILU_MC_OMP
    REAL *tz = (REAL*)fasp_mem_calloc(A->ROW*A->nb, sizeof(REAL));
    REAL *tzr = (REAL*)fasp_mem_calloc(A->ROW*A->nb, sizeof(REAL));
    perm(A->ROW, A->nb, zr, iludata->jlevL, tzr);

    fasp_gettime(&start);
    fasp_precond_dbsr_ilu_mc_omp(tzr,tz,iludata);
    fasp_gettime(&end);

    invperm(A->ROW, A->nb, tz, iludata->jlevL, z);
    fasp_mem_free(tzr); tzr = NULL;
    fasp_mem_free(tz);  tz  = NULL;
#else
    fasp_gettime(&start);
    fasp_precond_dbsr_ilu_ls_omp(zr,z,iludata);
    fasp_gettime(&end);
#endif

    ilu_solve_time += end-start;

#else

    fasp_gettime(&start);
    fasp_precond_dbsr_ilu(zr,z,iludata);
    fasp_gettime(&end);
    ilu_solve_time += end-start;

#endif

    fasp_blas_darray_axpy(m,1,z,xval);

    return;

MEMERR:
    printf("### ERROR: ILU needs %d memory, only %d available!  [%s:%d]\n",
           memneed, iludata->nwork, __FILE__, __LINE__);
    fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
}
```

```

/*----------------------------------*/
/*--      Private Functions       --*/
/*----------------------------------*/

#ifdef _OPENMP

#if ILU_MC_OMP

static inline void perm (const INT    n,
                         const INT    nb,
                         const REAL  *x,
                         const INT   *p,
                         REAL        *y)
{
    INT i, j, indx, indy;

#ifdef _OPENMP
#pragma omp parallel for private(i, j, indx, indy)
#endif
    for (i=0; i<n; ++i) {
        indx = p[i]*nb;
        indy = i*nb;
        for (j=0; j<nb; ++j) {
            y[indy+j] = x[indx+j];
        }
    }
}

static inline void invperm (const INT    n,
                            const INT    nb,
                            const REAL  *x,
                            const INT   *p,
                            REAL        *y)
{
    INT i, j, indx, indy;

#ifdef _OPENMP
#pragma omp parallel for private(i, j, indx, indy)
#endif
    for (i=0; i<n; ++i) {
        indx = i*nb;
        indy = p[i]*nb;
        for (j=0; j<nb; ++j) {
            y[indy+j] = x[indx+j];
        }
    }
}

#endif // end of ILU_MC_OMP

#endif // end of _OPENMP

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

9.101 ItrSmootherCSR.c File Reference

Smoothers for dCSRmat matrices.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.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 (Itr) functions. It requires: AuxArray.c, AuxMemory.c, AuxMessage.c, AuxThreads.c, BlaArray.c, and BlaSpmvCSR.c

Definition in file ItrSmootherCSR.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

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)

Parameters

i↩ _↩ *1*	Starting index
i↩ _↩ *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 ItrSmootherCSR.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 ItrSmootherCSR.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

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
x	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
data	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 ItrSmootherCSR.c.

9.101.2.4 fasp_smoother_dcsr_jacobi()

```
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.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i↩ _↩ *1*	Starting index

Parameters

i←↩ _←↩ 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 ItrSmootherCSR.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←↩ _←↩ 1	Starting index
i←↩ _←↩ 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 ItrSmootherCSR.c.

9.101.2.6 fasp_smoother_dcsr_L1diag()

```
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.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i↩ *_*↩ *1*	Starting index
i↩ *_*↩ *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 ItrSmootherCSR.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

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

Author

Xiaozhe Hu

Date

10/26/2010

Modified by Chunsheng Feng, Zheng Li on 09/01/2012
Definition at line 628 of file ItrSmootherCSR.c.

9.101.2.8 fasp_smoother_dcsr_sor()

```
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.

Parameters

u	Pointer to dvector: the unknowns (IN: initial, OUT: approximation)
i_1	Starting index
i_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

10/26/2010

Modified by Chunsheng Feng, Zheng Li on 09/01/2012
Definition at line 744 of file ItrSmootherCSR.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

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
w	Over-relaxation weight
mark	C/F marker array
order	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 ItrSmootherCSR.c.

9.102 ItrSmootherCSR.c

Go to the documentation of this file.

```

#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"
```

```

/*---------------------------------*/
/*--      Public 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)
{
    const INT    N   = ABS(i_n - i_1) + 1;
    const INT   *ia = A->IA, *ja = A->JA;
    const REAL  *aval = A->val, *bval = b->val;
    REAL        *uval = u->val;

    // local variables
    INT i,j,k,begin_row,end_row;

    // OpenMP variables
#ifdef _OPENMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    REAL *t = (REAL *)fasp_mem_calloc(N,sizeof(REAL));
    REAL *d = (REAL *)fasp_mem_calloc(N,sizeof(REAL));

    while (L--) {

        if (s>0) {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, begin_row, end_row, i, k, j)
                for (myid=0; myid<nthreads; ++myid) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin += i_1; myend += i_1;
                    for (i=mybegin; i<myend; i+=s) {
                        t[i]=bval[i];
                        begin_row=ia[i],end_row=ia[i+1];
                        for (k=begin_row; k<end_row; ++k) {
                            j=ja[k];
                            if (i!=j) t[i]-=aval[k]*uval[j];
                            else d[i]=aval[k];
                        }
                    }
                }
            }
            else {
#endif
                for (i=i_1;i<=i_n;i+=s) {
                    t[i]=bval[i];
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        if (i!=j) t[i]-=aval[k]*uval[j];
                        else d[i]=aval[k];
                    }
                }
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
#pragma omp parallel for private (i)
#endif
            for (i=i_1;i<=i_n;i+=s) {
                if (ABS(d[i])>SMALLREAL) uval[i]=(1-w)*uval[i]+ w*t[i]/d[i];
            }

        }

        else {

#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, k, j)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin = i_1-mybegin; myend = i_1-myend;
                    for (i=mybegin; i>myend; i+=s) {
                        t[i]=bval[i];
                        begin_row=ia[i],end_row=ia[i+1];
                        for (k=begin_row; k<end_row; ++k) {
                            j=ja[k];
                            if (i!=j) t[i]-=aval[k]*uval[j];
                            else d[i]=aval[k];
                        }
                    }
                }
            }
            else {
#endif
                for (i=i_1;i>=i_n;i+=s) {
                    t[i]=bval[i];
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        if (i!=j) t[i]-=aval[k]*uval[j];
                        else d[i]=aval[k];
                    }
                }
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
#pragma omp parallel for private(i)
#endif
            for (i=i_1;i>=i_n;i+=s) {
                if (ABS(d[i])>SMALLREAL) uval[i]=(1-w)*uval[i]+ w*t[i]/d[i];
            }

        }

    } // end while

    fasp_mem_free(t); t = NULL;
    fasp_mem_free(d); d = NULL;

    return;
}

void fasp_smoother_dcsr_gs (dvector    *u,
                            const INT   i_1,
                            const INT   i_n,
                            const INT   s,
                            dCSRmat    *A,
                            dvector    *b,
                            INT         L)
{
    const INT   *ia = A->IA, *ja = A->JA;
    const REAL  *aval = A->val, *bval = b->val;
    REAL        *uval = u->val;

    // local variables
    INT   i,j,k,begin_row,end_row;
    REAL  t,d=0.0;

#ifdef _OPENMP
    const INT    N = ABS(i_n - i_1)+1;
    INT   myid, mybegin, myend;
    INT   nthreads = fasp_get_num_threads();
#endif

    if (s > 0) {

        while (L--) {
#ifdef _OPENMP
            if (N >OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, d, k, j)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin += i_1, myend += i_1;
                    for (i=mybegin; i<myend; i+=s) {
                        t = bval[i];
                        begin_row=ia[i],end_row=ia[i+1];
#if DIAGONAL_PREF // diagonal first
                        d=aval[begin_row];
```

```
                              if (ABS(d)>SMALLREAL) {
                                  for (k=begin_row+1;k<end_row;++k) {
                                      j=ja[k];
                                      t-=aval[k]*uval[j];
                                  }
                                  uval[i]=t/d;
                              }
#else // general order
                              for (k=begin_row;k<end_row;++k) {
                                  j=ja[k];
                                  if (i!=j)
                                      t-=aval[k]*uval[j];
                                  else if (ABS(aval[k])>SMALLREAL) d=1.e+0/aval[k];
                              }
                              uval[i]=t*d;
#endif // end DIAGONAL_PREF
                          } // end for i
                      }

                  }

                  else {
#endif
                      for (i=i_1;i<=i_n;i+=s) {
                          t = bval[i];
                          begin_row=ia[i]; end_row=ia[i+1];

#if DIAGONAL_PREF // diagonal first
                          d=aval[begin_row];
                          if (ABS(d)>SMALLREAL) {
                              for (k=begin_row+1;k<end_row;++k) {
                                  j=ja[k];
                                  t-=aval[k]*uval[j];
                              }
                              uval[i]=t/d;
                          }
#else // general order
                          for (k=begin_row;k<end_row;++k) {
                              j=ja[k];
                              if (i!=j)
                                  t-=aval[k]*uval[j];
                              else if (ABS(aval[k])>SMALLREAL) d=1.e+0/aval[k];
                          }
                          uval[i]=t*d;
#endif
                      } // end for i
#ifdef _OPENMP
                  }
#endif
              } // end while

          } // if s
          else {

              while (L--) {
#ifdef _OPENMP
                  if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, d, k, j, t)
                      for (myid=0; myid<nthreads; myid++) {
                          fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                          mybegin = i_1 - mybegin; myend = i_1 - myend;
                          for (i=mybegin; i>myend; i+=s) {
                              t=bval[i];
                              begin_row=ia[i],end_row=ia[i+1];
#if DIAGONAL_PREF // diagonal first
                              d=aval[begin_row];
                              if (ABS(d)>SMALLREAL) {
                                  for (k=begin_row+1;k<end_row;++k) {
                                      j=ja[k];
                                      t-=aval[k]*uval[j];
                                  }
                                  uval[i]=t/d;
                              }
#else // general order
                              for (k=begin_row;k<end_row;++k) {
                                  j=ja[k];
                                  if (i!=j)
                                      t-=aval[k]*uval[j];
                                  else if (ABS(aval[k])>SMALLREAL) d=1.0/aval[k];
                              }
                              uval[i]=t*d;
```

```
#endif
                    } // end for i
                }
            }
            else {
#endif
                for (i=i_1;i>=i_n;i+=s) {
                    t=bval[i];
                    begin_row=ia[i]; end_row=ia[i+1];
#if DIAGONAL_PREF // diagonal first
                    d=aval[begin_row];
                    if (ABS(d)>SMALLREAL) {
                        for (k=begin_row+1;k<end_row;++k) {
                            j=ja[k];
                            t-=aval[k]*uval[j];
                        }
                        uval[i]=t/d;
                    }
#else // general order
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        if (i!=j)
                            t-=aval[k]*uval[j];
                        else if (ABS(aval[k])>SMALLREAL) d=1.0/aval[k];
                    }
                    uval[i]=t*d;
#endif
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while

    } // end if

    return;
}

void fasp_smoother_dcsr_gs_cf (dvector   *u,
                               dCSRmat   *A,
                               dvector   *b,
                               INT        L,
                               INT       *mark,
                               const INT  order)
{
    const INT    nrow = b->row; // number of rows
    const INT   *ia = A->IA, *ja = A->JA;
    const REAL  *aval = A->val, *bval = b->val;
    REAL        *uval = u->val;

    INT i,j,k,begin_row,end_row;
    REAL t,d=0.0;

#ifdef _OPENMP
    INT myid,mybegin,myend;
    INT nthreads = fasp_get_num_threads();
#endif

    // F-point first, C-point second
    if (order == FPFIRST) {

        while (L--) {

#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i,t,begin_row,end_row,k,j,d)
                for (myid = 0; myid < nthreads; myid ++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i=mybegin; i<myend; i++) {
                        if (mark[i] != 1) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                            d = aval[begin_row];
                            for (k = begin_row+1; k < end_row; k ++) {
                                j = ja[k];
                                t -= aval[k]*uval[j];
                            } // end for k
#else
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
```

```c
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
#endif // end if DIAG_PREF
                            if (ABS(d) > SMALLREAL) uval[i] = t/d;
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] != 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                        d = aval[begin_row];
                        for (k = begin_row+1; k < end_row; k ++) {
                            j = ja[k];
                            t -= aval[k]*uval[j];
                        } // end for k
#else
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
#endif // end if DIAG_PREF
                        if (ABS(d) > SMALLREAL) uval[i] = t/d;
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid,mybegin,myend,i,t,begin_row,end_row,k,j,d)
                for (myid = 0; myid < nthreads; myid ++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i=mybegin; i<myend; i++) {
                        if (mark[i] == 1) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                            d = aval[begin_row];
                            for (k = begin_row+1; k < end_row; k ++) {
                                j = ja[k];
                                t -= aval[k]*uval[j];
                            } // end for k
#else
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
#endif // end if DIAG_PREF
                            if (ABS(d) > SMALLREAL) uval[i] = t/d;
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] == 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                        d = aval[begin_row];
                        for (k = begin_row+1; k < end_row; k ++) {
                            j = ja[k];
                            t -= aval[k]*uval[j];
                        } // end for k
#else
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
#endif // end if DIAG_PREF
```

```
                        if (ABS(d) > SMALLREAL) uval[i] = t/d;
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while

    }

    // C-point first, F-point second
    else {

        while (L--) {
#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid,mybegin,myend,t,i,begin_row,end_row,k,j,d)
                for (myid = 0; myid < nthreads; myid ++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i=mybegin; i<myend; i++) {
                        if (mark[i] == 1) {
                            t = bval[i];
                            begin_row = ia[i],end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                            d = aval[begin_row];
                            for (k = begin_row+1; k < end_row; k ++) {
                                j = ja[k];
                                t -= aval[k]*uval[j];
                            } // end for k
#else
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
#endif // end if DIAG_PREF
                            if (ABS(d) > SMALLREAL) uval[i] = t/d;
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++)  {
                    if (mark[i] == 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 09/22/2012
                        d = aval[begin_row];
                        for (k = begin_row+1; k < end_row; k ++) {
                            j = ja[k];
                            t -= aval[k]*uval[j];
                        } // end for k
#else
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
#endif // end if DIAG_PREF
                        if (ABS(d) > SMALLREAL) uval[i] = t/d;
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i,t,begin_row,end_row,k,j,d)
                for (myid = 0; myid < nthreads; myid ++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i=mybegin; i<myend; i++) {
                        if (mark[i] != 1) {
                            t = bval[i];
                            begin_row = ia[i],end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 01/17/2013
                            d = aval[begin_row];
                            for (k = begin_row+1; k < end_row; k ++) {
                                j = ja[k];
                                t -= aval[k]*uval[j];
                            } // end for k
#else
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
#endif // end if DIAG_PREF
                            if (ABS(d) > SMALLREAL) uval[i] = t/d;
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] != 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
#if DIAGONAL_PREF // Added by Chensong on 09/22/2012
                        d = aval[begin_row];
                        for (k = begin_row+1; k < end_row; k ++) {
                            j = ja[k];
                            t -= aval[k]*uval[j];
                        } // end for k
#else
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
#endif // end if DIAG_PREF
                        if (ABS(d) > SMALLREAL) uval[i] = t/d;
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while

    } // end if order

    return;
}

void fasp_smoother_dcsr_sgs (dvector *u,
                             dCSRmat *A,
                             dvector *b,
                             INT      L)
{
    const INT    nm1=b->row-1;
    const INT   *ia=A->IA,*ja=A->JA;
    const REAL  *aval=A->val,*bval=b->val;
    REAL        *uval=u->val;

    // local variables
    INT   i,j,k,begin_row,end_row;
    REAL  t,d=0;

#ifdef _OPENMP
    INT  myid, mybegin, myend, up;
    INT  nthreads = fasp_get_num_threads();
#endif

    while (L--) {
        // forward sweep
#ifdef _OPENMP
        up = nm1 + 1;
        if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, j, k, d)
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
                for (i=mybegin; i<myend; i++) {
                    t=bval[i];
                    begin_row=ia[i], end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        if (i!=j) t-=aval[k]*uval[j];
                        else d=aval[k];
                    } // end for k
                    if (ABS(d)>SMALLREAL) uval[i]=t/d;
                } // end for i
            }
        }
        else {
#endif
            for (i=0;i<=nm1;++i) {
                t=bval[i];
                begin_row=ia[i]; end_row=ia[i+1];
                for (k=begin_row;k<end_row;++k) {
                    j=ja[k];
                    if (i!=j) t-=aval[k]*uval[j];
                    else d=aval[k];
                } // end for k
                if (ABS(d)>SMALLREAL) uval[i]=t/d;
            } // end for i
#ifdef _OPENMP
        }
#endif

        // backward sweep
#ifdef _OPENMP
        up = nm1;
        if (up > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
            for (myid=0; myid<nthreads; myid++) {
                fasp_get_start_end(myid, nthreads, up, &mybegin, &myend);
                mybegin = nm1-1-mybegin; myend = nm1-1-myend;
                for (i=mybegin; i>myend; i--) {
                    t=bval[i];
                    begin_row=ia[i], end_row=ia[i+1];
                    for (k=begin_row; k<end_row; k++) {
                        j=ja[k];
                        if (i!=j) t-=aval[k]*uval[j];
                        else d=aval[k];
                    } // end for k
                    if (ABS(d)>SMALLREAL) uval[i]=t/d;
                } // end for i
            }
        }
        else {
#endif
            for (i=nm1-1;i>=0;--i) {
                t=bval[i];
                begin_row=ia[i]; end_row=ia[i+1];
                for (k=begin_row;k<end_row;++k) {
                    j=ja[k];
                    if (i!=j) t-=aval[k]*uval[j];
                    else d=aval[k];
                } // end for k
                if (ABS(d)>SMALLREAL) uval[i]=t/d;
            } // end for i
#ifdef _OPENMP
        }
#endif
    } // end while

    return;
}

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)
{
    const INT   *ia=A->IA,*ja=A->JA;
    const REAL  *aval=A->val,*bval=b->val;
    REAL        *uval=u->val;

    // local variables
    INT    i,j,k,begin_row,end_row;
    REAL   t, d=0;

#ifdef _OPENMP
    const INT    N = ABS(i_n - i_1)+1;
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif
```

```

    while (L--) {
        if (s>0) {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin += i_1, myend += i_1;
                    for (i=mybegin; i<myend; i+=s) {
                        t=bval[i];
                        begin_row=ia[i], end_row=ia[i+1];
                        for (k=begin_row; k<end_row; k++) {
                            j=ja[k];
                            if (i!=j)
                                t-=aval[k]*uval[j];
                            else
                                d=aval[k];
                        }
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                }

            }
            else {
#endif
                for (i=i_1; i<=i_n; i+=s) {
                    t=bval[i];
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row; k<end_row; ++k) {
                        j=ja[k];
                        if (i!=j)
                            t-=aval[k]*uval[j];
                        else
                            d=aval[k];
                    }
                    if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                }
#ifdef _OPENMP
            }
#endif
        }
        else {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin = i_1 - mybegin, myend = i_1 - myend;
                    for (i=mybegin; i>myend; i+=s) {
                        t=bval[i];
                        begin_row=ia[i],end_row=ia[i+1];
                        for (k=begin_row;k<end_row;++k) {
                            j=ja[k];
                            if (i!=j)
                                t-=aval[k]*uval[j];
                            else
                                d=aval[k];
                        }
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                }
            }
            else {
#endif
                for (i=i_1;i>=i_n;i+=s) {
                    t=bval[i];
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        if (i!=j)
                            t-=aval[k]*uval[j];
                        else
                            d=aval[k];
                    }
                    if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                }
#ifdef _OPENMP
            }
#endif
        }
    }  // end while

    return;
}

void fasp_smoother_dcsr_sor_cf (dvector    *u,
                                dCSRmat    *A,
                                dvector    *b,
                                INT         L,
                                const REAL  w,
                                INT        *mark,
                                const INT   order )
{
    const INT    nrow = b->row; // number of rows
    const INT   *ia = A->IA, *ja=A->JA;
    const REAL  *aval = A->val,*bval=b->val;
    REAL        *uval = u->val;

    // local variables
    INT    i,j,k,begin_row,end_row;
    REAL   t,d=0.0;

#ifdef _OPENMP
    INT    myid, mybegin, myend;
    INT    nthreads = fasp_get_num_threads();
#endif

    // F-point first
    if (order == -1) {
        while (L--) {
#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private (myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i = mybegin; i < myend; i ++) {
                        if (mark[i] == 0 || mark[i] == 2) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
                            if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                        }
                    }
                }
            } // end for i
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] == 0 || mark[i] == 2) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, i, mybegin, myend, t, begin_row, end_row, k, j, d)
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i = mybegin; i < myend; i++) {
                        if (mark[i] == 1) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
                            if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] == 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while
    }
    else {
        while (L--) {
#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, t, k, j, d, begin_row, end_row)
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i = mybegin; i < myend; i++) {
                        if (mark[i] == 1) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
                            if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                        }
                    } // end for i
                }
            }
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] == 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif

#ifdef _OPENMP
            if (nrow > OPENMP_HOLDS) {
#pragma omp parallel for private (myid, mybegin, myend, i, t, begin_row, end_row, k, j, d)
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, nrow, &mybegin, &myend);
                    for (i = mybegin; i < myend; i++) {
                        if (mark[i] != 1) {
                            t = bval[i];
                            begin_row = ia[i], end_row = ia[i+1];
                            for (k = begin_row; k < end_row; k ++) {
                                j = ja[k];
                                if (i!=j) t -= aval[k]*uval[j];
                                else d = aval[k];
                            } // end for k
                            if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                        }
                    }
                }
            } // end for i
            else {
#endif
                for (i = 0; i < nrow; i ++) {
                    if (mark[i] != 1) {
                        t = bval[i];
                        begin_row = ia[i]; end_row = ia[i+1];
                        for (k = begin_row; k < end_row; k ++) {
                            j = ja[k];
                            if (i!=j) t -= aval[k]*uval[j];
                            else d = aval[k];
                        } // end for k
                        if (ABS(d)>SMALLREAL) uval[i]=w*(t/d)+(1-w)*uval[i];
                    }
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while
    }

    return;
}

void fasp_smoother_dcsr_ilu (dCSRmat *A,
                             dvector *b,
                             dvector *x,
                             void    *data)
{
    const INT m=A->row, m2=2*m, memneed=3*m;
    const ILU_data *iludata=(ILU_data *)data;

    REAL *zz = iludata->work;
    REAL *zr = iludata->work+m;
    REAL *z  = iludata->work+m2;

    if (iludata->nwork<memneed) goto MEMERR;

    {
        INT i, j, jj, begin_row, end_row;
        REAL *lu = iludata->luval;
        INT *ijlu = iludata->ijlu;
        REAL *xval = x->val, *bval = b->val;

        fasp_darray_cp(m,bval,zr); fasp_blas_dcsr_aAxpy(-1.0,A,xval,zr);

        // forward sweep:  solve unit lower matrix equation L*zz=zr
        zz[0]=zr[0];
        for (i=1;i<m;++i) {
            begin_row=ijlu[i]; end_row=ijlu[i+1];
            for (j=begin_row;j<end_row;++j) {
                jj=ijlu[j];
                if (jj<i) zr[i]-=lu[j]*zz[jj];
                else break;
            }
            zz[i]=zr[i];
        }

        // backward sweep:  solve upper matrix equation U*z=zz
        z[m-1]=zz[m-1]*lu[m-1];
        for (i=m-2;i>=0;--i) {
            begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
            for (j=end_row;j>=begin_row;--j) {
                jj=ijlu[j];
                if (jj>i) zz[i]-=lu[j]*z[jj];
                else break;
            }
            z[i]=zz[i]*lu[i];
        }

        fasp_blas_darray_axpy(m,1,z,xval);
    }

    return;

MEMERR:
    printf("### ERROR: ILU needs %d memory, only %d available!  [%s:%d]\n",
           memneed, iludata->nwork, __FILE__, __LINE__);
    fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
}

```

```
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)
{
    const INT   *ia=A->IA,*ja=A->JA;
    const REAL  *aval=A->val,*bval=b->val;
    REAL        *uval=u->val;

    // local variables
    INT   i,j,k,begin_row,end_row;
    REAL  temp1,temp2,alpha;

#ifdef _OPENMP
    const INT    N = ABS(i_n - i_1)+1;
    INT   myid, mybegin, myend;
    INT   nthreads = fasp_get_num_threads();
#endif

    if (s > 0) {

        while (L--) {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, temp1, temp2, begin_row, end_row, k, alpha, j)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin += i_1, myend += i_1;
                    for (i=mybegin; i<myend; i+=s) {
                        temp1 = 0; temp2 = 0;
                        begin_row=ia[i], end_row=ia[i+1];
                        for (k=begin_row; k<end_row; k++) {
                            j=ja[k];
                            temp1 += aval[k]*aval[k];
                            temp2 += aval[k]*uval[j];
                        } // end for k
                    }
                    alpha = (bval[i] - temp2)/temp1;
                    for (k=begin_row; k<end_row; ++k){
                        j = ja[k];
                        uval[j] += w*alpha*aval[k];
                    }// end for k
                } // end for i
            }
            else {
#endif
                for (i=i_1;i<=i_n;i+=s) {
                    temp1 = 0; temp2 = 0;
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        temp1 += aval[k]*aval[k];
                        temp2 += aval[k]*uval[j];
                    } // end for k
                    alpha = (bval[i] - temp2)/temp1;
                    for (k=begin_row;k<end_row;++k){
                        j = ja[k];
                        uval[j] += w*alpha*aval[k];
                    }// end for k
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while

    } // if s

    else {
        while (L--) {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, temp1, temp2, begin_row, end_row, k, alpha, j)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin = i_1 - mybegin, myend = i_1 - myend;
                    for (i=mybegin; i>myend; i+=s) {
                        temp1 = 0; temp2 = 0;
                        begin_row=ia[i], end_row=ia[i+1];
                        for (k=begin_row;k<end_row;++k) {
                            j=ja[k];
                            temp1 += aval[k]*aval[k];
                            temp2 += aval[k]*uval[j];
                        } // end for k
                        alpha = (bval[i] - temp2)/temp1;
                        for (k=begin_row;k<end_row;++k){
                            j = ja[k];
                            uval[j] += w*alpha*aval[k];
                        }// end for k
                    } // end for i
                }
            }
            else {
#endif
                for (i=i_1;i>=i_n;i+=s) {
                    temp1 = 0; temp2 = 0;
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        temp1 += aval[k]*aval[k];
                        temp2 += aval[k]*uval[j];
                    } // end for k
                    alpha = (bval[i] - temp2)/temp1;
                    for (k=begin_row;k<end_row;++k){
                        j = ja[k];
                        uval[j] += w*alpha*aval[k];
                    }// end for k
                } // end for i
#ifdef _OPENMP
            }
#endif
        } // end while

    } // end if

    return;
}

void fasp_smoother_dcsr_L1diag (dvector    *u,
                                const INT   i_1,
                                const INT   i_n,
                                const INT   s,
                                dCSRmat    *A,
                                dvector    *b,
                                INT         L)
{
    const INT    N = ABS(i_n - i_1)+1;
    const INT   *ia=A->IA, *ja=A->JA;
    const REAL  *aval=A->val,*bval=b->val;
    REAL        *uval=u->val;

    // local variables
    INT   i,j,k,begin_row,end_row;

#ifdef _OPENMP
    INT   myid, mybegin, myend;
    INT   nthreads = fasp_get_num_threads();
#endif

    // Checks should be outside of for; t,d can be allocated before calling!!!  --Chensong
    REAL *t = (REAL *)fasp_mem_calloc(N,sizeof(REAL));
    REAL *d = (REAL *)fasp_mem_calloc(N,sizeof(REAL));

    while (L--) {
        if (s>0) {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, begin_row, end_row, k, j)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin += i_1, myend += i_1;
                    for (i=mybegin; i<myend; i+=s) {
                        t[i]=bval[i]; d[i]=0.0;
                        begin_row=ia[i], end_row=ia[i+1];
                        for (k=begin_row; k<end_row; k++) {
                            j=ja[k];
                            t[i]-=aval[k]*uval[j];
                            d[i]+=ABS(aval[k]);
                        }
                    }
                }
#pragma omp parallel for private(i)
                for (i=i_1;i<=i_n;i+=s) {
                    if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
                }
            }
            else {
#endif
                for (i=i_1;i<=i_n;i+=s) {
                    t[i]=bval[i]; d[i]=0.0;
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        t[i]-=aval[k]*uval[j];
                        d[i]+=ABS(aval[k]);
                    }
                }

                for (i=i_1;i<=i_n;i+=s) {
                    if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
                }
#ifdef _OPENMP
            }
#endif
        }
        else {
#ifdef _OPENMP
            if (N > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, k, j, begin_row, end_row)
                for (myid=0; myid<nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, N, &mybegin, &myend);
                    mybegin = i_1 - mybegin, myend = i_1 - myend;
                    for (i=mybegin; i>myend; i+=s) {
                        t[i]=bval[i]; d[i]=0.0;
                        begin_row=ia[i], end_row=ia[i+1];
                        for (k=begin_row; k<end_row; k++) {
                            j=ja[k];
                            t[i]-=aval[k]*uval[j];
                            d[i]+=ABS(aval[k]);
                        }
                    }
                }
#pragma omp parallel for private(i)
                for (i=i_1;i>=i_n;i+=s) {
                    if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
                }
            }
            else {
#endif
                for (i=i_1;i>=i_n;i+=s) {
                    t[i]=bval[i];d[i]=0.0;
                    begin_row=ia[i]; end_row=ia[i+1];
                    for (k=begin_row;k<end_row;++k) {
                        j=ja[k];
                        t[i]-=aval[k]*uval[j];
                        d[i]+=ABS(aval[k]);
                    }
                }

                for (i=i_1;i>=i_n;i+=s) {
                    if (ABS(d[i])>SMALLREAL) u->val[i]+=t[i]/d[i];
                }
#ifdef _OPENMP
            }
#endif
        }

    } // end while

    fasp_mem_free(t); t = NULL;
    fasp_mem_free(d); d = NULL;

    return;
}

#if 0
static dCSRmat form_contractor (dCSRmat    *A,
                                const INT   smoother,
                                const INT   steps,
                                const INT   ndeg,
                             const REAL  relax,
                             const REAL  dtol)
{
    const INT   n=A->row;
    INT         i;

    REAL *work = (REAL *)fasp_mem_calloc(2*n,sizeof(REAL));

    dvector b, x;
    b.row=x.row=n;
    b.val=work; x.val=work+n;

    INT *index = (INT *)fasp_mem_calloc(n,sizeof(INT));

    for (i=0; i<n; ++i) index[i]=i;

    dCSRmat B = fasp_dcsr_create(n, n, n*n); // too much memory required, need to change!!

    dCSRmat C, D;

    for (i=0; i<n; ++i){

        // get i-th column
        fasp_dcsr_getcol(i, A, b.val);

        // set x =0.0
        fasp_dvec_set(n, &x, 0.0);

        // smooth
        switch (smoother) {
            case GS:
                fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
                break;
            case POLY:
                fasp_smoother_dcsr_poly(A, &b, &x, n, ndeg, steps);
                break;
            case JACOBI:
                fasp_smoother_dcsr_jacobi(&x, 0, n-1, 1, A, &b, steps);
                break;
            case SGS:
                fasp_smoother_dcsr_sgs(&x, A, &b, steps);
                break;
            case SOR:
                fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
                break;
            case SSOR:
                fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
                fasp_smoother_dcsr_sor(&x, n-1, 0,-1, A, &b, steps, relax);
                break;
            case GSOR:
                fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
                fasp_smoother_dcsr_sor(&x, n-1, 0, -1, A, &b, steps, relax);
                break;
            case SGSOR:
                fasp_smoother_dcsr_gs(&x, 0, n-1, 1, A, &b, steps);
                fasp_smoother_dcsr_gs(&x, n-1, 0,-1, A, &b, steps);
                fasp_smoother_dcsr_sor(&x, 0, n-1, 1, A, &b, steps, relax);
                fasp_smoother_dcsr_sor(&x, n-1, 0,-1, A, &b, steps, relax);
                break;
            default:
                printf("### ERROR: Unknown smoother type!  [%s:%d]\n",
                       __FILE__, __LINE__);
                fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
        }

        // store to B
        B.IA[i] = i*n;
        memcpy(&(B.JA[i*n]), index, n*sizeof(INT));
        memcpy(&(B.val[i*n]), x.val, x.row*sizeof(REAL));

    }

    B.IA[n] = n*n;

    // drop small entries
    compress_dCSRmat(&B, &D, dtol);

    // get contractor
    fasp_dcsr_trans(&D, &C);

    // clean up
    fasp_mem_free(work); work = NULL;
    fasp_dcsr_free(&B);
    fasp_dcsr_free(&D);

    return C;
}
#endif

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.103 ItrSmootherCSRcr.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 (Itr) functions. It requires: AuxMessage.c

// TODO: Need to optimize routines here! –Chensong
Definition in file ItrSmootherCSRcr.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

pt	Relax type, e.g., cpt, fpt, etc..
n	Number of variables
u	Iterated solution
ia	Row pointer
ja	Column index
a	Pointers to sparse matrix values in CSR format
b	Pointer to right hand side
L	Number of iterations
CF	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 ItrSmootherCSRcr.c.

9.104 ItrSmootherCSRcr.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                         else {
00072                             d=a[k];
00073                         }
```

```
                        if (ABS(d)>SMALLREAL) {
                            u[i]=t/d;
                        }
                        else {
                            printf("### ERROR: Diagonal entry_%d (%e) close to 0!\n",
                                   i, d);
                            fasp_chkerr(ERROR_MISC, __FUNCTION__);
                        }
                    }
                }
            }
            else {
                u[i]=0.e0;
            }
        }
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.105 ItrSmootherCSRpoly.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&LTZ2010

9.105.1 Detailed Description

Smoothers for dCSRmat matrices using poly. approx. to A^{-1}.

Note

This file contains Level-2 (Itr) 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-leve 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 ItrSmootherCSRpoly.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

Amat	Pointer to stiffness matrix, consider square matrix.
brhs	Pointer to right hand side
usol	Pointer to solution
n	Problem size
ndeg	Degree of poly
L	Number of iterations

Author

Fei Cao, Xiaozhe Hu

Date

05/24/2012

Definition at line 67 of file ItrSmootherCSRpoly.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&LTZ2010

Parameters

Amat	Pointer to stiffness matrix
brhs	Pointer to right hand side
usol	Pointer to solution
n	Problem size
ndeg	Degree of poly
L	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 ItrSmootherCSRpoly.c.

9.106 ItrSmootherCSRpoly.c

Go to the documentation of this file.

```c

#include <math.h>
#include <time.h>
#include <float.h>
#include <limits.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static void bminax (REAL *,INT *,INT *, REAL *, REAL *,INT *, REAL *);
static void Diaginv (dCSRmat *, REAL *);
static REAL DinvAnorminf (dCSRmat *, REAL *);
static void Diagx (REAL *, INT, REAL *, REAL *);
static void Rr (dCSRmat *, REAL *, REAL *, REAL *, REAL *, REAL *, REAL *, REAL *, INT);
static void fasp_aux_uuplv0_ (REAL *, REAL *, INT *);
static void fasp_aux_norm1_ (INT *, INT *, REAL *, INT *, REAL *);

/*---------------------------------*/
/*--      Public Function        --*/
/*---------------------------------*/

void fasp_smoother_dcsr_poly (dCSRmat *Amat,
                              dvector *brhs,
                              dvector *usol,
                              INT      n,
                              INT      ndeg,
                              INT      L)
{
    // local variables
    INT i;
    REAL *b = brhs->val, *u = usol->val;
    REAL *Dinv = NULL, *r = NULL, *rbar = NULL, *v0 = NULL, *v1 = NULL;
    REAL *error = NULL, *k = NULL;
    REAL mu0, mu1, smu0, smu1;

    /* allocate memory */
    Dinv  = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    r     = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    rbar  = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    v0    = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    v1    = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    error = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    k     = (REAL *) fasp_mem_calloc(6,sizeof(REAL)); // coefficients for calculation

    // get the inverse of the diagonal of A
    Diaginv(Amat, Dinv);

    // set up parameter
    mu0 = DinvAnorminf(Amat, Dinv); // get the inf norm of Dinv*A;

    mu0 = 1.0/mu0; mu1 = 4.0*mu0; // default set 8;
    smu0 =  sqrt(mu0); smu1 = sqrt(mu1);

    k[1] = (mu0+mu1)/2.0;
    k[2] = (smu0 + smu1)*(smu0 + smu1)/2.0;
    k[3] = mu0 * mu1;

    // 4.0*mu0*mu1/(sqrt(mu0)+sqrt(mu1))/(sqrt(mu0)+sqrt(mu1));
    k[4] = 2.0*k[3]/k[2];

    // square of (sqrt(kappa)-1)/(sqrt(kappa)+1);
    k[5] = (mu1-2.0*smu0*smu1+mu0)/(mu1+2.0*smu0*smu1+mu0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // Update
    for ( i=0; i<L; i++ ) {
        // get residual
        fasp_blas_dcsr_mxv(Amat, u, r);// r= Amat*u;
        fasp_blas_darray_axpyz(n, -1, r, b, r);// r= -r+b;

        // Get correction error = R*r
        Rr(Amat, Dinv, r, rbar, v0, v1, error, k, ndeg);

        // update solution
        fasp_blas_darray_axpy(n, 1, error, u);

    }

#if DEBUG_MODE > 1
    printf("### DEBUG: Degree of polysmoothing is:  %d\n",ndeg);
#endif

    // free memory
    fasp_mem_free(Dinv);  Dinv  = NULL;
    fasp_mem_free(r);     r     = NULL;
    fasp_mem_free(rbar);  rbar  = NULL;
    fasp_mem_free(v0);    v0    = NULL;
    fasp_mem_free(v1);    v1    = NULL;
    fasp_mem_free(error); error = NULL;
    fasp_mem_free(k);     k     = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

void fasp_smoother_dcsr_poly_old (dCSRmat *Amat,
                                  dvector *brhs,
                                  dvector *usol,
                                  INT      n,
                                  INT      ndeg,
                                  INT      L)
{
    INT  *ia=Amat->IA,*ja=Amat->JA;
    INT   i,j,k,it,jk,iaa,iab,ndeg0;  // id and ij for scaling of A

    REAL *a=Amat->val, *b=brhs->val, *u=usol->val;
    REAL *v,*v0,*r,*vsave;  // one can get away without r as well;
    REAL  smaxa,smina,delinv,s,smu0,smu1,skappa,th,th1,sq;
    REAL  ri,ari,vj,ravj,snj,sm,sm01,smsqrt,delta,delta2,chi;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /* WORKING MEM */
    v     = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    v0    = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    vsave = (REAL *) fasp_mem_calloc(n,sizeof(REAL));
    r     = (REAL *) fasp_mem_calloc(n,sizeof(REAL));

    /* COMPUTE PARAMS*/
    // min INT for approx -- could be done upfront
    // i.e., only once per level...  only norm1 ...
    fasp_aux_norm1_(ia,ja,a,&n,&smaxa);
    smina=smaxa/8;
    delinv=(smaxa+smina)/(smaxa-smina);
    th=delinv+sqrt(delinv*delinv-1e+00);
    th1=1e+00/th;
    sq=(th-th1)*(th-th1);
    //
    ndeg0=(int)floor(log(2*(2e0+th+th1)/sq)/log(th)+1e0);
    if (ndeg0 < ndeg) ndeg0=ndeg;
    //
    smu0=1e+00/smaxa;
    smu1=1e+00/smina;
    skappa=sqrt(smaxa/smina);
    delta=(skappa-1e+00)/(skappa+1);
    delta2=delta*delta;
    s=sqrt(smu0)+sqrt(smu1);
    s=s*s;
    smsqrt=0.5e+00*s;
    chi=4e+00*smu0*smu1/s;
    sm=0.5e+00*(smu0+smu1);
    sm01=smu0*smu1;

#if DEBUG_MODE > 1
    printf("### DEBUG: Degree of polysmoothing is:  %d\n",ndeg);
#endif

    /* BEGIN POLY ITS */

    /* auv_(ia,ja,a,u,u,&n,&err0); NA: u = 0 */
    //bminax(b,ia,ja,a,u,&n,r);
    //for (i=0; i < n; ++i) {res0 += r[i]*r[i];}
    //res0=sqrt(res0);

    for (it = 0 ; it < L; it++) {
        bminax(b,ia,ja,a,u,&n,r);
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,iaa,iab,ari,jk,j,ri) if(n>OPENMP_HOLDS)
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
#else
            for (i=0; i < n ; ++i) {
#endif
                iaa = ia[i];
                iab = ia[i+1];
                ari=0e+00; /* ari is (A*r)[i] */
                if(iab > iaa) {
                    for (jk = iaa; jk < iab; jk++) {
                        j=ja[jk];
                        ari += a[jk] * r[j];
                    }
                }
                ri=r[i];
                v0[i]=sm*ri;
                v[i]=smsqrt*ri-sm01*ari;
            }
#ifdef _OPENMP
        }
#endif
        for (i=1; i < ndeg0; ++i) {
            //for (j=0; j < n ; ++j) vsave[j]=v[j];
            fasp_darray_cp(n, v, vsave);

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,j,ravj,iaa,iab,jk,k,vj,snj) if(n>OPENMP_HOLDS)
            for (myid=0; myid<nthreads; ++myid) {
                fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
                for (j=mybegin; j<myend; ++j) {
#else
                for (j=0; j < n ; ++j) {
#endif
                    /* ravj = (r- A*v)[j] */
                    ravj= r[j];
                    iaa = ia[j];
                    iab = ia[j+1];
                    if(iab > iaa) {
                        for (jk = iaa; jk < iab; jk++) {
                            k=ja[jk];
                            ravj -= a[jk] * vsave[k];
                        }
                    }
                    vj=v[j];
                    snj = chi*ravj+delta2*(vj-v0[j]);
                    v0[j]=vj;
                    v[j]=vj+snj;
                }
            }
#ifdef _OPENMP
        }
#endif
        fasp_aux_uuplv0_(u,v,&n);
        //bminax(b,ia,ja,a,u,&n,r);
        //for (i=0; i < n ; ++i)
        //resk += r[i]*r[i];
        //resk=sqrt(resk);
        //fprintf("\nres0=%12.5g\n",res0);
        //fprintf("\nresk=%12.5g\n",resk);
        //res0=resk;
        //resk=0.0e0;
    }

    fasp_mem_free(v);     v     = NULL;
    fasp_mem_free(v0);    v0    = NULL;
    fasp_mem_free(r);     r     = NULL;
    fasp_mem_free(vsave); vsave = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void bminax (REAL *b,
                    INT  *ia,
                    INT  *ja,
                    REAL *a,
                    REAL *x,
                    INT  *nn,
                    REAL *res)
{
    /* Computes b-A*x */

    INT i,j,jk,iaa,iab;
    INT n;
    REAL u;
    n=*nn;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,iaa,iab,u,jk,j) if(n>OPENMP_HOLDS)
    for (myid=0; myid<nthreads; ++myid) {
        fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
        for (i=mybegin; i<myend; ++i) {
#else
        for (i=0; i < n ; ++i) {
#endif
            iaa = ia[i];
            iab = ia[i+1];
            u = b[i];
            if(iab > iaa)
            for (jk = iaa; jk < iab; jk++) {
                j=ja[jk];
                u -= a[jk] * x[j];
            }
            res[i] = u;
        }
#ifdef _OPENMP
    }
#endif
    return;
}
```

```

static void Diaginv (dCSRmat *Amat,
                     REAL    *Dinv)
{
    const INT   n  = Amat->row;
    const INT  *ia = Amat->IA, *ja = Amat->JA;
    const REAL *a  = Amat->val;
    INT i,j;

#ifdef _OPENMP
#pragma omp parallel for private(j) if(n>OPENMP_HOLDS)
#endif
    for (i=0; i<n; i++) {
        for(j=ia[i]; j<ia[i+1]; j++) {
            if(i==ja[j]) // find the diagonal
                break;
        }
        Dinv[i] = 1.0/a[j];
    }
    return;
}

static REAL DinvAnorminf (dCSRmat *Amat,
                          REAL    *Dinv)
{
    //local variable
    const INT   n  = Amat->row;
    const INT  *ia = Amat->IA;
    const REAL *a  = Amat->val;

    INT i,j;
    REAL norm, temp;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    REAL sub_norm = 0.0;
    INT nthreads = fasp_get_num_threads();
#endif

    norm = 0.0;

    // get the infinity norm of Dinv*A
#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,temp,sub_norm) if(n>OPENMP_HOLDS)
    for (myid=0; myid<nthreads; ++myid) {
        fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
        sub_norm = 0.0;
        for (i=mybegin; i<myend; ++i) {
#else
        for (i=0; i<n; i++) {
#endif
            temp = 0.0;
            for (j=ia[i]; j<ia[i+1]; j++) {
                temp += ABS(a[j]);
            }
            temp *= Dinv[i]; // temp is the L1 norm of the ith row of Dinv*A;
#ifdef _OPENMP
            sub_norm = MAX(sub_norm, temp);
#else
            norm = MAX(norm, temp);
#endif
        }
#ifdef _OPENMP
#pragma omp critical(norm)
        norm = MAX(norm, sub_norm);
    }
#endif

    return norm;
}

static void Diagx (REAL *Dinv,
                   INT   n,
                   REAL *x,
                   REAL *b)
{
    INT i;

    // Variables for OpenMP
    SHORT nthreads = 1, use_openmp = FALSE;
    INT myid, mybegin, myend;

#ifdef _OPENMP
    if (n > OPENMP_HOLDS) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    if (use_openmp) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i = mybegin; i < myend; i++) {
                b[i] = Dinv[i] * x[i];
            }
        }
    }
    else {
        for (i=0; i<n; i++) {
            b[i] = Dinv[i] * x[i];
        }
    }
    return;
}

static void Rr (dCSRmat *Amat,
                REAL    *Dinv,
                REAL    *r,
                REAL    *rbar,
                REAL    *v0,
                REAL    *v1,
                REAL    *vnew,
                REAL    *k,
                INT      m)
{
    // local variables
    const INT   n  = Amat->row;
    INT i,j;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    //1 set up rbar
    Diagx(Dinv, n, r, rbar);// rbar = Dinv *r;

    //2 set up v0, v1;
    fasp_blas_dcsr_mxv(Amat, rbar, v1);//v1= A*rbar;
    Diagx(Dinv, n, v1, v1); // v1=Dinv *v1;

#ifdef _OPENMP
#pragma omp parallel for if(n>OPENMP_HOLDS)
#endif
    for(i=0;i<n;i++) {
        v0[i] = k[1] * rbar[i];
        v1[i] = k[2] * rbar[i] - k[3] * v1[i];
    }

    //3 iterate to get v_(j+1)

    for (j=1;j<m;j++) {
        fasp_blas_dcsr_mxv(Amat, v1, rbar);//rbar= A*v_(j);

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i) if(n>OPENMP_HOLDS)
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, n, &mybegin, &myend);
            for (i=mybegin; i<myend; ++i) {
#else
            for(i=0;i<n;i++) {
#endif
                rbar[i] = (r[i] - rbar[i])*Dinv[i];// indeed rbar=Dinv*(r-A*v_(j));
                vnew[i] = v1[i] + k[5] *(v1[i] - v0[i]) + k[4] * rbar[i];// compute v_(j+1)
                // prepare for next cycle
                v0[i]=v1[i];
                v1[i]=vnew[i];
            }
#ifdef _OPENMP
        }
#endif
    }
}

static void fasp_aux_uuplv0_ (REAL *u,
                              REAL *v,
                              INT *n)
{
    /*
This computes y = y + x.
*/
    INT i;
    for ( i=0; i < *n; i++ ) u[i] += v[i];
    return;
}

static void fasp_aux_norm1_ (INT   *ia,
                             INT   *ja,
                             REAL  *a,
                             INT   *nn,
                             REAL  *a1norm)
{
    INT  n,i,jk,iaa,iab;
    REAL sum,s;
    /* computes one norm of a matrix a and stores it in the variable
pointed to by *a1norm*/
    n = *nn;
    s = 0.0;
    for ( i=0; i < n ; i++ ) {
        iaa = ia[i];
        iab = ia[i+1];
        sum = 0e+00;
        for ( jk = iaa; jk < iab; jk++ ) sum += fabs(a[jk]);
        if ( sum > s ) s = sum;
    }
    *a1norm=s;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.107 ItrSmootherSTR.c File Reference

Smoothers for dSTRmat matrices.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.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 (Itr) functions. It requires: AuxArray.c, AuxMemory.c, AuxMessage.c, BlaSmallMat.c, BlaSmallMatInv.c, BlaSmallMatLU.c, and BlaSpmvSTR.c

Definition in file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
neigh	Pointer to ivector: neighborhoods
diaginv	Pointer to dvector: the inverse of the diagonals
pivot	Pointer to ivector: the pivot of diagonal blocks

Author

Xiaozhe Hu

Date

10/01/2011

Definition at line 1543 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
order	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
mark	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 ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
order	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
mark	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
diaginv	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 ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	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 ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
mark	Pointer to the user-defined order array
order	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 ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	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 ItrSmootherSTR.c.

9.107.2.7 fasp_smoother_dstr_gs_order()

```
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.

Parameters

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
mark	Pointer to the user-defined order array

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 556 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 43 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	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 ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
order	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
mark	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 873 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
order	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
mark	Pointer to the user-defined ordering(when order=0) or CF_marker array(when order!=0)
diaginv	Inverse of the diagonal entries
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 935 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 981 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
mark	Pointer to the user-defined order array
order	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
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 1355 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 1102 of file ItrSmootherSTR.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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
diaginv	All the inverse matrices for all the diagonal block of A when (A->nc)>1, and NULL when (A->nc)=1
mark	Pointer to the user-defined order array
weight	Over-relaxation weight

Author

Shiquan Zhang, Zhiyang Zhou

Date

10/10/2010

Definition at line 1224 of file ItrSmootherSTR.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 ItrSmootherSTR.c.

9.108 ItrSmootherSTR.c

Go to the documentation of this file.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static void blkcontr2 (INT, INT, INT, INT, REAL *, REAL *, REAL *);
static void aAxpby    (REAL, REAL, INT, REAL *, REAL *, REAL *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_smoother_dstr_jacobi (dSTRmat *A,
                                dvector *b,
                                dvector *u)
{
    INT     nc     = A->nc;    // size of each block (number of components)
    INT     ngrid = A->ngrid; // number of grids
    REAL   *diag  = A->diag;  // Diagonal entries
    REAL   *diaginv = NULL;   // Diagonal inverse, same size and storage scheme as A->diag

    INT nc2   = nc*nc;
    INT size  = nc2*ngrid;
    INT block = 0;
    INT start = 0;

    if (nc > 1) {
        // allocate memory
        diaginv = (REAL *)fasp_mem_calloc(size,sizeof(REAL));

        // diaginv = diag;
        fasp_darray_cp(size,diag,diaginv);

        // generate diaginv
        for (block = 0; block < ngrid; block ++) {
            fasp_smat_inv(diaginv+start, nc);
            start += nc2;
        }
    }

    fasp_smoother_dstr_jacobi1(A, b, u, diaginv);

    fasp_mem_free(diaginv); diaginv = NULL;
}


void fasp_smoother_dstr_jacobi1 (dSTRmat *A,
                                 dvector *b,
                                 dvector *u,
                                 REAL    *diaginv)
{
    // information of A
    INT       ngrid = A->ngrid;    // number of grids
    INT          nc = A->nc;       // size of each block (number of components)
    INT       nband = A->nband;    // number of off-diag band
    INT    *offsets = A->offsets;  // offsets of the off-diagals
    REAL      *diag = A->diag;     // Diagonal entries
    REAL  **offdiag = A->offdiag;  // Off-diagonal entries

    // values of dvector b and u
    REAL     *b_val = b->val;
    REAL     *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT size  = nc*ngrid;
    INT nc2   = nc*nc;
    INT start  = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecb = 0;
    INT start_vecu = 0;

    // auxiliary array
    REAL *b_tmp = NULL;

    // this should be done once and for all!!
    b_tmp = (REAL *)fasp_mem_calloc(size,sizeof(REAL));

    // b_tmp = b_val
    fasp_darray_cp(size,b_val,b_tmp);

    // It's not necessary to assign the smoothing order since the results doesn't depend on it
    if (nc == 1) {
        for (point = 0; point < ngrid; point ++) {
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        b_tmp[point] -= offdiag[band][column]*u_val[column];
                    }
                }
                else {// width > 0
                    if (column < ngrid) {
                        b_tmp[point] -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band
        } // end for point

        for (point = 0; point < ngrid; point ++) {
            // zero-diagonal should be tested previously
            u_val[point] = b_tmp[point] / diag[point];
        }
    } // end if (nc == 1)
    else if (nc > 1) {
        for (block = 0; block < ngrid; block ++) {
            start_DATA = nc2*block;
            start_vecb = nc*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, start_vecb,
                                   nc, offdiag[band], u_val, b_tmp );
                    }
                }
                else {// width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, start_vecb,
                                   nc, offdiag[band], u_val, b_tmp );
                    }
                }
            } // end for band
        } // end for block

        for (block = 0; block < ngrid; block ++) {
            start = nc*block;
            fasp_blas_smat_mxv(diaginv+nc2*block, b_tmp+start, u_val+start, nc);
        }
    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(b_tmp); b_tmp = NULL;
}

void fasp_smoother_dstr_gs (dSTRmat    *A,
                            dvector    *b,
                            dvector    *u,
                            const INT   order,
                            INT        *mark)
{
    INT   nc    = A->nc;    // size of each block (number of components)
    INT   ngrid = A->ngrid; // number of grids
    REAL *diag  = A->diag;  // Diagonal entries
    REAL *diaginv = NULL;   // Diagonal inverse(when nc>1),same size and storage scheme as A->diag

    INT nc2   = nc*nc;
    INT size  = nc2*ngrid;
    INT block = 0;
    INT start = 0;

    if (nc > 1) {
        // allocate memory
        diaginv = (REAL *)fasp_mem_calloc(size,sizeof(REAL));

        // diaginv = diag;
        fasp_darray_cp(size,diag,diaginv);

        // generate diaginv
        for (block = 0; block < ngrid; block ++) {
            fasp_smat_inv(diaginv+start, nc);
            start += nc2;
        }
    }

    fasp_smoother_dstr_gs1(A, b, u, order, mark, diaginv);

    fasp_mem_free(diaginv); diaginv = NULL;
}

void fasp_smoother_dstr_gs1 (dSTRmat    *A,
                             dvector    *b,
                             dvector    *u,
                             const INT   order,
                             INT        *mark,
                             REAL       *diaginv)
{

    if (!mark) {
        if (order == ASCEND)        // smooth ascendingly
            {
                fasp_smoother_dstr_gs_ascend(A, b, u, diaginv);
            }
        else if (order == DESCEND) // smooth descendingly
            {
                fasp_smoother_dstr_gs_descend(A, b, u, diaginv);
            }
    }
    else {
        if (order == USERDEFINED)  // smooth according to the order 'mark' defined by user
            {
                fasp_smoother_dstr_gs_order(A, b, u, diaginv, mark);
            }
        else // smooth according to 'mark', where 'mark' is a CF_marker array
            {
                fasp_smoother_dstr_gs_cf(A, b, u, diaginv, mark, order);
            }
    }
}

void fasp_smoother_dstr_gs_ascend (dSTRmat *A,
                                   dvector *b,
                                   dvector *u,
                                   REAL    *diaginv)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband;  // number of off-diag band
    INT  *offsets = A->offsets; // offsets of the off-diagals
    REAL *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (point = 0; point < ngrid; point ++) {
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = rhs / diag[point];

        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        for (block = 0; block < ngrid; block ++) {
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            fasp_blas_smat_mxv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);

        } // end for block

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_gs_descend (dSTRmat *A,
                                    dvector *b,
                                    dvector *u,
                                    REAL    *diaginv)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (point = ngrid-1; point >= 0; point --) {
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = rhs / diag[point];

        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        for (block = ngrid-1; block >= 0; block --) {
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            fasp_blas_smat_mxv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);

        } // end for block

    } // end else if (nc > 1)

    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_gs_order (dSTRmat *A,
                                  dvector *b,
                                  dvector *u,
                                  REAL    *diaginv,
                                  INT     *mark)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband;  // number of off-diag band
    INT  *offsets = A->offsets; // offsets of the off-diagals
    REAL *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT index = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (index = 0; index < ngrid; index ++) {
            point = mark[index];
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = rhs / diag[point];

        } // end for index

    } // end if (nc == 1)

    else if (nc > 1) {
        for (index = 0; index < ngrid; index ++) {
            block = mark[index];
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            fasp_blas_smat_mxv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);

        } // end for index

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_gs_cf (dSTRmat     *A,
                               dvector     *b,
                               dvector     *u,
                                  REAL       *diaginv,
                                  INT        *mark,
                                  const INT   order)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;         // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    INT FIRST  = order;  // which kind of points to be smoothed firstly?
    INT SECOND = -order; // which kind of points to be smoothed secondly?

    REAL rhs = 0.0;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        // deal with the points marked FIRST
        for (point = 0; point < ngrid; point ++) {
            if (mark[point] == FIRST) {
                rhs = b_val[point];
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = point + width;
                    if (width < 0) {
                        if (column >= 0) {
                            rhs -= offdiag[band][column]*u_val[column];
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            rhs -= offdiag[band][point]*u_val[column];
                        }
                    }
                } // end for band

                // zero-diagonal should be tested previously
                u_val[point] = rhs / diag[point];
            } // end if (mark[point] == FIRST)
        } // end for point

        // deal with the points marked SECOND
        for (point = 0; point < ngrid; point ++) {
            if (mark[point] == SECOND) {
                rhs = b_val[point];
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = point + width;
                    if (width < 0) {
                        if (column >= 0) {
                            rhs -= offdiag[band][column]*u_val[column];
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            rhs -= offdiag[band][point]*u_val[column];
                        }
                    }
                } // end for band

                // zero-diagonal should be tested previously
                u_val[point] = rhs / diag[point];
            } // end if (mark[point] == SECOND)
        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        // deal with the blocks marked FIRST
        for (block = 0; block < ngrid; block ++) {
            if (mark[block] == FIRST) {
                ncb = nc*block;
                for (point = 0; point < nc; point ++) {
                    vec_tmp[point] = b_val[ncb+point];
                }
                start_DATA = nc2*block;
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = block + width;
                    if (width < 0) {
                        if (column >= 0) {
                            start_data = nc2*column;
                            start_vecu = nc*column;
                            blkcontr2( start_data, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            start_vecu = nc*column;
                            blkcontr2( start_DATA, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                } // end for band

                // subblock smoothing
                fasp_blas_smat_mxv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
            } // end if (mark[block] == FIRST)

        } // end for block

        // deal with the blocks marked SECOND
        for (block = 0; block < ngrid; block ++) {
            if (mark[block] == SECOND) {
                ncb = nc*block;
                for (point = 0; point < nc; point ++) {
                    vec_tmp[point] = b_val[ncb+point];
                }
                start_DATA = nc2*block;
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = block + width;
                    if (width < 0) {
                        if (column >= 0) {
                            start_data = nc2*column;
                            start_vecu = nc*column;
                            blkcontr2( start_data, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            start_vecu = nc*column;
                            blkcontr2( start_DATA, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                } // end for band

                // subblock smoothing
                fasp_blas_smat_mxv(diaginv+start_DATA, vec_tmp, u_val+nc*block, nc);
            } // end if (mark[block] == SECOND)

        } // end for block

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_sor (dSTRmat    *A,
                             dvector    *b,
                             dvector    *u,
                             const INT   order,
                             INT        *mark,
                             const REAL  weight)
{
    INT     nc    = A->nc;    // size of each block (number of components)
    INT     ngrid = A->ngrid; // number of grids
    REAL   *diag  = A->diag;  // Diagonal entries
    REAL *diaginv = NULL;   // Diagonal inverse(when nc>1),same size and storage scheme as A->diag

    INT nc2   = nc*nc;
    INT size  = nc2*ngrid;
    INT block = 0;
    INT start = 0;

    if (nc > 1) {
        // allocate memory
        diaginv = (REAL *)fasp_mem_calloc(size,sizeof(REAL));

        // diaginv = diag;
        fasp_darray_cp(size,diag,diaginv);

        // generate diaginv
        for (block = 0; block < ngrid; block ++) {
            fasp_smat_inv(diaginv+start, nc);
            start += nc2;
        }
    }

    fasp_smoother_dstr_sor1(A, b, u, order, mark, diaginv, weight);

    fasp_mem_free(diaginv); diaginv = NULL;
}

void fasp_smoother_dstr_sor1 (dSTRmat    *A,
                              dvector    *b,
                              dvector    *u,
                              const INT   order,
                              INT        *mark,
                              REAL       *diaginv,
                              const REAL  weight)
{
    if (!mark) {
        if (order == ASCEND)       // smooth ascendingly
            {
                fasp_smoother_dstr_sor_ascend(A, b, u, diaginv, weight);
            }
        else if (order == DESCEND) // smooth descendingly
            {
                fasp_smoother_dstr_sor_descend(A, b, u, diaginv, weight);
            }
    }
    else {
        if (order == USERDEFINED)  // smooth according to the order 'mark' defined by user
            {
                fasp_smoother_dstr_sor_order(A, b, u, diaginv, mark, weight);
            }
        else // smooth according to 'mark', where 'mark' is a CF_marker array
            {
                fasp_smoother_dstr_sor_cf(A, b, u, diaginv, mark, order, weight);
            }
    }
}

void fasp_smoother_dstr_sor_ascend (dSTRmat *A,
                                    dvector *b,
                                    dvector *u,
                                    REAL    *diaginv,
                                    REAL     weight)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;
    REAL one_minus_weight = 1.0 - weight;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (point = 0; point < ngrid; point ++) {
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = one_minus_weight*u_val[point] +
                weight*(rhs / diag[point]);

        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        for (block = 0; block < ngrid; block ++) {
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            aAxpby(weight, one_minus_weight, nc,
                    diaginv+start_DATA, vec_tmp, u_val+nc*block);

        } // end for block

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_sor_descend (dSTRmat *A,
                                     dvector *b,
                                     dvector *u,
                                     REAL    *diaginv,
                                     REAL     weight)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;
    REAL one_minus_weight = 1.0 - weight;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (point = ngrid-1; point >= 0; point --) {
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = one_minus_weight*u_val[point] +
                weight*(rhs / diag[point]);

        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        for (block = ngrid-1; block >= 0; block --) {
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            aAxpby(weight, one_minus_weight, nc,
                   diaginv+start_DATA, vec_tmp, u_val+nc*block);

        } // end for block

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_sor_order (dSTRmat *A,
                                   dvector *b,
                                   dvector *u,
                                   REAL    *diaginv,
                                   INT     *mark,
                                   REAL     weight)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries

    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT index  = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;
    REAL one_minus_weight = 1.0 - weight;

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        for (index = 0; index < ngrid; index ++) {
            point = mark[index];
            rhs = b_val[point];
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = point + width;
                if (width < 0) {
                    if (column >= 0) {
                        rhs -= offdiag[band][column]*u_val[column];
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        rhs -= offdiag[band][point]*u_val[column];
                    }
                }
            } // end for band

            // zero-diagonal should be tested previously
            u_val[point] = one_minus_weight*u_val[point] +
                weight*(rhs / diag[point]);

        } // end for index

    } // end if (nc == 1)

    else if (nc > 1) {
        for (index = 0; index < ngrid; index ++) {
            block = mark[index];
            ncb = nc*block;
            for (point = 0; point < nc; point ++) {
                vec_tmp[point] = b_val[ncb+point];
            }
            start_DATA = nc2*block;
            for (band = 0; band < nband; band ++) {
                width  = offsets[band];
                column = block + width;
                if (width < 0) {
                    if (column >= 0) {
                        start_data = nc2*column;
                        start_vecu = nc*column;
                        blkcontr2( start_data, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
                else { // width > 0
                    if (column < ngrid) {
                        start_vecu = nc*column;
                        blkcontr2( start_DATA, start_vecu, 0, nc,
                                   offdiag[band], u_val, vec_tmp );
                    }
                }
            } // end for band

            // subblock smoothing
            aAxpby(weight, one_minus_weight, nc,
                   diaginv+start_DATA, vec_tmp, u_val+nc*block);

        } // end for index

    } // end else if (nc > 1)

    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_smoother_dstr_sor_cf (dSTRmat    *A,
                                dvector    *b,
                                dvector    *u,
                                REAL       *diaginv,
                                INT        *mark,
                                const INT   order,
                                const REAL  weight)
{
    // information of A
    INT ngrid = A->ngrid;  // number of grids
    INT nc = A->nc;        // size of each block (number of components)
    INT nband = A->nband ; // number of off-diag band
    INT *offsets = A->offsets; // offsets of the off-diagals
    REAL  *diag = A->diag;       // Diagonal entries
    REAL **offdiag = A->offdiag; // Off-diagonal entries
    // values of dvector b and u
    REAL *b_val = b->val;
    REAL *u_val = u->val;

    // local variables
    INT block = 0;
    INT point = 0;
    INT band  = 0;
    INT width = 0;
    INT nc2   = nc*nc;
    INT ncb   = 0;
    INT column = 0;
    INT start_data = 0;
    INT start_DATA = 0;
    INT start_vecu = 0;
    REAL rhs = 0.0;
    REAL one_minus_weight = 1.0 - weight;
    INT FIRST  = order;  // which kind of points to be smoothed firstly?
    INT SECOND = -order; // which kind of points to be smoothed secondly?

    // auxiliary array(nc*1 vector)
    REAL *vec_tmp = NULL;

    vec_tmp = (REAL *)fasp_mem_calloc(nc,sizeof(REAL));

    if (nc == 1) {
        // deal with the points marked FIRST
        for (point = 0; point < ngrid; point ++) {
            if (mark[point] == FIRST) {
                rhs = b_val[point];
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = point + width;
                    if (width < 0) {
                        if (column >= 0) {
                            rhs -= offdiag[band][column]*u_val[column];
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            rhs -= offdiag[band][point]*u_val[column];
                        }
                    }
                } // end for band

                // zero-diagonal should be tested previously
                u_val[point] = one_minus_weight*u_val[point] +
                    weight*(rhs / diag[point]);

            } // end if (mark[point] == FIRST)
        } // end for point

        // deal with the points marked SECOND
        for (point = 0; point < ngrid; point ++) {
            if (mark[point] == SECOND) {
                rhs = b_val[point];
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = point + width;
                    if (width < 0) {
                        if (column >= 0) {
                            rhs -= offdiag[band][column]*u_val[column];
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            rhs -= offdiag[band][point]*u_val[column];
                        }
                    }
                } // end for band

                // zero-diagonal should be tested previously
                u_val[point] = rhs / diag[point];
            } // end if (mark[point] == SECOND)
        } // end for point

    } // end if (nc == 1)

    else if (nc > 1) {
        // deal with the blocks marked FIRST
        for (block = 0; block < ngrid; block ++) {
            if (mark[block] == FIRST) {
                ncb = nc*block;
                for (point = 0; point < nc; point ++) {
                    vec_tmp[point] = b_val[ncb+point];
                }
                start_DATA = nc2*block;
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = block + width;
                    if (width < 0) {
                        if (column >= 0) {
                            start_data = nc2*column;
                            start_vecu = nc*column;
                            blkcontr2( start_data, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            start_vecu = nc*column;
                            blkcontr2( start_DATA, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                } // end for band

                // subblock smoothing
                aAxpby(weight, one_minus_weight, nc,
                       diaginv+start_DATA, vec_tmp, u_val+nc*block);
            } // end if (mark[block] == FIRST)

        } // end for block

        // deal with the blocks marked SECOND
        for (block = 0; block < ngrid; block ++) {
            if (mark[block] == SECOND) {
                ncb = nc*block;
                for (point = 0; point < nc; point ++) {
                    vec_tmp[point] = b_val[ncb+point];
                }
                start_DATA = nc2*block;
                for (band = 0; band < nband; band ++) {
                    width  = offsets[band];
                    column = block + width;
                    if (width < 0) {
                        if (column >= 0) {
                            start_data = nc2*column;
                            start_vecu = nc*column;
                            blkcontr2( start_data, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                    else { // width > 0
                        if (column < ngrid) {
                            start_vecu = nc*column;
                            blkcontr2( start_DATA, start_vecu, 0, nc,
                                       offdiag[band], u_val, vec_tmp );
                        }
                    }
                } // end for band

                // subblock smoothing
                aAxpby(weight, one_minus_weight, nc,
                       diaginv+start_DATA, vec_tmp, u_val+nc*block);
            } // end if (mark[block] == SECOND)

        } // end for block

    } // end else if (nc > 1)
    else {
        printf("### ERROR: nc is illegal!  [%s:%d]\n", __FILE__, __LINE__);
        return;
    }

    fasp_mem_free(vec_tmp); vec_tmp = NULL;
}

void fasp_generate_diaginv_block (dSTRmat *A,
                                  ivector *neigh,
                                  dvector *diaginv,
                                     ivector *pivot)
{
    // information about A
    const INT nc = A->nc;
    const INT ngrid = A->ngrid;
    const INT nband = A->nband;

    INT *offsets = A->offsets;
    REAL *diag = A->diag;
    REAL **offdiag = A->offdiag;

    // information about neighbors
    INT nneigh;
    if (!neigh) {
        nneigh = 0;
    }
    else {
        nneigh= neigh->row/ngrid;
    }

    // local variable
    INT i, j, k, l, m, n, nbd, p;
    INT count;
    INT block_size;
    INT mem_inv = 0;
    INT mem_pivot = 0;

    // allocation
    REAL *temp = (REAL *)fasp_mem_calloc(((nneigh+1)*nc)*((nneigh+1)*nc)*ngrid, sizeof(REAL));
    INT *tmp = (INT *)fasp_mem_calloc(((nneigh+1)*nc)*ngrid, sizeof(INT));

    // main loop
    for (i=0; i<ngrid; ++i) {
        // count number of neighbors of node i
        count = 1;
        for (l=0; l<nneigh; ++l) {
            if (neigh->val[i*nneigh+l] >= 0) count++ ;
        }

        // prepare the inverse of diagonal block i
        block_size = count*nc;

        diaginv[i].row = block_size*block_size;
        diaginv[i].val = temp + mem_inv;
        mem_inv += diaginv[i].row;

        pivot[i].row = block_size;
        pivot[i].val = tmp + mem_pivot;
        mem_pivot += pivot[i].row;

        // put the diagonal block corresponding to node i
        for (j=0; j<nc; ++j) {
            for (k=0; k<nc; ++k) {
                diaginv[i].val[j*block_size+k] = diag[i*nc*nc + j*nc + k];
            }
        }

        // put the blocks corresponding to the neighbor of node i
        count = 1;
        for (l=0; l<nneigh; ++l) {
            p = neigh->val[i*nneigh+l];
            if (p >= 0){
                // put the diagonal block corresponding to this neighbor
                for (j=0; j<nc; ++j) {
                    for (k=0; k<nc; ++k) {
                        m = count*nc + j; n = count*nc+k;
                        diaginv[i].val[m*block_size+n] = diag[p*nc*nc+j*nc+k];
                    }
                }

                for (nbd=0; nbd<nband; nbd++) {
                    // put the block corresponding to (i, p)
                    if ( offsets[nbd] == (p-i) ) {
                        for (j=0; j<nc; ++j) {
                            for(k=0; k<nc; ++k) {
                                m = j; n = count*nc + k;
                                diaginv[i].val[m*block_size+n] = offdiag[nbd][(p-MAX(p-i,
0))*nc*nc+j*nc+k];
                            }
                        }
                    }

                    // put the block corresponding to (p, i)
                    if ( offsets[nbd] == (i-p) ) {
                        for (j=0; j<nc; ++j) {
                            for(k=0; k<nc; ++k) {
                                m = count*nc + j; n = k;
                                diaginv[i].val[m*block_size+n] = offdiag[nbd][(i-MAX(i-p,
0))*nc*nc+j*nc+k];
                            }
                        }
                    }
                }
                count++;
            } //end if
        } // end for (l=0; l<nneigh; ++l)

        //fasp_smat_inv(diaginv[i].val, block_size);
        fasp_smat_lu_decomp(diaginv[i].val, pivot[i].val, block_size);

    } // end of main loop
}

void fasp_smoother_dstr_swz (dSTRmat *A,
                             dvector *b,
                             dvector *u,
                             dvector *diaginv,
                             ivector *pivot,
                             ivector *neigh,
                             ivector *order)
{
    // information about A
    const INT ngrid = A->ngrid;
    const INT nc = A->nc;

    // information about neighbors
    INT nneigh;
    if (!neigh) {
        nneigh = 0;
    }
    else {
        nneigh= neigh->row/ngrid;
    }

    // local variable
    INT i, j, k, l, p, ti;

    // work space
    REAL *temp = (REAL *)fasp_mem_calloc(b->row + (nneigh+1)*nc + (nneigh+1)*nc, sizeof(REAL));
    dvector r, e, ri;
    r.row = b->row; r.val = temp;
    e.row = (nneigh+1)*nc; e.val = temp + b->row;
    ri.row = (nneigh+1)*nc; ri.val = temp + b->row + (nneigh+1)*nc;

    // initial residual
    fasp_dvec_cp(b,&r);fasp_blas_dstr_aAxpy(-1.0,A,u->val,r.val);

    // main loop
    if (!order) {
        for (i=0; i<ngrid; ++i) {
            //-----------------------------------------------------
            // right hand side for A_ii e_i = r_i
            // rhs corresponding to node i
            for (j=0; j<nc; ++j) {
                ri.val[j] = r.val[i*nc + j];
            }
            // rhs corrsponding to the neighbors of node i
            k = 1;
            for (l=0; l<nneigh; ++l) {
                p=neigh->val[nneigh*i+l];
                if ( p>=0 ) {
                    for (j=0; j<nc; ++j) {
                        ri.val[k*nc+j] = r.val[p*nc+j];
                    }

                    ++k;
                } // end if
            }

            ri.row = k*nc;
            //----------------------------------------------------
            //----------------------------------------------------
            // solve local problem
            e.row = k*nc;
            //fasp_blas_smat_mxv(diaginv[ti].val, ri.val, k*nc, e.val);
            fasp_smat_lu_solve(diaginv[i].val, ri.val, pivot[i].val, e.val, k*nc);
            //-----------------------------------------------------
            //-----------------------------------------------------
            // update solution
            // solution corresponding to node i
            for (j=0; j<nc; ++j) {
                u->val[i*nc + j] += e.val[j];
            }
            // solution corresponding to the neighbor of node i
            k = 1;
            for (l=0; l<nneigh; ++l) {
                p=neigh->val[nneigh*i+l];
                if ( p>=0 ) {
                    for (j=0; j<nc; ++j) {
                        u->val[p*nc+j] += e.val[k*nc+j];
                    }

                    ++k;
                } // end if
            }
            //-----------------------------------------------------
            //-----------------------------------------------------
            // update residule
            fasp_dvec_cp(b,&r); fasp_blas_dstr_aAxpy(-1.0,A,u->val,r.val);
        }
    }
    else {
        for (i=0; i<ngrid; ++i) {
            ti = order->val[i];
            //------------------------------------------------------
            // right hand side for A_ii e_i = r_i
            // rhs corresponding to node i
            for (j=0; j<nc; ++j) {
                ri.val[j] = r.val[ti*nc + j];
            }
            // rhs corrsponding to the neighbors of node i
            k = 1;
            for (l=0; l<nneigh; ++l) {
                p=neigh->val[nneigh*ti+l];
                if ( p>=0 ) {
                    for (j=0; j<nc; ++j) {
                        ri.val[k*nc+j] = r.val[p*nc+j];
                    }

                    ++k;
                } // end if
            }

            ri.row = k*nc;
            //-----------------------------------------------------
            //-----------------------------------------------------
            // solve local problem
            e.row = k*nc;
            //fasp_blas_smat_mxv(diaginv[ti].val, ri.val, k*nc, e.val);
            fasp_smat_lu_solve(diaginv[ti].val, ri.val, pivot[ti].val, e.val, k*nc);
            //-----------------------------------------------------
            //-----------------------------------------------------
            // update solution
            // solution corresponding to node i
            for (j=0; j<nc; ++j) {
                u->val[ti*nc + j] += e.val[j];
            }
            // solution corresponding to the neighbor of node i
            k = 1;
            for (l=0; l<nneigh; ++l) {
                p=neigh->val[nneigh*ti+l];
                if ( p>=0 ) {
                    for (j=0; j<nc; ++j) {
                        u->val[p*nc+j] += e.val[k*nc+j];
                    }

                    ++k;
                } // end if
            }
            //-----------------------------------------------------
            //-----------------------------------------------------
            // update residule
            fasp_dvec_cp(b,&r); fasp_blas_dstr_aAxpy(-1.0,A,u->val,r.val);
        }
    }// end of main loop
}


/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void blkcontr2 (INT   start_data,
                       INT   start_vecx,
                       INT   start_vecy,
                       INT   nc,
                       REAL *data,
                       REAL *x,
                       REAL *y)
{
    INT i,j,k,m;
    if (start_vecy == 0) {
        for (i = 0; i < nc; i ++) {
            k = start_data + i*nc;
            for (j = 0; j < nc; j ++) {
                y[i] -= data[k+j]*x[start_vecx+j];
            }
        }
    }
    else {
        for (i = 0; i < nc; i ++) {
            k = start_data + i*nc;
            m = start_vecy + i;
            for (j = 0; j < nc; j ++) {
                y[m] -= data[k+j]*x[start_vecx+j];
            }
        }
    }
}

static void aAxpby (REAL  alpha,
                    REAL  beta,
                    INT   size,
                    REAL *A,
                    REAL *x,
                    REAL *y)
{
    INT i,j;
    REAL tmp = 0.0;
    if (alpha == 0) {
        for (i = 0; i < size; i ++) {
            y[i] *= beta;
        }
        return;
    }
    tmp = beta / alpha;
    // y:=(beta/alpha)y
    for (i = 0; i < size; i ++) {
        y[i] *= tmp;
    }
    // y:=y+Ax
    for (i = 0; i < size; i ++) {
        for (j = 0; j < size; j ++) {
            y[i] += A[i*size+j]*x[j];
        }
    }
    // y:=alpha*y
    for (i = 0; i < size; i ++) {
        y[i] *= alpha;
    }
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
u	Pointer to dvector of DOFs
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
                precond * pc,
                const REAL tol,
                const INT MaxIt,
                const SHORT StopType,
                const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving Au=b for BSR matrix.

Parameters

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
u	Pointer to dvector of DOFs
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping

Parameters

MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving Au=b for CSR matrix.

Parameters

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
u	Pointer to dvector of DOFs
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving Au=b for STR matrix.

Parameters

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
u	Pointer to dvector of DOFs
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned BiCGstab method for solving Au=b.

Parameters

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector of right hand side
u	Pointer to dvector of DOFs
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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.

```

#include <math.h>
#include <float.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const INT    m = b->row;

    // local variables
    REAL     n2b,tolb;
    INT      iter=0, stag = 1, moresteps = 1, maxmsteps=1;
    INT      flag, maxstagsteps, half_step=0;
    REAL     absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
    REAL     alpha,beta,omega,rho,rho1,rtv,tt;
    REAL     normr,normr_act,normph,normx,imin;
    REAL     norm_sh,norm_xhalf,normrmin,factor;
    REAL     *x = u->val, *bval=b->val;

    // allocate temp memory (need 10*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
    REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
    REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
    REAL *t = sh+m, *xmin = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    n2b = fasp_blas_darray_norm2(m,r);

    flag = 1;
    fasp_darray_cp(m,x,xmin);
    imin = 0;

    iter = 0;

    tolb = n2b*tol;

    fasp_blas_dcsr_aAxpy(-1.0, A, x, r);
    normr     = fasp_blas_darray_norm2(m,r);
    normr_act = normr;
    relres    = normr/n2b;

    // if initial residual is small, no need to iterate!
    if ( normr <= tolb ) {
        flag = 0;
        iter = 0;
        goto FINISHED;
    }

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);

    // shadow residual rt = r* := r
    fasp_darray_cp(m,r,rt);
    normrmin  = normr;

    rho = 1.0;
    omega = 1.0;
    stag = 0;
    alpha = 0.0;

    moresteps = 0;
    maxmsteps = 10;
    maxstagsteps = 3;

    // loop over maxit iterations (unless convergence or failure)
    for (iter=1;iter <= MaxIt;iter++) {

        rho1 = rho;
        rho  = fasp_blas_darray_dotprod(m,rt,r);

        if ((rho ==0.0 )|| (ABS(rho) >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        if (iter==1) {
            fasp_darray_cp(m,r,p);
        }
        else  {
            beta = (rho/rho1)*(alpha/omega);

            if ((beta == 0)||( ABS(beta) > DBL_MAX )) {
                flag = 4;
                goto FINISHED;
            }

            // p = r + beta * (p - omega * v);
            fasp_blas_darray_axpy(m,-omega,v,p);         //p=p - omega*v
            fasp_blas_darray_axpby(m,1.0, r, beta, p);  //p = 1.0*r +beta*p
        }

        // pp = precond(p) ,ph
        if ( pc != NULL )
            pc->fct(p,ph,pc->data); /* Apply preconditioner */
        // if ph all is infinite then exit need add
        else
            fasp_darray_cp(m,p,ph); /* No preconditioner */

        // v = A*ph
        fasp_blas_dcsr_mxv(A,ph,v);
        rtv = fasp_blas_darray_dotprod(m,rt,v);

        if (( rtv==0.0 )||( ABS(rtv) > DBL_MAX )){
            flag = 4;
            goto FINISHED;
        }

        alpha = rho/rtv;

        if ( ABS(alpha) > DBL_MAX ){
            flag = 4;
            ITS_DIVZERO;
            goto FINISHED;
        }

        normx =  fasp_blas_darray_norm2(m,x);
        normph = fasp_blas_darray_norm2(m,ph);
        if (ABS(alpha)*normph < DBL_EPSILON*normx )
            stag = stag + 1;
        else
            stag = 0;

        // xhalf = x + alpha * ph;        // form the "half" iterate
        // s = r - alpha * v;             // residual associated with xhalf
        fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf);  // z= ax + y
        fasp_blas_darray_axpyz(m, -alpha, v, r, s);
        normr = fasp_blas_darray_norm2(m,s);  // normr = norm(s);
        normr_act = normr;

        // compute reduction factor of residual ||r||
        absres = normr_act;
        factor = absres/absres0;
        fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);

        // check for convergence
        if ((normr <= tolb)||(stag >= maxstagsteps)||moresteps)
        {
            fasp_darray_cp(m,bval,s);
            fasp_blas_dcsr_aAxpy(-1.0,A,xhalf,s);
            normr_act = fasp_blas_darray_norm2(m,s);

            if (normr_act <= tolb) {
                // x = xhalf;
                fasp_darray_cp(m,xhalf,x);    // x = xhalf;
                flag = 0;
                imin = iter - 0.5;
                half_step++;
                if ( PrtLvl >= PRINT_MORE )
                    printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                           flag,stag,imin,half_step);
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps) {
                    // if ~warned
                    flag = 3;
                    fasp_darray_cp(m,xhalf,x);
                    goto FINISHED;
                }
            }
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( normr_act < normrmin ) // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,xhalf,xmin);
            imin = iter - 0.5;
            half_step++;
            if ( PrtLvl >= PRINT_MORE )
                printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                       flag,stag,imin,half_step);
        }

        // sh = precond(s)
        if ( pc != NULL ) {
            pc->fct(s,sh,pc->data); /* Apply preconditioner */
        }
        else
            fasp_darray_cp(m,s,sh); /* No preconditioner */

        // t = A*sh;
        fasp_blas_dcsr_mxv(A,sh,t);
        // tt = t' * t;
        tt = fasp_blas_darray_dotprod(m,t,t);
        if ( (tt == 0) ||( tt >= DBL_MAX ) ) {
            flag = 4;
            goto FINISHED;
        }

        // omega = (t' * s) / tt;
        omega = fasp_blas_darray_dotprod(m,s,t)/tt;
        if ( ABS(omega) > DBL_MAX ) {
            flag = 4;
            goto FINISHED;
        }

        norm_sh = fasp_blas_darray_norm2(m,sh);
        norm_xhalf = fasp_blas_darray_norm2(m,xhalf);

        if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
            stag = stag + 1;
        else
            stag = 0;

        fasp_blas_darray_axpyz(m, omega,sh,xhalf, x);  // x = xhalf + omega * sh;
        fasp_blas_darray_axpyz(m, -omega, t, s, r);    // r = s - omega * t;
        normr = fasp_blas_darray_norm2(m,r);           // normr = norm(r);
        normr_act = normr;

        // check for convergence
        if ( (normr <= tolb) || (stag >= maxstagsteps) || moresteps )
        {
            fasp_darray_cp(m,bval,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,x,r);
            normr_act = fasp_blas_darray_norm2(m,r);
            if ( normr_act <= tolb ) {
                flag = 0;
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if ( moresteps >= maxmsteps ) {
                    flag = 3;
                    goto FINISHED;
                }
            }
        }

        // update minimal norm quantities
        if ( normr_act < normrmin ) {
            normrmin = normr_act;
            fasp_darray_cp(m,x,xmin);
            imin = iter;
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

        absres0 = absres;
    }   // for iter = 1 :  maxit

FINISHED:  // finish iterative method
    // returned solution is first with minimal residual
    if (flag == 0)
        relres = normr_act / n2b;
    else {
        fasp_darray_cp(m, bval,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,xmin,r);
        normr = fasp_blas_darray_norm2(m,r);

        if ( normr <= normr_act) {
            fasp_darray_cp(m, xmin, x);
            iter = imin;
            relres = normr/n2b;
        }
        else {
            relres = normr_act/n2b;
        }
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    if ( PrtLvl >= PRINT_MORE )
        printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
               flag,stag,imin,half_step);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT    m = b->row;

    // local variables
    REAL     n2b,tolb;
    INT      iter=0, stag = 1, moresteps = 1, maxmsteps=1;
    INT      flag, maxstagsteps, half_step=0;
    REAL     absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
    REAL     alpha,beta,omega,rho,rho1,rtv,tt;
    REAL     normr,normr_act,normph,normx,imin;
    REAL     norm_sh,norm_xhalf,normrmin,factor;
    REAL     *x = u->val, *bval=b->val;

    // allocate temp memory (need 10*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
    REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
    REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
    REAL *t = sh+m, *xmin = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif
```

```

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    n2b = fasp_blas_darray_norm2(m,r);

    flag = 1;
    fasp_darray_cp(m,x,xmin);
    imin = 0;

    iter = 0;

    tolb = n2b*tol;

    fasp_blas_dbsr_aAxpy(-1.0, A, x, r);
    normr     = fasp_blas_darray_norm2(m,r);
    normr_act = normr;
    relres    = normr/n2b;

    // if initial residual is small, no need to iterate!
    if ( normr <= tolb ) {
        flag = 0;
        iter = 0;
        goto FINISHED;
    }

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);

    // shadow residual rt = r* := r
    fasp_darray_cp(m,r,rt);
    normrmin  = normr;

    rho = 1.0;
    omega = 1.0;
    stag = 0;
    alpha = 0.0;

    moresteps = 0;
    maxmsteps = 10;
    maxstagsteps = 3;

    // loop over maxit iterations (unless convergence or failure)
    for (iter=1;iter <= MaxIt;iter++) {

        rho1 = rho;
        rho  = fasp_blas_darray_dotprod(m,rt,r);

        if ((rho ==0.0 )|| (ABS(rho) >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        if (iter==1) {
            fasp_darray_cp(m,r,p);
        }
        else  {
            beta = (rho/rho1)*(alpha/omega);

            if ((beta == 0)||( ABS(beta) > DBL_MAX )) {
                flag = 4;
                goto FINISHED;
            }

            // p = r + beta * (p - omega * v);
            fasp_blas_darray_axpy(m,-omega,v,p);        //p=p - omega*v
            fasp_blas_darray_axpby(m,1.0, r, beta, p);  //p = 1.0*r +beta*p
        }

        // pp = precond(p) ,ph
        if ( pc != NULL )
            pc->fct(p,ph,pc->data); /* Apply preconditioner */
        // if ph all is infinite then exit need add
        else
            fasp_darray_cp(m,p,ph); /* No preconditioner */

        // v = A*ph
        fasp_blas_dbsr_mxv(A,ph,v);
        rtv = fasp_blas_darray_dotprod(m,rt,v);

        if (( rtv==0.0 )||( ABS(rtv) > DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        alpha = rho/rtv;

        if ( ABS(alpha) > DBL_MAX ) {
            flag = 4;
            ITS_DIVZERO;
            goto FINISHED;
        }

        normx =  fasp_blas_darray_norm2(m,x);
        normph = fasp_blas_darray_norm2(m,ph);
        if (ABS(alpha)*normph < DBL_EPSILON*normx )
            stag = stag + 1;
        else
            stag = 0;

        // xhalf = x + alpha * ph;        // form the "half" iterate
        // s = r - alpha * v;             // residual associated with xhalf
        fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf);  // z= ax + y
        fasp_blas_darray_axpyz(m, -alpha, v, r, s);
        normr = fasp_blas_darray_norm2(m,s);  // normr = norm(s);
        normr_act = normr;

        // compute reduction factor of residual ||r||
        absres = normr_act;
        factor = absres/absres0;
        fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);

        // check for convergence
        if ((normr <= tolb)||(stag >= maxstagsteps)||moresteps)
        {
            fasp_darray_cp(m,bval,s);
            fasp_blas_dbsr_aAxpy(-1.0,A,xhalf,s);
            normr_act = fasp_blas_darray_norm2(m,s);

            if (normr_act <= tolb) {
                // x = xhalf;
                fasp_darray_cp(m,xhalf,x);    // x = xhalf;
                flag = 0;
                imin = iter - 0.5;
                half_step++;
                if ( PrtLvl >= PRINT_MORE )
                    printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                           flag,stag,imin,half_step);
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0))  stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps){
                    // if ~warned
                    flag = 3;
                    fasp_darray_cp(m,xhalf,x);
                    goto FINISHED;
                }
            }
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( normr_act < normrmin ) // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,xhalf,xmin);
            imin = iter - 0.5;
            half_step++;
            if ( PrtLvl >= PRINT_MORE )
                printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                       flag,stag,imin,half_step);
        }

        // sh = precond(s)
        if ( pc != NULL ) {
            pc->fct(s,sh,pc->data); /* Apply preconditioner */
        }
        else
            fasp_darray_cp(m,s,sh); /* No preconditioner */

        // t = A*sh;
        fasp_blas_dbsr_mxv(A,sh,t);
        // tt = t' * t;
        tt = fasp_blas_darray_dotprod(m,t,t);
        if ( (tt == 0) ||( tt >= DBL_MAX ) ) {
            flag = 4;
            goto FINISHED;
        }

        // omega = (t' * s) / tt;
        omega = fasp_blas_darray_dotprod(m,s,t)/tt;
        if ( ABS(omega) > DBL_MAX ) {
            flag = 4;
            goto FINISHED;
        }

        norm_sh = fasp_blas_darray_norm2(m,sh);
        norm_xhalf = fasp_blas_darray_norm2(m,xhalf);

        if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
            stag = stag + 1;
        else
            stag = 0;

        fasp_blas_darray_axpyz(m, omega,sh,xhalf, x);  // x = xhalf + omega * sh;
        fasp_blas_darray_axpyz(m, -omega, t, s, r);    // r = s - omega * t;
        normr = fasp_blas_darray_norm2(m,r);           // normr = norm(r);
        normr_act = normr;

        // check for convergence
        if ( (normr <= tolb) || (stag >= maxstagsteps) || moresteps )
        {
            fasp_darray_cp(m,bval,r);
            fasp_blas_dbsr_aAxpy(-1.0,A,x,r);
            normr_act = fasp_blas_darray_norm2(m,r);
            if ( normr_act <= tolb ) {
                flag = 0;
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if ( moresteps >= maxmsteps ) {
                    flag = 3;
                    goto FINISHED;
                }
            }
        }

        // update minimal norm quantities
        if ( normr_act < normrmin ) {
            normrmin = normr_act;
            fasp_darray_cp(m,x,xmin);
            imin = iter;
        }

        if ( stag >= maxstagsteps )
        {
            flag = 3;
            goto FINISHED;
        }

        if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

        absres0 = absres;
    }   // for iter = 1 :  maxit

FINISHED:  // finish iterative method
    // returned solution is first with minimal residual
    if (flag == 0)
        relres = normr_act / n2b;
    else {
        fasp_darray_cp(m, bval,r);
        fasp_blas_dbsr_aAxpy(-1.0,A,xmin,r);
        normr = fasp_blas_darray_norm2(m,r);

        if ( normr <= normr_act) {
            fasp_darray_cp(m, xmin,x);
            iter = imin;
            relres = normr/n2b;
        }
        else {
            relres = normr_act/n2b;
        }
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    if ( PrtLvl >= PRINT_MORE )
        printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
               flag,stag,imin,half_step);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT    m = b->row;

    // local variables
    REAL     n2b,tolb;
    INT      iter=0, stag = 1, moresteps = 1, maxmsteps=1;
    INT      flag, maxstagsteps, half_step=0;
    REAL     absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
    REAL     alpha,beta,omega,rho,rho1,rtv,tt;
    REAL     normr,normr_act,normph,normx,imin;
    REAL     norm_sh,norm_xhalf,normrmin,factor;
    REAL     *x = u->val, *bval=b->val;

    // allocate temp memory (need 10*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
    REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
    REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
    REAL *t = sh+m, *xmin = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    n2b = fasp_blas_darray_norm2(m,r);

    flag = 1;
    fasp_darray_cp(m,x,xmin);
    imin = 0;

    iter = 0;

    tolb = n2b*tol;

    fasp_blas_dblc_aAxpy(-1.0, A, x, r);
    normr     = fasp_blas_darray_norm2(m,r);
    normr_act = normr;
    relres    = normr/n2b;

    // if initial residual is small, no need to iterate!
    if ( normr <= tolb ) {
        flag = 0;
        iter = 0;
        goto FINISHED;
    }

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);

    // shadow residual rt = r* := r
    fasp_darray_cp(m,r,rt);
    normrmin  = normr;

    rho = 1.0;
    omega = 1.0;
    stag = 0;
    alpha = 0.0;

    moresteps = 0;
    maxmsteps = 10;
    maxstagsteps = 3;

    // loop over maxit iterations (unless convergence or failure)
    for (iter=1;iter <= MaxIt;iter++) {

        rho1 = rho;
        rho  = fasp_blas_darray_dotprod(m,rt,r);

        if ((rho ==0.0 )|| (ABS(rho) >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        if (iter==1) {
            fasp_darray_cp(m,r,p);
        }
        else  {
            beta = (rho/rho1)*(alpha/omega);

            if ((beta == 0)||( ABS(beta) > DBL_MAX )) {
                flag = 4;
                goto FINISHED;
            }

            // p = r + beta * (p - omega * v);
            fasp_blas_darray_axpy(m,-omega,v,p);         //p=p - omega*v
            fasp_blas_darray_axpby(m,1.0, r, beta, p);   //p = 1.0*r +beta*p
        }

        // pp = precond(p) ,ph
        if ( pc != NULL )
            pc->fct(p,ph,pc->data); /* Apply preconditioner */
        // if ph all is infinite then exit need add
        else
            fasp_darray_cp(m,p,ph); /* No preconditioner */

        // v = A*ph
        fasp_blas_dblc_mxv(A,ph,v);
        rtv = fasp_blas_darray_dotprod(m,rt,v);

        if (( rtv==0.0 )||( ABS(rtv) > DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        alpha = rho/rtv;

        if ( ABS(alpha) > DBL_MAX ) {
            flag = 4;
            ITS_DIVZERO;
            goto FINISHED;
        }

        normx =  fasp_blas_darray_norm2(m,x);
        normph = fasp_blas_darray_norm2(m,ph);
        if (ABS(alpha)*normph < DBL_EPSILON*normx )
            stag = stag + 1;
        else
            stag = 0;

        // xhalf = x + alpha * ph;        // form the "half" iterate
        // s = r - alpha * v;              // residual associated with xhalf
        fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf);  // z= ax + y
        fasp_blas_darray_axpyz(m, -alpha, v, r, s);
        normr = fasp_blas_darray_norm2(m,s);  // normr = norm(s);
        normr_act = normr;

        // compute reduction factor of residual ||r||
        absres = normr_act;
        factor = absres/absres0;
        fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);

        // check for convergence
        if ((normr <= tolb)||(stag >= maxstagsteps)||moresteps)
        {
            fasp_darray_cp(m,bval,s);
            fasp_blas_dblc_aAxpy(-1.0,A,xhalf,s);
            normr_act = fasp_blas_darray_norm2(m,s);

            if (normr_act <= tolb) {
                // x = xhalf;
                fasp_darray_cp(m,xhalf,x);    // x = xhalf;
                flag = 0;
                imin = iter - 0.5;
                half_step++;
                if ( PrtLvl >= PRINT_MORE )
                    printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                           flag,stag,imin,half_step);
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0))  stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps) {
                    // if ~warned
                    flag = 3;
                    fasp_darray_cp(m,xhalf,x);
                    goto FINISHED;
                }
            }
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( normr_act < normrmin ) // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,xhalf,xmin);
            imin = iter - 0.5;
            half_step++;
            if ( PrtLvl >= PRINT_MORE )
                printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                       flag,stag,imin,half_step);
        }

        // sh = precond(s)
        if ( pc != NULL ) {
            pc->fct(s,sh,pc->data); /* Apply preconditioner */
        }
        else
            fasp_darray_cp(m,s,sh); /* No preconditioner */

        // t = A*sh;
        fasp_blas_dblc_mxv(A,sh,t);
        // tt = t' * t;
        tt = fasp_blas_darray_dotprod(m,t,t);
        if ( (tt == 0) ||( tt >= DBL_MAX ) ) {
            flag = 4;
            goto FINISHED;
        }

        // omega = (t' * s) / tt;
        omega = fasp_blas_darray_dotprod(m,s,t)/tt;
        if ( ABS(omega) > DBL_MAX ) {
            flag = 4;
            goto FINISHED;
        }

        norm_sh = fasp_blas_darray_norm2(m,sh);
        norm_xhalf = fasp_blas_darray_norm2(m,xhalf);

        if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
            stag = stag + 1;
        else
            stag = 0;

        fasp_blas_darray_axpyz(m, omega,sh,xhalf, x);  // x = xhalf + omega * sh;
        fasp_blas_darray_axpyz(m, -omega, t, s, r);    // r = s - omega * t;
        normr = fasp_blas_darray_norm2(m,r);           // normr = norm(r);
        normr_act = normr;

        // check for convergence
        if ( (normr <= tolb) || (stag >= maxstagsteps) || moresteps )
        {
            fasp_darray_cp(m,bval,r);
            fasp_blas_dblc_aAxpy(-1.0,A,x,r);
            normr_act = fasp_blas_darray_norm2(m,r);
            if ( normr_act <= tolb ) {
                flag = 0;
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if ( moresteps >= maxmsteps ) {
                    flag = 3;
                    goto FINISHED;
                }
            }
        }

        // update minimal norm quantities
        if ( normr_act < normrmin ) {
            normrmin = normr_act;
            fasp_darray_cp(m,x,xmin);
            imin = iter;
        }

        if ( stag >= maxstagsteps )
        {
            flag = 3;
            goto FINISHED;
        }

        if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

        absres0 = absres;
    }   // for iter = 1 :  maxit

FINISHED:  // finish iterative method
    // returned solution is first with minimal residual
    if (flag == 0)
        relres = normr_act / n2b;
    else {
        fasp_darray_cp(m, bval,r);
        fasp_blas_dblc_aAxpy(-1.0,A,xmin,r);
        normr = fasp_blas_darray_norm2(m,r);

        if ( normr <= normr_act) {
            fasp_darray_cp(m, xmin,x);
            iter = imin;
            relres = normr/n2b;
        }
        else {
            relres = normr_act/n2b;
        }
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    if ( PrtLvl >= PRINT_MORE )
        printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
               flag,stag,imin,half_step);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
```

```
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT    m = b->row;

    // local variables
    REAL     n2b,tolb;
    INT      iter=0, stag = 1, moresteps = 1, maxmsteps=1;
    INT      flag, maxstagsteps, half_step=0;
    REAL     absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
    REAL     alpha,beta,omega,rho,rho1,rtv,tt;
    REAL     normr,normr_act,normph,normx,imin;
    REAL     norm_sh,norm_xhalf,normrmin,factor;
    REAL     *x = u->val, *bval=b->val;

    // allocate temp memory (need 10*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
    REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
    REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
    REAL *t = sh+m, *xmin = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    n2b = fasp_blas_darray_norm2(m,r);

    flag = 1;
    fasp_darray_cp(m,x,xmin);
    imin = 0;

    iter = 0;

    tolb = n2b*tol;

    fasp_blas_dstr_aAxpy(-1.0, A, x, r);
    normr     = fasp_blas_darray_norm2(m,r);
    normr_act = normr;
    relres    = normr/n2b;

    // if initial residual is small, no need to iterate!
    if ( normr <= tolb ) {
        flag = 0;
        iter = 0;
        goto FINISHED;
    }

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);

    // shadow residual rt = r* := r
    fasp_darray_cp(m,r,rt);
    normrmin  = normr;

    rho = 1.0;
    omega = 1.0;
    stag = 0;
    alpha = 0.0;

    moresteps = 0;
    maxmsteps = 10;
    maxstagsteps = 3;

    // loop over maxit iterations (unless convergence or failure)
    for (iter=1;iter <= MaxIt;iter++) {

        rho1 = rho;
        rho  = fasp_blas_darray_dotprod(m,rt,r);

        if ((rho ==0.0 )|| (ABS(rho) >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        if (iter==1) {
            fasp_darray_cp(m,r,p);
        }
        else  {
            beta = (rho/rho1)*(alpha/omega);

            if ((beta == 0)||( ABS(beta) > DBL_MAX )) {
                flag = 4;
                goto FINISHED;
            }

            // p = r + beta * (p - omega * v);
            fasp_blas_darray_axpy(m,-omega,v,p);         //p=p - omega*v
            fasp_blas_darray_axpby(m,1.0, r, beta, p);   //p = 1.0*r +beta*p
        }

        // pp = precond(p) ,ph
        if ( pc != NULL )
            pc->fct(p,ph,pc->data); /* Apply preconditioner */
        // if ph all is infinite then exit need add
        else
            fasp_darray_cp(m,p,ph); /* No preconditioner */

        // v = A*ph
        fasp_blas_dstr_mxv(A,ph,v);
        rtv = fasp_blas_darray_dotprod(m,rt,v);

        if (( rtv==0.0 )||( ABS(rtv) > DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        alpha = rho/rtv;

        if ( ABS(alpha) > DBL_MAX ) {
            flag = 4;
            ITS_DIVZERO;
            goto FINISHED;
        }

        normx =  fasp_blas_darray_norm2(m,x);
        normph = fasp_blas_darray_norm2(m,ph);
        if (ABS(alpha)*normph < DBL_EPSILON*normx )
            stag = stag + 1;
        else
            stag = 0;

        // xhalf = x + alpha * ph;        // form the "half" iterate
        // s = r - alpha * v;             // residual associated with xhalf
        fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf);  // z= ax + y
        fasp_blas_darray_axpyz(m, -alpha, v, r, s);
        normr = fasp_blas_darray_norm2(m,s);  // normr = norm(s);
        normr_act = normr;

        // compute reduction factor of residual ||r||
        absres = normr_act;
        factor = absres/absres0;
        fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);

        // check for convergence
        if ((normr <= tolb)||(stag >= maxstagsteps)||moresteps)
        {
            fasp_darray_cp(m,bval,s);
            fasp_blas_dstr_aAxpy(-1.0,A,xhalf,s);
            normr_act = fasp_blas_darray_norm2(m,s);

            if (normr_act <= tolb){
                // x = xhalf;
                fasp_darray_cp(m,xhalf,x);     // x = xhalf;
                flag = 0;
                imin = iter - 0.5;
                half_step++;
                if ( PrtLvl >= PRINT_MORE )
                    printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                           flag,stag,imin,half_step);
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0))  stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps){
                    // if ~warned
                    flag = 3;
                    fasp_darray_cp(m,xhalf,x);
                    goto FINISHED;
                }
            }
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( normr_act < normrmin )      // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,xhalf,xmin);
            imin = iter - 0.5;
            half_step++;
            if ( PrtLvl >= PRINT_MORE )
                printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                       flag,stag,imin,half_step);
        }

        // sh = precond(s)
        if ( pc != NULL ) {
            pc->fct(s,sh,pc->data); /* Apply preconditioner */
        }
        else
            fasp_darray_cp(m,s,sh); /* No preconditioner */

        // t = A*sh;
        fasp_blas_dstr_mxv(A,sh,t);
        // tt = t' * t;
        tt = fasp_blas_darray_dotprod(m,t,t);
        if ( (tt == 0) ||( tt >= DBL_MAX ) ) {
            flag = 4;
            goto FINISHED;
        }

        // omega = (t' * s) / tt;
        omega = fasp_blas_darray_dotprod(m,s,t)/tt;
        if ( ABS(omega) > DBL_MAX ) {
            flag = 4;
            goto FINISHED;
        }

        norm_sh = fasp_blas_darray_norm2(m,sh);
        norm_xhalf = fasp_blas_darray_norm2(m,xhalf);

        if ( ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
            stag = stag + 1;
        else
            stag = 0;

        fasp_blas_darray_axpyz(m, omega,sh,xhalf, x);  // x = xhalf + omega * sh;
        fasp_blas_darray_axpyz(m, -omega, t, s, r);    // r = s - omega * t;
        normr = fasp_blas_darray_norm2(m,r);           // normr = norm(r);
        normr_act = normr;

        // check for convergence
        if ( (normr <= tolb) || (stag >= maxstagsteps) || moresteps )
        {
            fasp_darray_cp(m,bval,r);
            fasp_blas_dstr_aAxpy(-1.0,A,x,r);
            normr_act = fasp_blas_darray_norm2(m,r);
            if ( normr_act <= tolb ) {
                flag = 0;
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if ( moresteps >= maxmsteps ) {
                    flag = 3;
                    goto FINISHED;
                }
            }
        }

        // update minimal norm quantities
        if ( normr_act < normrmin ) {
            normrmin = normr_act;
            fasp_darray_cp(m,x,xmin);
            imin = iter;
        }

        if ( stag >= maxstagsteps )
        {
            flag = 3;
            goto FINISHED;
        }

        if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

        absres0 = absres;
    }   // for iter = 1 :  maxit

FINISHED:  // finish iterative method
    // returned solution is first with minimal residual
    if (flag == 0)
        relres = normr_act / n2b;
    else {
        fasp_darray_cp(m, bval,r);
        fasp_blas_dstr_aAxpy(-1.0,A,xmin,r);
        normr = fasp_blas_darray_norm2(m,r);

        if ( normr <= normr_act ) {
            fasp_darray_cp(m, xmin,x);
            iter = imin;
            relres = normr/n2b;
        }
        else {
            relres = normr_act/n2b;
        }
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    if ( PrtLvl >= PRINT_MORE )
        printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
               flag,stag,imin,half_step);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT     m = b->row;

    // local variables
    REAL     n2b,tolb;
    INT      iter=0, stag = 1, moresteps = 1, maxmsteps=1;
    INT      flag, maxstagsteps, half_step=0;
    REAL     absres0 = BIGREAL, absres = BIGREAL, relres = BIGREAL;
    REAL     alpha,beta,omega,rho,rho1,rtv,tt;
    REAL     normr,normr_act,normph,normx,imin;
    REAL     norm_sh,norm_xhalf,normrmin,factor;
    REAL     *x = u->val, *bval=b->val;

    // allocate temp memory (need 10*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(10*m,sizeof(REAL));
    REAL *r=work, *rt=r+m, *p=rt+m, *v=p+m;
    REAL *ph=v+m, *xhalf=ph+m, *s=xhalf+m, *sh=s+m;
    REAL *t = sh+m, *xmin = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling BiCGstab solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    n2b = fasp_blas_darray_norm2(m,r);

    flag = 1;
    fasp_darray_cp(m,x,xmin);
    imin = 0;

    iter = 0;

    tolb = n2b*tol;

    // r = b-A*x
    mf->fct(mf->data, x, r);
    fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
    normr = fasp_blas_darray_norm2(m,r);
    normr_act = normr;

    relres = normr/n2b;
    // if initial residual is small, no need to iterate!
    if (normr <= tolb) {
        flag =0;
        iter =0;
        goto FINISHED;
    }

    // output iteration information if needed

    fasp_itinfo(PrtLvl,StopType,iter,relres,n2b,0.0);

    // shadow residual rt = r* := r
    fasp_darray_cp(m,r,rt);
    normrmin  = normr;

    rho = 1.0;
    omega = 1.0;
    stag = 0;
    alpha =0.0;

    moresteps = 0;
    maxmsteps = 10;
    maxstagsteps = 3;

    // loop over maxit iterations (unless convergence or failure)
    for (iter=1;iter <= MaxIt;iter++) {

        rho1 = rho;
        rho  = fasp_blas_darray_dotprod(m,rt,r);

        if ((rho ==0.0 )|| (ABS(rho) >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        if (iter==1) {
            fasp_darray_cp(m,r,p);
        }
        else  {
            beta = (rho/rho1)*(alpha/omega);

            if ((beta == 0)||( ABS(beta) > DBL_MAX )) {
                flag = 4;
                goto FINISHED;
            }

            // p = r + beta * (p - omega * v);
            fasp_blas_darray_axpy(m,-omega,v,p);        //p=p - omega*v
            fasp_blas_darray_axpby(m,1.0, r, beta, p);  //p = 1.0*r +beta*p
        }

        // pp = precond(p) ,ph
        if ( pc != NULL )
            pc->fct(p,ph,pc->data); /* Apply preconditioner */
        // if ph all is infinite then exit need add
        else
            fasp_darray_cp(m,p,ph); /* No preconditioner */

        // v = A*ph
        mf->fct(mf->data, ph, v);
        rtv = fasp_blas_darray_dotprod(m,rt,v);

        if (( rtv==0.0 )||( ABS(rtv) > DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        alpha = rho/rtv;

        if ( ABS(alpha) > DBL_MAX ) {
            flag = 4;
            ITS_DIVZERO;
            goto FINISHED;
        }

        normx =  fasp_blas_darray_norm2(m,x);
        normph = fasp_blas_darray_norm2(m,ph);
        if (ABS(alpha)*normph < DBL_EPSILON*normx )
            stag = stag + 1;
        else
            stag = 0;

        // xhalf = x + alpha * ph;        // form the "half" iterate
        // s = r - alpha * v;             // residual associated with xhalf
        fasp_blas_darray_axpyz(m, alpha, ph, x , xhalf);  // z= ax + y
        fasp_blas_darray_axpyz(m, -alpha, v, r, s);
        normr = fasp_blas_darray_norm2(m,s);  // normr = norm(s);
        normr_act = normr;

        // compute reduction factor of residual ||r||
        absres = normr_act;
        factor = absres/absres0;
        fasp_itinfo(PrtLvl,StopType,iter,normr_act/n2b,absres,factor);

        // check for convergence
        if ((normr <= tolb)||(stag >= maxstagsteps)||moresteps)
        {
            // s = b-A*xhalf
            mf->fct(mf->data, xhalf, s);
            fasp_blas_darray_axpby(m, 1.0, bval, -1.0, s);
            normr_act = fasp_blas_darray_norm2(m,s);

            if (normr_act <= tolb){
                // x = xhalf;
                fasp_darray_cp(m,xhalf,x);    // x = xhalf;
                flag = 0;
                imin = iter - 0.5;
                half_step++;
                if ( PrtLvl >= PRINT_MORE )
                    printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                           flag,stag,imin,half_step);
                goto FINISHED;
            }
            else {
                if ((stag >= maxstagsteps) && (moresteps == 0))  stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps){
                    //      if ~warned
                    flag = 3;
                    fasp_darray_cp(m,xhalf,x);
                    goto FINISHED;
                }
            }
        }

        if ( stag >= maxstagsteps ) {
            flag = 3;
            goto FINISHED;
        }

        if ( normr_act < normrmin ) // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,xhalf,xmin);
            imin = iter - 0.5;
            half_step++;
            if ( PrtLvl >= PRINT_MORE )
                printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
                       flag,stag,imin,half_step);
        }

        // sh = precond(s)
        if ( pc != NULL ){
            pc->fct(s,sh,pc->data); /* Apply preconditioner */
            //if all is finite
        }
        else
            fasp_darray_cp(m,s,sh); /* No preconditioner */

        // t = A*sh;
        mf->fct(mf->data, sh, t);
        // tt = t' * t;
        tt = fasp_blas_darray_dotprod(m,t,t);
        if ((tt == 0) ||( tt >= DBL_MAX )) {
            flag = 4;
            goto FINISHED;
        }

        // omega = (t' * s) / tt;
        omega = fasp_blas_darray_dotprod(m,s,t)/tt;
        if (ABS(omega) > DBL_MAX )
        {
            flag = 4;
            goto FINISHED;
        }

        norm_sh = fasp_blas_darray_norm2(m,sh);
        norm_xhalf = fasp_blas_darray_norm2(m,xhalf);

        if (ABS(omega)*norm_sh < DBL_EPSILON*norm_xhalf )
            stag = stag + 1;
        else
            stag = 0;

        fasp_blas_darray_axpyz(m, omega,sh,xhalf, x);  //  x = xhalf + omega * sh;
        fasp_blas_darray_axpyz(m, -omega, t, s, r);    //  r = s - omega * t;
        normr = fasp_blas_darray_norm2(m,r);           //normr = norm(r);
        normr_act = normr;

        // check for convergence
        if ( (normr <= tolb)||(stag >= maxstagsteps)||moresteps )
        {
            // normr_act = norm(r);
            // r = b-A*x
            mf->fct(mf->data, x, r);
            fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
            normr_act = fasp_blas_darray_norm2(m,r);
            if (normr_act <= tolb)
            {
                flag = 0;
                goto FINISHED;
            }
            else
            {
                if ((stag >= maxstagsteps) && (moresteps == 0)) stag = 0;

                moresteps = moresteps + 1;
                if (moresteps >= maxmsteps)
                {
                    flag = 3;
                    goto FINISHED;
                }
            }
        }

        if (normr_act < normrmin) // update minimal norm quantities
        {
            normrmin = normr_act;
            fasp_darray_cp(m,x,xmin);
            imin = iter;
        }

        if (stag >= maxstagsteps)
        {
            flag = 3;
            goto FINISHED;
        }

        if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

        absres0 = absres;
    }   // for iter = 1 :  maxit

FINISHED:  // finish iterative method
    // returned solution is first with minimal residual
    if (flag == 0)
        relres = normr_act / n2b;
    else {
        // r = b-A*xmin
        mf->fct(mf->data, xmin, r);
        fasp_blas_darray_axpby(m, 1.0, bval, -1.0, r);
        normr = fasp_blas_darray_norm2(m,r);

        if ( normr <= normr_act ) {
            fasp_darray_cp(m, xmin,x);
            iter = imin;
            relres = normr/n2b;
        }
        else {
            relres = normr_act/n2b;
        }
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    if ( PrtLvl >= PRINT_MORE )
        printf("Flag = %d Stag = %d Itermin = %.1f Half_step = %d\n",
               flag,stag,imin,half_step);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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, ... , p_k};
- prepare for next iteration;

- print the result of k-th iteration; END FOR

Convergence check: norm(r)/norm(b) < tol
Stagnation check:

- IF norm(alpha∗p_k)/norm(x_{k+1}) < 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 norm(r_{k+1})/norm(b) < 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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving Au=b.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving Au=b.

Parameters

A	Pointer to dBSRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving Au=b.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

Preconditioned conjugate gradient method for solving Au=b.

Parameters

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned conjugate gradient (CG) method for solving Au=b.

Parameters

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;

    // allocate temp memory (need 4*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m;

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

    if ( pc != NULL )
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol  || absres0 < 1e-12*tol  ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t = A*p
        fasp_blas_dcsr_mxv(A,p,t);

        // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            ITS_DIVZERO; goto FINISHED;
        }

        // u_k = u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k = r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute norm of residual
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reduction factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
            normuinf = fasp_blas_darray_norminf(m, u->val);
            if ( normuinf <= sol_inf_tol ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if stagnated, try to restart
            normu = fasp_blas_darray_norm2(m, u->val);

            // compute relative difference
            reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
            if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(reldiff,relres);
                    ITS_RESTART;
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

                // compute residual norms
                switch ( StopType ) {
                    case STOP_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        // z = B(r)
                        if ( pc != NULL )
                            pc->fct(r,z,pc->data); /* Apply preconditioner */
                        else
                            fasp_darray_cp(m,r,z); /* No preconditioner */
                        absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normu;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p,0.0);
                    ++stag;
                }

            } // end of stagnation check!

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            REAL updated_relres = relres;

            // compute true residual r = b - Ax and update residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residual norms
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            // check convergence
            if ( relres < tol ) break;

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(updated_relres); ITS_REALRES(relres);
            }

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p,0.0);
            ++more_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;

    // allocate temp memory (need 4*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dbsr_aAxpy(-1.0,A,u->val,r);

    if ( pc != NULL )
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t = A*p
        fasp_blas_dbsr_mxv(A,p,t);

        // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            ITS_DIVZERO; goto FINISHED;
        }

        // u_k = u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k = r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute norm of residual
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reduction factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
            normuinf = fasp_blas_darray_norminf(m, u->val);
            if ( normuinf <= sol_inf_tol ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if stagnated, try to restart
            normu = fasp_blas_darray_norm2(m, u->val);

            // compute relative difference
            reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
            if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(reldiff,relres);
                    ITS_RESTART;
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dbsr_aAxpy(-1.0,A,u->val,r);

                // compute residual norms
                switch ( StopType ) {
                    case STOP_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        // z = B(r)
                        if ( pc != NULL )
                            pc->fct(r,z,pc->data); /* Apply preconditioner */
                        else
                            fasp_darray_cp(m,r,z); /* No preconditioner */
                        absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normu;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p,0.0);
                    ++stag;
                }

            } // end of stagnation check!

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            REAL updated_relres = relres;

            // compute true residual r = b - Ax and update residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dbsr_aAxpy(-1.0,A,u->val,r);

            // compute residual norms
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            // check convergence
            if ( relres < tol ) break;

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(updated_relres); ITS_REALRES(relres);
            }

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p,0.0);
            ++more_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.
```

```

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;

    // allocate temp memory (need 4*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

    if ( pc != NULL )
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t = A*p
        fasp_blas_dblc_mxv(A,p,t);

        // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            ITS_DIVZERO; goto FINISHED;
        }

        // u_k = u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k = r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute norm of residual
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reduction factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
            normuinf = fasp_blas_darray_norminf(m, u->val);
            if ( normuinf <= sol_inf_tol ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if stagnated, try to restart
            normu = fasp_blas_darray_norm2(m, u->val);

            // compute relative difference
            reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
            if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(reldiff,relres);
                    ITS_RESTART;
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

                // compute residual norms
                switch ( StopType ) {
                    case STOP_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        // z = B(r)
                        if ( pc != NULL )
                            pc->fct(r,z,pc->data); /* Apply preconditioner */
                        else
                            fasp_darray_cp(m,r,z); /* No preconditioner */
                        absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normu;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p,0.0);
                    ++stag;
                }

            } // end of stagnation check!

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            REAL updated_relres = relres;

            // compute true residual r = b - Ax and update residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

            // compute residual norms
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            // check convergence
            if ( relres < tol ) break;

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(updated_relres); ITS_REALRES(relres);
            }

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p,0.0);
            ++more_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // stagnation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;

    // allocate temp memory (need 4*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

    if ( pc != NULL )
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t = A*p
        fasp_blas_dstr_mxv(A,p,t);

        // alpha_k = (z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            ITS_DIVZERO; goto FINISHED;
        }

        // u_k = u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k = r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute norm of residual
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reduction factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if solution is close to zero, return ERROR_SOLVER_SOLSTAG
            normuinf = fasp_blas_darray_norminf(m, u->val);
            if ( normuinf <= sol_inf_tol ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if stagnated, try to restart
            normu = fasp_blas_darray_norm2(m, u->val);

            // compute relative difference
            reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
            if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(reldiff,relres);
                    ITS_RESTART;
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

                // compute residual norms
                switch ( StopType ) {
                    case STOP_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        // z = B(r)
                        if ( pc != NULL )
                            pc->fct(r,z,pc->data); /* Apply preconditioner */
                        else
                            fasp_darray_cp(m,r,z); /* No preconditioner */
                        absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        absres = fasp_blas_darray_norm2(m,r);
                        relres = absres/normu;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p,0.0);
                    ++stag;
                }

            } // end of stagnation check!

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            REAL updated_relres = relres;

            // compute true residual r = b - Ax and update residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

            // compute residual norms
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            // check convergence
            if ( relres < tol ) break;

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(updated_relres); ITS_REALRES(relres);
            }

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p,0.0);
            ++more_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
```

```
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m=b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag, more_step, restart_step;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, infnormu;
    REAL         alpha, beta, temp1, temp2;

    // allocate temp memory (need 4*m REAL numbers)
    REAL *work=(REAL *)fasp_mem_calloc(4*m,sizeof(REAL));
    REAL *p=work, *z=work+m, *r=z+m, *t=r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling CG solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // initialize counters
    stag=1; more_step=1; restart_step=1;

    // r = b-A*u
    mf->fct(mf->data, u->val, r);
    fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

    if (pc != NULL)
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_PRECRES:
            absres0=sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0=MAX(SMALLREAL,absres0);
            relres=absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0=fasp_blas_darray_norm2(m,r);
            normu=MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres=absres0/normu;
            break;
        default:
            absres0=fasp_blas_darray_norm2(m,r);
            normr0=MAX(SMALLREAL,absres0);
            relres=absres0/normr0;
            break;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    fasp_darray_cp(m,z,p);
    temp1=fasp_blas_darray_dotprod(m,z,r);

    while ( iter++ < MaxIt ) {

        // t=A*p
        mf->fct(mf->data, p, t);

        // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2=fasp_blas_darray_dotprod(m,t,p);
        alpha=temp1/temp2;

        // u_k=u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k=r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);
        absres=fasp_blas_darray_norm2(m,r);

        // compute reducation factor of residual ||r||
        factor=absres/absres0;

        // compute relative residual
        switch (StopType) {
            case STOP_REL_PRECRES:
                // z = B(r)
                if (pc != NULL)
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                temp2=fasp_blas_darray_dotprod(m,z,r);
                relres=sqrt(ABS(temp2))/normr0;
                break;
            case STOP_MOD_REL_RES:
                relres=absres/normu;
                break;
            default:
                relres=absres/normr0;
                break;
        }

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // solution check, if soultion is too small, return ERROR_SOLVER_SOLSTAG.
        infnormu = fasp_blas_darray_norminf(m, u->val);
        if ( infnormu <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        // compute relative difference
        normu = fasp_blas_darray_norm2(m, u->val);
        reldiff = ABS(alpha)*fasp_blas_darray_norm2(m, p)/normu;

        // stagnation check
        if ( (stag<=MaxStag) & (reldiff<maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            mf->fct(mf->data, u->val, r);
            fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);
            absres = fasp_blas_darray_norm2(m,r);

            // relative residual
            switch (StopType) {
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if (pc != NULL)
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    temp2=fasp_blas_darray_dotprod(m,z,r);
                    relres=sqrt(ABS(temp2))/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    relres=absres/normu;
                    break;
                default:
                    relres=absres/normr0;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                fasp_darray_set(m,p,0.0);
                ++stag;
                ++restart_step;
            }
        } // end of stagnation check!

        // safe-guard check
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            mf->fct(mf->data, u->val, r);
            fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

            // relative residual
            switch (StopType) {
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if (pc != NULL)
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    temp2=fasp_blas_darray_dotprod(m,z,r);
                    relres=sqrt(ABS(temp2))/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres=fasp_blas_darray_norm2(m,r);
                    relres=absres/normu;
                    break;
                default:
                    absres=fasp_blas_darray_norm2(m,r);
                    relres=absres/normr0;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p,0.0);
            ++more_step;
            ++restart_step;

        } // end of safe-guard check!

        // update relative residual here
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2=fasp_blas_darray_dotprod(m,z,r);
        beta=temp2/temp1;
        temp1=temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
```

```
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 generilzed 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 generilzed 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

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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

Preconditioned generilzed conjugate gradient (GCG) method for solving Au=b.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
                precond * pc,
                const REAL tol,
                const INT MaxIt,
                const SHORT StopType,
                const SHORT PrtLvl )
```

Preconditioned generilzed conjugate gradient (GCG) method for solving Au=b.

Parameters

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type – DOES not support this parameter
PrtLvl	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    INT    iter=0, m=A->row, i;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normb  = BIGREAL;
    REAL   alpha, factor;

    // allocate temp memory
    REAL *work = (REAL *)fasp_mem_calloc(2*m+MaxIt+MaxIt*m,sizeof(REAL));

    REAL *r, *Br, *beta, *p;
    r = work; Br = r + m; beta = Br + m; p = beta + MaxIt;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GCG solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    normb=fasp_blas_darray_norm2(m,b->val);

    // -------------------------------------
    // 1st iteration (Steepest descent)
    // -------------------------------------
    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

    // Br
    if (pc != NULL)
        pc->fct(r,p,pc->data); /* Preconditioning */
    else
        fasp_darray_cp(m,r,p); /* No preconditioner, B=I */

    // alpha = (p'r)/(p'Ap)
    alpha = fasp_blas_darray_dotprod (m,r,p) / fasp_blas_dcsr_vmv (A, p, p);

    // u = u + alpha *p
    fasp_blas_darray_axpy(m, alpha , p, u->val);

    // r = r - alpha *Ap
    fasp_blas_dcsr_aAxpy((-1.0*alpha),A,p,r);

    // norm(r), factor
    absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;

    // compute relative residual
    relres = absres/normb;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

    // update relative residual here
    absres0 = absres;

    for ( iter = 1; iter < MaxIt ; iter++) {

        // Br
        if (pc != NULL)
            pc->fct(r, Br ,pc->data); // Preconditioning
        else
            fasp_darray_cp(m,r, Br); // No preconditioner, B=I

        // form p
        fasp_darray_cp(m, Br, p+iter*m);

        for (i=0; i<iter; i++) {
            beta[i] = (-1.0) * ( fasp_blas_dcsr_vmv (A, Br, p+i*m)
                                /fasp_blas_dcsr_vmv (A, p+i*m, p+i*m) );

            fasp_blas_darray_axpy(m, beta[i], p+i*m, p+iter*m);
        }

        // -------------------------------------
        // next iteration
        // -------------------------------------

        // alpha = (p'r)/(p'Ap)
        alpha = fasp_blas_darray_dotprod(m,r,p+iter*m)
              / fasp_blas_dcsr_vmv (A, p+iter*m, p+iter*m);

        // u = u + alpha *p
        fasp_blas_darray_axpy(m, alpha , p+iter*m, u->val);

        // r = r - alpha *Ap
        fasp_blas_dcsr_aAxpy((-1.0*alpha),A,p+iter*m,r);

        // norm(r), factor
        absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;

        // compute relative residual
        relres = absres/normb;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if (relres < tol) break;

        // update relative residual here
        absres0 = absres;

    } // end of main GCG loop.

    // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    INT    iter=0, m=b->row, i;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normb  = BIGREAL;
    REAL   alpha, factor, gama_1, gama_2;

    // allocate temp memory
    REAL *work = (REAL *)fasp_mem_calloc(3*m+MaxIt+MaxIt*m,sizeof(REAL));

    REAL *r, *Br, *beta, *p, *q;
    q = work; r = q + m; Br = r + m; beta = Br + m; p = beta + MaxIt;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GCG solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    normb=fasp_blas_darray_norm2(m,b->val);

    // -------------------------------------
    // 1st iteration (Steepest descent)
    // -------------------------------------
    // r = b-A*u
    mf->fct(mf->data, u->val, r);
    fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

    // Br
    if (pc != NULL)
        pc->fct(r,p,pc->data); /* Preconditioning */
    else
        fasp_darray_cp(m,r,p); /* No preconditioner, B=I */

    // alpha = (p'r)/(p'Ap)
    mf->fct(mf->data, p, q);
    alpha = fasp_blas_darray_dotprod (m,r,p) / fasp_blas_darray_dotprod (m, p, q);

    // u = u + alpha *p
    fasp_blas_darray_axpy(m, alpha , p, u->val);

    // r = r - alpha *Ap
    mf->fct(mf->data, p, q);
    fasp_blas_darray_axpby(m, (-1.0*alpha), q, 1.0, r);

    // norm(r), factor
    absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;

    // compute relative residual
    relres = absres/normb;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter+1,relres,absres,factor);

    // update relative residual here
    absres0 = absres;

    for ( iter = 1; iter < MaxIt ; iter++) {

        // Br
        if (pc != NULL)
            pc->fct(r, Br ,pc->data); // Preconditioning
        else
            fasp_darray_cp(m,r, Br); // No preconditioner, B=I

        // form p
        fasp_darray_cp(m, Br, p+iter*m);

        for (i=0; i<iter; i++) {
            mf->fct(mf->data, Br, q);
            gama_1 = fasp_blas_darray_dotprod(m, p+i*m, q);
            mf->fct(mf->data, p+i*m, q);
            gama_2 = fasp_blas_darray_dotprod(m, p+i*m, q);
            beta[i] = (-1.0) * ( gama_1 / gama_2 );

            fasp_blas_darray_axpy(m, beta[i], p+i*m, p+iter*m);
        }

        // -------------------------------------
        // next iteration
        // -------------------------------------

        // alpha = (p'r)/(p'Ap)
        mf->fct(mf->data, p+iter*m, q);
        alpha = fasp_blas_darray_dotprod(m,r,p+iter*m)
        / fasp_blas_darray_dotprod (m, q, p+iter*m);

        // u = u + alpha *p
        fasp_blas_darray_axpy(m, alpha , p+iter*m, u->val);

        // r = r - alpha *Ap
        mf->fct(mf->data, p+iter*m, q);
        fasp_blas_darray_axpby(m, (-1.0*alpha), q, 1.0, r);

        // norm(r), factor
        absres = fasp_blas_darray_norm2(m,r); factor = absres/absres0;

        // compute relative residual
        relres = absres/normb;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter+1,relres,absres,factor);

        if (relres < tol) break;

        // update relative residual here
        absres0 = absres;

    } // end of main GCG loop.

    // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.115 KryPgcr.c File Reference

Krylov subspace methods – Preconditioned GCR.

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

```
#include "fasp_functs.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

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,
             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.

Parameters

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
x	Pointer to dvector of dofs
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopage
MaxIt	Maximal number of iterations

Parameters

restart	Restart number for GCR
StopType	Stopping type
PrtLvl	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,
            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.

Parameters

A	Pointer to coefficient matrix
b	Pointer to dvector of right hand side
x	Pointer to dvector of dofs
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopage
MaxIt	Maximal number of iterations
restart	Restart number for GCR
StopType	Stopping type
PrtLvl	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_aAtxpby (INT, INT, REAL *, REAL, REAL *, REAL, REAL *);
00029
00055 INT fasp_solver_dcsr_pgcr (dCSRmat     *A,
00056                            dvector     *b,
00057                            dvector     *x,
00058                            precond     *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__);
```

```
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GCR restart number set to %d!\n", Restart);
    }

    r = work; z = r+n; c = z + Restart*n; alp = c + Restart*n; tmpx = alp + Restart;

    h = (REAL **)fasp_mem_calloc(Restart, sizeof(REAL *));
    for (i = 0; i < Restart; i++) h[i] = (REAL*)fasp_mem_calloc(Restart, sizeof(REAL));

    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    // r = b-A*x
    fasp_darray_cp(n, b->val, r);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);

    absres = fasp_blas_darray_dotprod(n, r, r);

    absres0 = MAX(SMALLREAL,absres);

    relres  = absres/absres0;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,sqrt(absres0),0.0);

    // store initial residual
    norms[0] = relres;

    checktol = MAX(tol*tol*absres0, absres*1.0e-4);

    while ( iter < MaxIt && sqrt(relres) > tol ) {

        i = -1; rst ++;

        while ( i < Restart-1 && iter < MaxIt ) {

            i++; iter++;

            // z = B^-1r
            if ( pc == NULL )
                fasp_darray_cp(n, r, &z[i*n]);
            else
                pc->fct(r, &z[i*n], pc->data);

            // c = Az
            fasp_blas_dcsr_mxv(A, &z[i*n], &c[i*n]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                gamma = fasp_blas_darray_dotprod(n, &c[j*n], &c[i*n]);
                h[i][j] = gamma/h[j][j];
                fasp_blas_darray_axpy(n, -h[i][j], &c[j*n], &c[i*n]);
            }
            // gamma = (c,c)
            gamma = fasp_blas_darray_dotprod(n, &c[i*n], &c[i*n]);

            h[i][i] = gamma;

            // alpha = (c, r)
            alpha = fasp_blas_darray_dotprod(n, &c[i*n], r);

            beta = alpha/gamma;

            alp[i] = beta;

            // r = r - beta*c
            fasp_blas_darray_axpy(n, -beta, &c[i*n], r);

            // equivalent to ||r||_2
            absres = absres - alpha*alpha/gamma;

            if (absres < checktol) {
                absres = fasp_blas_darray_dotprod(n, r, r);
                checktol = MAX(tol*tol*absres0, absres*1.0e-4);
            }

            relres = absres / absres0;

            norms[iter] = relres;

            fasp_itinfo(PrtLvl, StopType, iter, sqrt(relres), sqrt(absres),
                        sqrt(norms[iter]/norms[iter-1]));

            if (sqrt(relres) < tol)  break;
        }

        for ( k = i; k >=0; k-- ) {
            tmpx[k] = alp[k];
            for (j=0; j<k; ++j) {
                alp[j] -= h[k][j]*tmpx[k];
            }
        }

        if (rst==0) dense_aAtxpby(n, i+1, z, 1.0, tmpx, 0.0, x->val);
        else dense_aAtxpby(n, i+1, z, 1.0, tmpx, 1.0, x->val);

    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,sqrt(relres));

    // clean up memory
    for (i = 0; i < Restart; i++) {
        fasp_mem_free(h[i]); h[i] = NULL;
    }
    fasp_mem_free(h); h = NULL;

    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT   n = b->row;

    // local variables
    INT      iter = 0;
    int      i, j, k, rst = -1; // must be signed!  -zcs

    REAL     gamma, alpha, beta, checktol;
    REAL     absres0 = BIGREAL, absres = BIGREAL;
    REAL     relres  = BIGREAL;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *z = NULL, *alp = NULL, *tmpx = NULL;
    REAL    *norms = NULL, *r = NULL, *work = NULL;
    REAL    **h = NULL;

    INT      Restart = MIN(restart, MaxIt);
    LONG     worksize = n+2*Restart*n+Restart+Restart;

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GCR solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GCR */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = n+2*Restart*n+Restart+Restart;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory for GCR! [%s:%d]\n",
               __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GCR restart number set to %d!\n", Restart);
    }

    r = work; z = r+n; c = z + Restart*n; alp = c + Restart*n; tmpx = alp + Restart;

    h = (REAL **)fasp_mem_calloc(Restart, sizeof(REAL *));
    for (i = 0; i < Restart; i++) h[i] = (REAL*)fasp_mem_calloc(Restart, sizeof(REAL));

    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    // r = b-A*x
    fasp_darray_cp(n, b->val, r);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);

    absres = fasp_blas_darray_dotprod(n, r, r);

    absres0 = MAX(SMALLREAL,absres);

    relres  = absres/absres0;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,sqrt(absres0),0.0);

    // store initial residual
    norms[0] = relres;

    checktol = MAX(tol*tol*absres0, absres*1.0e-4);

    while ( iter < MaxIt && sqrt(relres) > tol ) {

        i = 0; rst ++;
        while ( i < Restart && iter < MaxIt ) {

            iter++;

            // z = B^-1r
            if ( pc == NULL )
                fasp_darray_cp(n, r, &z[i*n]);
            else
                pc->fct(r, &z[i*n], pc->data);

            // c = Az
            fasp_blas_dblc_mxv(A, &z[i*n], &c[i*n]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                gamma = fasp_blas_darray_dotprod(n, &c[j*n], &c[i*n]);
                h[i][j] = gamma/h[j][j];
                fasp_blas_darray_axpy(n, -h[i][j], &c[j*n], &c[i*n]);
            }
            // gamma = (c,c)
            gamma = fasp_blas_darray_dotprod(n, &c[i*n], &c[i*n]);

            h[i][i] = gamma;

            // alpha = (c, r)
            alpha = fasp_blas_darray_dotprod(n, &c[i*n], r);

            beta = alpha/gamma;

            alp[i] = beta;

            // r = r - beta*c
            fasp_blas_darray_axpy(n, -beta, &c[i*n], r);

            // equivalent to ||r||_2
            absres = absres - alpha*alpha/gamma;

            if (absres < checktol) {
                absres = fasp_blas_darray_dotprod(n, r, r);
                checktol = MAX(tol*tol*absres0, absres*1.0e-4);
            }

            relres = absres / absres0;

            norms[iter] = relres;

            fasp_itinfo(PrtLvl, StopType, iter, sqrt(relres), sqrt(absres),
                        sqrt(norms[iter]/norms[iter-1]));

            if (sqrt(relres) < tol)  break;

            i++;
        }

        for ( k = i; k >=0; k-- ) {
            tmpx[k] = alp[k];
            for (j=0; j<k; ++j) {
                alp[j] -= h[k][j]*tmpx[k];
            }
        }

        if (rst==0) dense_aAtxpby(n, i+1, z, 1.0, tmpx, 0.0, x->val);
        else dense_aAtxpby(n, i+1, z, 1.0, tmpx, 1.0, x->val);
    }

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,sqrt(relres));

    // clean up memory
    for (i = 0; i < Restart; i++) {
        fasp_mem_free(h[i]); h[i] = NULL;
    }
    fasp_mem_free(h); h = NULL;

    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--    Private Functions        --*/
/*---------------------------------*/

static void dense_aAtxpby (INT   n,
                           INT   m,
                           REAL *A,
                           REAL  alpha,
                           REAL *x,
                           REAL  beta,
                           REAL *y)
{
    INT i, j;

    for (i=0; i<m; i++) fasp_blas_darray_ax(n, x[i], &A[i*n]);

    for (j=1; j<m; j++) {
        for (i=0; i<n; i++) {
            A[i] += A[i+j*n];
        }
    }

    fasp_blas_darray_axpby(n, alpha, A, beta, y);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, precond ∗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, precond ∗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, precond ∗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, precond ∗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, precond ∗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

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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT restart,
            const SHORT StopType,
            const SHORT PrtLvl )
```

Preconditioned GMRES method for solving Au=b.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
              precond * pc,
              const REAL tol,
              const INT MaxIt,
              const SHORT restart,
              const SHORT StopType,
              const SHORT PrtLvl )
```

Preconditioned GMRES method for solving Au=b.

Parameters

A	Pointer to dBSRmat: coefficient matrix
b	Pointer to dvector: right hand side

Parameters

x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             precond * 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

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT restart,
             const SHORT StopType,
             const SHORT PrtLvl )
```

Preconditioned GMRES method for solving Au=b.

Parameters

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             precond * 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

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type – DOES not support this parameter
PrtLvl	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>
```

```

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dcsr_pgmres (dCSRmat     *A,
                             dvector     *b,
                             dvector     *x,
                             precond     *pc,
                             const REAL   tol,
                             const INT    MaxIt,
                             const SHORT  restart,
                             const SHORT  StopType,
                             const SHORT  PrtLvl)
{
    const INT   n          = b->row;
    const INT   MIN_ITER   = 0;

    // local variables
    INT      iter = 0;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     absres0 = BIGREAL, absres = BIGREAL;
    REAL     relres  = BIGREAL, normu  = BIGREAL;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL;
    REAL   **p = NULL, **hh = NULL;

    INT   Restart  = MIN(restart, MaxIt);
    INT   Restart1 = Restart + 1;
    LONG  worksize = (Restart+4)*(Restart+n)+1-n;

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        Restart1 = Restart + 1;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1,  sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i]  = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // compute initial residual:  r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, p[0]);
    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute stopping criteria
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0.0);

    // store initial residual
    norms[0] = relres;

    /* GMRES(M) outer iteration */
    while ( iter < MaxIt && relres > tol ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dcsr_mxv(A, r, p[i]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;

            if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]*hh[i][i-1];
            t += hh[i-1][i-1]*hh[i-1][i-1];

            gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // exit restart cycle if reaches tolerance
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else { // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */
FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

INT fasp_solver_dbsr_pgmres (dBSRmat     *A,
                             dvector     *b,
                             dvector     *x,
                             precond     *pc,
                             const REAL   tol,
                             const INT    MaxIt,
                             const SHORT  restart,
                             const SHORT  StopType,
                             const SHORT  PrtLvl)
{
    const INT   n         = b->row;
    const INT   MIN_ITER  = 0;

    // local variables
    INT      iter = 0;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     absres0 = BIGREAL, absres = BIGREAL;
    REAL     relres  = BIGREAL, normu  = BIGREAL;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL;
    REAL   **p = NULL, **hh = NULL;

    INT   Restart  = MIN(restart, MaxIt);
    INT   Restart1 = Restart + 1;
    LONG  worksize = (Restart+4)*(Restart+n)+1-n;

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        Restart1 = Restart + 1;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i]  = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // compute initial residual:  r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dbsr_aAxpy(-1.0, A, x->val, p[0]);
    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute stopping criteria
    switch (StopType) {
            case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
            case STOP_REL_PRECRES:
            if ( pc == NULL )
            fasp_darray_cp(n, p[0], r);
            else
            pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
            case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
            default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0.0);

    // store initial residual
    norms[0] = relres;

    /* GMRES(M) outer iteration */
    while ( iter < MaxIt && relres > tol ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
            fasp_darray_cp(n, p[i-1], r);
            else
            pc->fct(p[i-1], r, pc->data);

            fasp_blas_dbsr_mxv(A, r, p[i]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;

            if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]*hh[i][i-1];
            t += hh[i-1][i-1]*hh[i-1][i-1];

            gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // exit restart cycle if reaches tolerance
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
        fasp_darray_cp(n, w, r);
        else
        pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dbsr_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                    case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                    case STOP_REL_PRECRES:
                    if ( pc == NULL )
                    fasp_darray_cp(n, r, w);
                    else
                    pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                    case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else { // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }


        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);      p     = NULL;
    fasp_mem_free(hh);     hh    = NULL;
    fasp_mem_free(norms);  norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
```

```

    if ( iter >= MaxIt )
    return ERROR_SOLVER_MAXIT;
    else
    return iter;
}

INT fasp_solver_dblc_pgmres (dBLCmat      *A,
                             dvector      *b,
                             dvector      *x,
                             precond      *pc,
                             const REAL    tol,
                             const INT     MaxIt,
                             const SHORT   restart,
                             const SHORT   StopType,
                             const SHORT   PrtLvl)
{
    const INT   n         = b->row;
    const INT   MIN_ITER  = 0;

    // local variables
    INT      iter = 0;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     absres0 = BIGREAL, absres = BIGREAL;
    REAL     relres  = BIGREAL, normu  = BIGREAL;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL;
    REAL   **p = NULL, **hh = NULL;

    INT   Restart  = MIN(restart, MaxIt);
    INT   Restart1 = Restart + 1;
    LONG  worksize = (Restart+4)*(Restart+n)+1-n;

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        Restart1 = Restart + 1;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i]  = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // compute initial residual:  r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, p[0]);
    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute stopping criteria
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0.0);

    // store initial residual
    norms[0] = relres;

    /* GMRES(M) outer iteration */
    while ( iter < MaxIt && relres > tol ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dblc_mxv(A, r, p[i]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;

            if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]*hh[i][i-1];
            t += hh[i-1][i-1]*hh[i-1][i-1];

            gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // exit restart cycle if reaches tolerance
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else { // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

INT fasp_solver_dstr_pgmres (dSTRmat     *A,
                             dvector     *b,
                             dvector     *x,
                             precond     *pc,
                             const REAL   tol,
                             const INT    MaxIt,
                             const SHORT  restart,
                             const SHORT  StopType,
                             const SHORT  PrtLvl)
{
    const INT   n         = b->row;
    const INT   MIN_ITER  = 0;

    // local variables
    INT      iter = 0;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     absres0 = BIGREAL, absres = BIGREAL;
    REAL     relres  = BIGREAL, normu  = BIGREAL;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL;
    REAL   **p = NULL, **hh = NULL;

    INT   Restart  = MIN(restart, MaxIt);
    INT   Restart1 = Restart + 1;
    LONG  worksize = (Restart+4)*(Restart+n)+1-n;

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (STR) ...\n");

```

```
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        Restart1 = Restart + 1;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i]  = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // compute initial residual:  r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dstr_aAxpy(-1.0, A, x->val, p[0]);
    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute stopping criteria
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-312*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0.0);

    // store initial residual
    norms[0] = relres;

    /* GMRES(M) outer iteration */
    while ( iter < MaxIt && relres > tol ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dstr_mxv(A, r, p[i]);

            /* Modified Gram_Schmidt orthogonalization */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;

            if ( ABS(t) > SMALLREAL ) { // If t=0, we get solution subspace
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]*hh[i][i-1];
            t += hh[i-1][i-1]*hh[i-1][i-1];

            gamma = MAX(sqrt(t), SMALLREAL); // Possible breakdown?
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // exit restart cycle if reaches tolerance
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for ( j = k+1; j < i; j++ ) t -= hh[k][j]*rs[j];
            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dstr_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else { // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

```

```
INT fasp_solver_pgmres (mxv_matfree  *mf,
                        dvector      *b,
                        dvector      *x,
                        precond      *pc,
                        const REAL    tol,
                        const INT     MaxIt,
                        const SHORT   restart,
                        const SHORT   StopType,
                        const SHORT   PrtLvl)
{
    const INT n         = b->row;
    const INT min_iter  = 0;

    // local variables
    INT      iter = 0;
    int      i, j, k; // must be signed!  -zcs

    REAL     epsmac = SMALLREAL;
    REAL     r_norm, b_norm, den_norm;
    REAL     epsilon, gamma, t;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL;
    REAL    **p = NULL, **hh = NULL;

    INT  Restart  = restart;
    INT  Restart1 = Restart + 1;
    LONG worksize = (Restart+4)*(Restart+n)+1-n;

    // Output some info for debugging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling GMRes solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: GMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + Restart;

    for (i = 0; i < Restart1; i ++) p[i] = s + Restart + i*n;

    for (i = 0; i < Restart1; i ++) hh[i] = p[Restart] + n + i*Restart;

    /* initialization */
    mf->fct(mf->data, x->val, p[0]);
    fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);

    if ( PrtLvl > PRINT_NONE) {
        norms[0] = r_norm;
        if ( PrtLvl >= PRINT_SOME ) {
            ITS_PUTNORM("right-hand side", b_norm);
            ITS_PUTNORM("residual", r_norm);
        }
    }

    if (b_norm > 0.0)  den_norm = b_norm;
    else               den_norm = r_norm;

    epsilon = tol*den_norm;

    /* outer iteration cycle */
    while (iter < MaxIt) {

        rs[0] = r_norm;
        if (r_norm == 0.0) {
            fasp_mem_free(work);   work  = NULL;
            fasp_mem_free(p);      p     = NULL;
            fasp_mem_free(hh);     hh    = NULL;
            fasp_mem_free(norms);  norms = NULL;
            return iter;
        }

        if (r_norm <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm <= epsilon) {
                break;
            }
            else {
                if (PrtLvl >= PRINT_SOME) ITS_FACONV;
            }
        }

        t = 1.0 / r_norm;
        //for (j = 0; j < n; j ++) p[0][j] *= t;
        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while (i < Restart && iter < MaxIt) {

            i ++;  iter ++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            mf->fct(mf->data, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                //for (j = 0; j < n; j ++) p[i][j] *= t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
            r_norm = fabs(rs[i]);

            norms[iter] = r_norm;

            if (b_norm > 0 ) {
                fasp_itinfo(PrtLvl,StopType,iter,norms[iter]/b_norm,
                            norms[iter],norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl,StopType,iter,norms[iter],norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* should we exit restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) {
                break;
            }
        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }
        fasp_darray_cp(n, p[i-1], w);
        //for (j = 0; j < n; j ++) w[j] *= rs[i-1];
        fasp_blas_darray_ax(n, rs[i-1], w);
        for (j = i-2; j >= 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if (pc == NULL)
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if (r_norm  <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm  <= epsilon) {
                break;
            }
            else {
                if (PrtLvl >= PRINT_SOME) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }
        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for (j = i-1 ; j > 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }
    } /* end of iteration while loop */

    if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);      p     = NULL;
    fasp_mem_free(hh);     hh    = NULL;
    fasp_mem_free(norms);  norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
```

```

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

#if 0
static double estimate_spectral_radius (const double **A, int n, size_t k = 20)
{
    double *x = (double *)malloc(n* sizeof(double));
    double *y = (double *)malloc(n* sizeof(double));
    double *z = (double *)malloc(n* sizeof(double));
    double t;
    int i1,j1;

    // initialize x to random values in [0,1)
    //    cusp::copy(cusp::detail::random_reals<ValueType>(N), x);
    dvector px;
    px.row = n;
    px.val = x;

    fasp_dvec_rand(n, &px);

    for(size_t i = 0; i < k; i++)
    {
        //cusp::blas::scal(x, ValueType(1.0) / cusp::blas::nrmmax(x));
        t= 1.0/ fasp_blas_darray_norminf(n, px);
        for(i1= 0; i1 <n; i1++) x[i1] *= t;

        //cusp::multiply(A, x, y);

        for(i1= 0; i1 <n; i1++) {
            t= 0.0
            for(j1= 0; j1 <n; j1++)  t +=  A[i1][j1] * x[j1];
            y[i1] = t;    }
        //          x.swap(y);
        for(i1= 0; i1 <n; i1++) z[i1] = x[i1];
        for(i1= 0; i1 <n; i1++) x[i1] = y[i1];
        for(i1= 0; i1 <n; i1++) y[i1] = z[i1];
    }

    free(x);
    free(y);
    free(z);

    if (k == 0)
        return 0;
    else
        //return cusp::blas::nrm2(x) / cusp::blas::nrm2(y);
        return fasp_blas_darray_norm2(n,x) / fasp_blas_darray_norm2(n,y) ;
}

static double spectral_radius (dCSRmat *A,
                               const SHORT restart)
{
    const INT n         = A->row;
    const INT MIN_ITER  = 0;

    // local variables
    INT      iter = 0;
    INT      Restart1 = restart + 1;
    INT      i, j, k;

    REAL     r_norm, den_norm;
    REAL     epsilon, gamma, t;

    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL   **p = NULL, **hh = NULL;
    REAL    *work = NULL;

    /* allocate memory */
    work = (REAL *)fasp_mem_calloc((restart+4)*(restart+n)+1-n, sizeof(REAL));
    p    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh   = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));

    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c  = rs + Restart1; s  = c + restart;

    for (i = 0; i < Restart1; i ++) p[i] = s + restart + i*n;
    for (i = 0; i < Restart1; i ++) hh[i] = p[restart] + n + i*restart;

    /* initialization */
    dvector p0;
    p0.row = n;
    p0.val = p[0];
    fasp_dvec_rand(n, &p0);

    r_norm = fasp_blas_darray_norm2(n, p[0]);
    t = 1.0 / r_norm;
    for (j = 0; j < n; j ++) p[0][j] *= t;

    int maxiter = MIN(n, restart) ;
    for ( j = 0; j < maxiter; j++ ) {
        fasp_blas_bdbsr_mxv(A, p[j], p[j+1]);

        for( i = 0; i <= j; i++ ) {
            hh[i][j] = fasp_blas_darray_dotprod(n, p[i], p[j+1]);
            fasp_blas_darray_axpy(n, -hh[i][j], p[i], p[ j+1 ]);
        }

        hh[j+1][j] =  fasp_blas_darray_norm2 (n, p[j+1]);
        if ( hh[j+1][j] < 1e-10) break;
        t = 1.0/hh[j+1][j];
        for (k = 0; k < n; k ++) p[j+1][k] *= t;
    }

    H    = (REAL **)fasp_mem_calloc(j, sizeof(REAL *));
    H[0] = (REAL *)fasp_mem_calloc(j*j, sizeof(REAL));
    for (i = 1; i < j; i ++) H[i] = H[i-1] + j;


    for( size_t row = 0; row < j; row++ )
        for( size_t col = 0; col < j; col++ )
            H[row][col] = hh[row][col];

    double spectral_radius = estimate_spectral_radius( H, j, 20);

    /*---------------------------------------------
* Clean up workspace
*---------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;
    fasp_mem_free(H[0]);  H[0]  = NULL;
    fasp_mem_free(H);     H     = NULL;

    return spectral_radius;
}
#endif

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.119 KryPminres.c File Reference

Krylov subspace methods – Preconditioned minimal residual.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.h"
#include "KryUtil.inl"
```

Functions

- INT fasp_solver_dcsr_pminres (dCSRmat ∗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_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

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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving Au=b.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
             precond * pc,
             const REAL tol,
             const INT MaxIt,
             const SHORT StopType,
             const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving Au=b.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving Au=b.

Parameters

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            precond * pc,
            const REAL tol,
            const INT MaxIt,
            const SHORT StopType,
            const SHORT PrtLvl )
```

A preconditioned minimal residual (Minres) method for solving Au=b.

Parameters

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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.

```
#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dcsr_pminres (dCSRmat      *A,
                              dvector      *b,
                              dvector      *u,
                              precond      *pc,
                              const REAL    tol,
                              const INT     MaxIt,
                              const SHORT   StopType,
                              const SHORT   PrtLvl)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, infnormu, factor;
    REAL         alpha, alpha0, alpha1, temp2;

    // allocate temp memory (need 11*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
    REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // p0 = 0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

    // p1 = B(r)
    if ( pc != NULL )
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu2;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // tp = A*p1
    fasp_blas_dcsr_mxv(A,p1,tp);

    // tz = B(tp)
    if ( pc != NULL )
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1 = p1/normp
    normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp = sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0 = A*p0 = 0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1 = tp/normp, z1 = tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    // main MinRes loop
    while ( iter++ < MaxIt ) {

        // alpha = <r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u = u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r = r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t = A*z1 alpha1 = <z1,t>
        fasp_blas_dcsr_mxv(A,z1,t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t = A*z0 alpha0 = <z1,t>
        fasp_blas_dcsr_mxv(A,z0,t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2 = z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp = A*p2
        fasp_blas_dcsr_mxv(A,p2,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2 = p2/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);

        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        normu2 = fasp_blas_darray_norm2(m,u->val);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normu2;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
            infnormu = fasp_blas_darray_norminf(m, u->val);
            if (infnormu <= sol_inf_tol) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if staggenated, try to restart
            normuu = fasp_blas_darray_norm2(m,p1);
            normuu = ABS(alpha)*(normuu/normu2);

            if ( normuu < maxdiff ) {

                if ( stag < MaxStag ) {
                    if ( PrtLvl >= PRINT_MORE ) {
                        ITS_DIFFRES(normuu,relres);
                        ITS_RESTART;
                    }
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p0,0.0);
                    ++stag;
                    ++restart_step;

                    // p1 = B(r)
                    if ( pc != NULL )
                        pc->fct(r,p1,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,p1); /* No preconditioner */

                    // tp = A*p1
                    fasp_blas_dcsr_mxv(A,p1,tp);

                    // tz = B(tp)
                    if ( pc != NULL )
                        pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                    else
                        fasp_darray_cp(m,tp,tz); /* No preconditioner */

                    // p1 = p1/normp
                    normp = fasp_blas_darray_dotprod(m,tz,tp);
                    normp = sqrt(normp);
                    fasp_darray_cp(m,p1,t);

                    // t0 = A*p0=0
                    fasp_darray_set(m,t0,0.0);
                    fasp_darray_cp(m,t0,z0);
                    fasp_darray_cp(m,t0,t1);
                    fasp_darray_cp(m,t0,z1);
                    fasp_darray_cp(m,t0,p1);

                    fasp_blas_darray_axpy(m,1/normp,t,p1);

                    // t1 = tp/normp, z1 = tz/normp
                    fasp_blas_darray_axpy(m,1/normp,tp,t1);
                    fasp_blas_darray_axpy(m,1/normp,tz,z1);
                }
            }

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p0,0.0);
            ++more_step;
            ++restart_step;

            // p1 = B(r)
            if ( pc != NULL )
                pc->fct(r,p1,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,p1); /* No preconditioner */

            // tp = A*p1
            fasp_blas_dcsr_mxv(A,p1,tp);

            // tz = B(tp)
            if ( pc != NULL )
                pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p1 = p1/normp
            normp = fasp_blas_darray_dotprod(m,tz,tp);
            normp = sqrt(normp);
            fasp_darray_cp(m,p1,t);

            // t0 = A*p0 = 0
            fasp_darray_set(m,t0,0.0);
            fasp_darray_cp(m,t0,z0);
            fasp_darray_cp(m,t0,t1);
            fasp_darray_cp(m,t0,z1);
            fasp_darray_cp(m,t0,p1);

            fasp_blas_darray_axpy(m,1/normp,t,p1);

            // t1=tp/normp,z1=tz/normp
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

        } // end of convergence check

        // update relative residual here
        absres0 = absres;

    } // end of the main loop

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, infnormu, factor;
    REAL         alpha, alpha0, alpha1, temp2;

    // allocate temp memory (need 11*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
    REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // p0 = 0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

    // p1 = B(r)
    if ( pc != NULL )
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu2;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // tp = A*p1
    fasp_blas_dblc_mxv(A,p1,tp);

    // tz = B(tp)
    if ( pc != NULL )
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1 = p1/normp
    normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp = sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0 = A*p0 = 0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1 = tp/normp, z1 = tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    // main MinRes loop
    while ( iter++ < MaxIt ) {

        // alpha = <r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u = u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r = r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t = A*z1 alpha1 = <z1,t>
        fasp_blas_dblc_mxv(A,z1,t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t = A*z0 alpha0 = <z1,t>
        fasp_blas_dblc_mxv(A,z0,t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2 = z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp = A*p2
        fasp_blas_dblc_mxv(A,p2,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2 = p2/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);

        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        normu2 = fasp_blas_darray_norm2(m,u->val);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normu2;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
            infnormu = fasp_blas_darray_norminf(m, u->val);
            if (infnormu <= sol_inf_tol) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if staggenated, try to restart
            normuu = fasp_blas_darray_norm2(m,p1);
            normuu = ABS(alpha)*(normuu/normu2);

            if ( normuu < maxdiff ) {

                if ( stag < MaxStag ) {
                    if ( PrtLvl >= PRINT_MORE ) {
                        ITS_DIFFRES(normuu,relres);
                        ITS_RESTART;
                    }
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p0,0.0);
                    ++stag;
                    ++restart_step;

                    // p1 = B(r)
                    if ( pc != NULL )
                        pc->fct(r,p1,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,p1); /* No preconditioner */

                    // tp = A*p1
                    fasp_blas_dblc_mxv(A,p1,tp);

                    // tz = B(tp)
                    if ( pc != NULL )
                        pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                    else
                        fasp_darray_cp(m,tp,tz); /* No preconditioner */

                    // p1 = p1/normp
                    normp = fasp_blas_darray_dotprod(m,tz,tp);
                    normp = sqrt(normp);
                    fasp_darray_cp(m,p1,t);

                    // t0 = A*p0=0
                    fasp_darray_set(m,t0,0.0);
                    fasp_darray_cp(m,t0,z0);
                    fasp_darray_cp(m,t0,t1);
                    fasp_darray_cp(m,t0,z1);
                    fasp_darray_cp(m,t0,p1);

                    fasp_blas_darray_axpy(m,1/normp,t,p1);

                    // t1 = tp/normp, z1 = tz/normp
                    fasp_blas_darray_axpy(m,1/normp,tp,t1);
                    fasp_blas_darray_axpy(m,1/normp,tz,z1);
                }
            }

        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p0,0.0);
            ++more_step;
            ++restart_step;

            // p1 = B(r)
            if ( pc != NULL )
                pc->fct(r,p1,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,p1); /* No preconditioner */

            // tp = A*p1
            fasp_blas_dblc_mxv(A,p1,tp);

            // tz = B(tp)
            if ( pc != NULL )
                pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p1 = p1/normp
            normp = fasp_blas_darray_dotprod(m,tz,tp);
            normp = sqrt(normp);
            fasp_darray_cp(m,p1,t);

            // t0 = A*p0 = 0
            fasp_darray_set(m,t0,0.0);
            fasp_darray_cp(m,t0,z0);
            fasp_darray_cp(m,t0,t1);
            fasp_darray_cp(m,t0,z1);
            fasp_darray_cp(m,t0,p1);

            fasp_blas_darray_axpy(m,1/normp,t,p1);

            // t1=tp/normp,z1=tz/normp
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

        } // end of convergence check

        // update relative residual here
        absres0 = absres;

    } // end of the main loop

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, infnormu, factor;
    REAL         alpha, alpha0, alpha1, temp2;

    // allocate temp memory (need 11*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
    REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // p0 = 0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

    // p1 = B(r)
    if ( pc != NULL )
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu2;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // tp = A*p1
    fasp_blas_dstr_mxv(A,p1,tp);

    // tz = B(tp)
    if ( pc != NULL )
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1 = p1/normp
    normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp = sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0 = A*p0 = 0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1 = tp/normp, z1 = tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    // main MinRes loop
    while ( iter++ < MaxIt ) {

        // alpha = <r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u = u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r = r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t = A*z1 alpha1 = <z1,t>
        fasp_blas_dstr_mxv(A,z1,t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t = A*z0 alpha0 = <z1,t>
        fasp_blas_dstr_mxv(A,z0,t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2 = z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp = A*p2
        fasp_blas_dstr_mxv(A,p2,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2 = p2/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);

        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        normu2 = fasp_blas_darray_norm2(m,u->val);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normu2;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        if ( factor > 0.9 ) { // Only check when converge slowly

            // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
            infnormu = fasp_blas_darray_norminf(m, u->val);
            if (infnormu <= sol_inf_tol) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if staggenated, try to restart
            normuu = fasp_blas_darray_norm2(m,p1);
            normuu = ABS(alpha)*(normuu/normu2);

            if ( normuu < maxdiff ) {

                if ( stag < MaxStag ) {
                    if ( PrtLvl >= PRINT_MORE ) {
                        ITS_DIFFRES(normuu,relres);
                        ITS_RESTART;
                    }
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p0,0.0);
                    ++stag;
                    ++restart_step;

                    // p1 = B(r)
                    if ( pc != NULL )
                        pc->fct(r,p1,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,p1); /* No preconditioner */

                    // tp = A*p1
                    fasp_blas_dstr_mxv(A,p1,tp);

                    // tz = B(tp)
                    if ( pc != NULL )
                        pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                    else
                        fasp_darray_cp(m,tp,tz); /* No preconditioner */

                    // p1 = p1/normp
                    normp = fasp_blas_darray_dotprod(m,tz,tp);
                    normp = sqrt(normp);
                    fasp_darray_cp(m,p1,t);

                    // t0 = A*p0=0
                    fasp_darray_set(m,t0,0.0);
                    fasp_darray_cp(m,t0,z0);
                    fasp_darray_cp(m,t0,t1);
                    fasp_darray_cp(m,t0,z1);
                    fasp_darray_cp(m,t0,p1);

                    fasp_blas_darray_axpy(m,1/normp,t,p1);

                    // t1 = tp/normp, z1 = tz/normp
                    fasp_blas_darray_axpy(m,1/normp,tp,t1);
                    fasp_blas_darray_axpy(m,1/normp,tz,z1);
                }
            }
        } // end of check I and II

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p0,0.0);
            ++more_step;
            ++restart_step;

            // p1 = B(r)
            if ( pc != NULL )
                pc->fct(r,p1,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,p1); /* No preconditioner */

            // tp = A*p1
            fasp_blas_dstr_mxv(A,p1,tp);

            // tz = B(tp)
            if ( pc != NULL )
                pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p1 = p1/normp
            normp = fasp_blas_darray_dotprod(m,tz,tp);
            normp = sqrt(normp);
            fasp_darray_cp(m,p1,t);

            // t0 = A*p0 = 0
            fasp_darray_set(m,t0,0.0);
            fasp_darray_cp(m,t0,z0);
            fasp_darray_cp(m,t0,t1);
            fasp_darray_cp(m,t0,z1);
            fasp_darray_cp(m,t0,p1);

            fasp_blas_darray_axpy(m,1/normp,t,p1);

            // t1=tp/normp,z1=tz/normp
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

        } // end of convergence check

        // update relative residual here
        absres0 = absres;

    } // end of the main loop

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m=b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, infnormu, factor;
    REAL         alpha, alpha0, alpha1, temp2;

    // allocate temp memory (need 11*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(11*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m;
    REAL *t0=z1+m, *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling MinRes solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // initialization counters
    stag=1; more_step=1; restart_step=1;

    // p0=0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    mf->fct(mf->data, u->val, r);
    fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

    // p1 = B(r)
    if (pc != NULL)
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_PRECRES:
            absres0=sqrt(ABS(fasp_blas_darray_dotprod(m,r,p1)));
            normr0=MAX(SMALLREAL,absres0);
            relres=absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0=fasp_blas_darray_norm2(m,r);
            normu2=MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres=absres0/normu2;
            break;
        default:  // STOP_REL_RES
            absres0=fasp_blas_darray_norm2(m,r);
            normr0=MAX(SMALLREAL,absres0);
            relres=absres0/normr0;
            break;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // tp=A*p1
    mf->fct(mf->data, p1, tp);

    // tz = B(tp)
    if (pc != NULL)
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1=p1/normp
    normp=ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp=sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0=A*p0=0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1=tp/normp,z1=tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    while( iter++ < MaxIt) {

        // alpha=<r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u=u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r=r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t=A*z1 alpha1=<z1,t>
        mf->fct(mf->data, z1, t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t=A*z0 alpha0=<z1,t>
        mf->fct(mf->data, z0, t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2=z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp=A*p2
        mf->fct(mf->data, p2, tp);

        // tz = B(tp)
        if (pc != NULL)
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2=p2/normp
        normp=ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp=sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);

        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        // relative residual = ||r||/||r0||
        temp2=fasp_blas_darray_dotprod(m,r,r);
        absres=sqrt(temp2);

        normu2=fasp_blas_darray_norm2(m,u->val);

        switch (StopType) {
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
                relres=sqrt(temp2)/normr0;
                break;
            case STOP_MOD_REL_RES:
                relres=sqrt(temp2)/normu2;
                break;
            default:  // STOP_REL_RES
                relres=sqrt(temp2)/normr0;
                break;
        }

        // compute reducation factor of residual ||r||
        factor=absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // solution check, if soultion is too small, return ERROR_SOLVER_SOLSTAG.
        infnormu = fasp_blas_darray_norminf(m, u->val);
        if ( infnormu <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        normuu=fasp_blas_darray_norm2(m,p1);
        normuu=ABS(alpha)*(normuu/normu2);

        // check convergence
        if (normuu<maxdiff) {
            if ( stag < MaxStag ) {
                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(normuu,relres);
                    ITS_RESTART;
                }
            }

            mf->fct(mf->data, u->val, r);
            fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

            temp2=fasp_blas_darray_dotprod(m,r,r);
            absres=sqrt(temp2);
            switch (StopType) {
                case STOP_REL_RES:
                    relres=sqrt(temp2)/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
                    relres=sqrt(temp2)/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    relres=sqrt(temp2)/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                ++stag;
                ++restart_step;

                fasp_darray_set(m,p0,0.0);

                // p1 = B(r)
                if (pc != NULL)
                    pc->fct(r,p1,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,p1); /* No preconditioner */

                // tp=A*p1
                mf->fct(mf->data, p1, tp);

                // tz = B(tp)
                if (pc == NULL)
                    pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                else
                    fasp_darray_cp(m,tp,tz); /* No preconditioner */

                // p1=p1/normp
                normp=fasp_blas_darray_dotprod(m,tz,tp);
                normp=sqrt(normp);
                fasp_darray_cp(m,p1,t);

                // t0=A*p0=0
                fasp_darray_set(m,t0,0.0);
                fasp_darray_cp(m,t0,z0);
                fasp_darray_cp(m,t0,t1);
                fasp_darray_cp(m,t0,z1);
                fasp_darray_cp(m,t0,p1);

                fasp_blas_darray_axpy(m,1/normp,t,p1);

                // t1=tp/normp,z1=tz/normp
                fasp_blas_darray_axpy(m,1/normp,tp,t1);
                fasp_blas_darray_axpy(m,1/normp,tz,z1);
            }
        }

        // safe guard
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            mf->fct(mf->data, u->val, r);
            fasp_blas_darray_axpby(m, 1.0, b->val, -1.0, r);

            temp2=fasp_blas_darray_dotprod(m,r,r);
            absres=sqrt(temp2);
            switch (StopType) {
                case STOP_REL_RES:
                    relres=sqrt(temp2)/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2=ABS(fasp_blas_darray_dotprod(m,r,t));
                    relres=sqrt(temp2)/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    relres=sqrt(temp2)/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            if ( more_step < MaxRestartStep ) {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
            }

            ++more_step;
            ++restart_step;

            fasp_darray_set(m,p0,0.0);

            // p1 = B(r)
            if (pc != NULL)
                pc->fct(r,p1,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,p1); /* No preconditioner */

            // tp = A*p1
            mf->fct(mf->data, p1, tp);

            // tz = B(tp)
            if (pc == NULL)
                pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p1 = p1/normp
            normp=fasp_blas_darray_dotprod(m,tz,tp);
            normp=sqrt(normp);
            fasp_darray_cp(m,p1,t);

            // t0=A*p0=0
            fasp_darray_set(m,t0,0.0);
            fasp_darray_cp(m,t0,z0);
            fasp_darray_cp(m,t0,t1);
            fasp_darray_cp(m,t0,z1);
            fasp_darray_cp(m,t0,p1);

            fasp_blas_darray_axpy(m,1/normp,t,p1);

            // t1=tp/normp,z1=tz/normp
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

        }

        // update relative residual here
        absres0 = absres;
    }

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.121 KryPvfgmres.c File Reference

Krylov subspace methods – Preconditioned variable-restarting FGMRes.

```
#include <math.h>
#include "fasp.h"
#include "fasp_functs.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 modifed from KryPvgmres.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.

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

A	Pointer to coefficient matrix
b	Pointer to right hand side vector
x	Pointer to solution vector
MaxIt	Maximal iteration number allowed
tol	Tolerance
pc	Pointer to preconditioner data
PrtLvl	How much information to print out
StopType	Stopping criterion, i.e.||r_k||/||r_0||<tol
restart	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,
             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

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition

Parameters

tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type – DO not support this parameter
PrtLvl	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,
            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

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type – DO not support this parameter
PrtLvl	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,
            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

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type – DO not support this parameter
PrtLvl	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.
```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const INT n                   = b->row;
    const INT min_iter            = 0;

    //---------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //---------------------------------------------//
    const REAL cr_max             = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min             = 0.174;   // = cos(80^o)  (experimental)

    // local variables
    INT    iter                   = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   epsmac                 = SMALLREAL;
    REAL   r_norm, b_norm, den_norm;
    REAL   epsilon, gamma, t;
    REAL   relres, normu, r_normb;

    REAL   *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
    REAL   **p = NULL, **hh = NULL, **z=NULL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart; // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    LONG worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

```

```
    /* allocate memory and setup temp work space */
    REAL *work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
    }

    p  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    z  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
    for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;

    /* initialization */
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);
    norms[0] = r_norm;

    if ( PrtLvl >= PRINT_SOME) {
        ITS_PUTNORM("right-hand side", b_norm);
        ITS_PUTNORM("residual", r_norm);
    }

    if ( b_norm > 0.0 ) den_norm = b_norm;
    else                den_norm = r_norm;

    epsilon = tol*den_norm;

    // if initial residual is small, no need to iterate!
    if ( r_norm < epsilon || r_norm < 1e-12*tol ) goto FINISHED;

    if ( b_norm > 0.0 ) {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
    }
    else {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
    }

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;
        r_norm_old = r_norm;
        if ( r_norm == 0.0 ) {
            fasp_mem_free(work);  work  = NULL;
            fasp_mem_free(p);     p     = NULL;
            fasp_mem_free(hh);    hh    = NULL;
            fasp_mem_free(norms); norms = NULL;
            fasp_mem_free(z);     z     = NULL;
            return iter;
        }

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//

        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) Restart -= d;
            else Restart = restart_max;
        }

        // Enter the cycle at the first iteration for at least one iteration
        t = 1.0 / r_norm;
        fasp_blas_darray_ax(n, t, p[0]);
        i = 0;

        // RESTART CYCLE (right-preconditioning)
        while ( i < Restart && iter < MaxIt ) {

            i ++;  iter ++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], z[i-1]);
            else
                pc->fct(p[i-1], z[i-1], pc->data);

            fasp_blas_dcsr_mxv(A, z[i-1], p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if ( t != 0.0 ) {
                t = 1.0 / t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] =  s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]   * hh[i][i-1];
            t += hh[i-1][i-1] * hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1] * rs[i-1];
            rs[i-1] =  c[i-1] * rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            r_norm = fabs(rs[i]);
            norms[iter] = r_norm;

            if ( b_norm > 0 ) {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
                            norms[iter], norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* Check:  Exit the restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */

        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, z[i-1], r);
        fasp_blas_darray_ax(n, rs[i-1], r);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if ( r_norm <= epsilon && iter >= min_iter ) {
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch (StopType) {
                case STOP_REL_RES:
                    relres  = r_norm/den_norm;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
                    else pc->fct(r, p[0], pc->data);
                    r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
                    relres  = r_normb/den_norm;
                    break;
                case STOP_MOD_REL_RES:
                    normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres  = r_norm/normu;
                    break;
                default:
                    printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
                    goto FINISHED;
            }

            if ( relres <= tol ) {
                break;
            }
            else {
                if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);

FINISHED:
    /*-------------------------------------------
* Free some stuff
*-------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);      p     = NULL;
    fasp_mem_free(hh);     hh    = NULL;
    fasp_mem_free(norms);  norms = NULL;
    fasp_mem_free(z);      z     = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
```

```
}

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)
{
    const INT n                  = b->row;
    const INT min_iter           = 0;

    //--------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //--------------------------------------------//
    const REAL cr_max           = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min           = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter                  = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   epsmac                = SMALLREAL;
    REAL   r_norm, b_norm, den_norm;
    REAL   epsilon, gamma, t;
    REAL   relres, normu, r_normb;

    REAL   *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
    REAL   **p = NULL, **hh = NULL, **z=NULL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart; // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    LONG worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    REAL *work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
    }

    p  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    z  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
    for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;

    /* initialization */
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dbsr_aAxpy(-1.0, A, x->val, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);
    norms[0] = r_norm;

    if ( PrtLvl >= PRINT_SOME) {
        ITS_PUTNORM("right-hand side", b_norm);
        ITS_PUTNORM("residual", r_norm);
    }

    if ( b_norm > 0.0 ) den_norm = b_norm;
    else                den_norm = r_norm;

    epsilon = tol*den_norm;

    // if initial residual is small, no need to iterate!
    if ( r_norm < epsilon || r_norm < 1e-12*tol ) goto FINISHED;

    if ( b_norm > 0.0 ) {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
    }
    else {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
    }

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;
        r_norm_old = r_norm;
        if ( r_norm == 0.0 ) {
            fasp_mem_free(work);  work  = NULL;
            fasp_mem_free(p);     p     = NULL;
            fasp_mem_free(hh);    hh    = NULL;
            fasp_mem_free(norms); norms = NULL;
            fasp_mem_free(z);     z     = NULL;
            return iter;
        }

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//

        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) Restart -= d;
            else Restart = restart_max;
        }

        // Enter the cycle at the first iteration for at least one iteration
        t = 1.0 / r_norm;
        fasp_blas_darray_ax(n, t, p[0]);
        i = 0;

        // RESTART CYCLE (right-preconditioning)
        while ( i < Restart && iter < MaxIt ) {

            i ++;  iter ++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], z[i-1]);
            else
                pc->fct(p[i-1], z[i-1], pc->data);

            fasp_blas_dbsr_mxv(A, z[i-1], p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if ( t != 0.0 ) {
                t = 1.0 / t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] =  s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]   * hh[i][i-1];
            t += hh[i-1][i-1] * hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1] * rs[i-1];
            rs[i-1] =  c[i-1] * rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            r_norm = fabs(rs[i]);
            norms[iter] = r_norm;

            if ( b_norm > 0 ) {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
                            norms[iter], norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* Check:  Exit the restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */

        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, z[i-1], r);
        fasp_blas_darray_ax(n, rs[i-1], r);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if ( r_norm <= epsilon && iter >= min_iter ) {
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dbsr_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch (StopType) {
                case STOP_REL_RES:
                    relres  = r_norm/den_norm;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
                    else pc->fct(r, p[0], pc->data);
                    r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
                    relres  = r_normb/den_norm;
                    break;
                case STOP_MOD_REL_RES:
                    normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres  = r_norm/normu;
                    break;
                default:
                    printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
                    goto FINISHED;
            }

            if ( relres <= tol ) {
                break;
            }
            else {
                if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);

FINISHED:
    /*-------------------------------------------
* Free some stuff
*-------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);      p     = NULL;
    fasp_mem_free(hh);     hh    = NULL;
    fasp_mem_free(norms);  norms = NULL;
    fasp_mem_free(z);      z     = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT n                  = b->row;
    const INT min_iter           = 0;

    //---------------------------------------------//
    //   Newly added parameters to monitor when    //
    //   to change the restart parameter           //
    //---------------------------------------------//
    const REAL cr_max            = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min            = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter                  = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   epsmac                = SMALLREAL;
    REAL   r_norm, b_norm, den_norm;
    REAL   epsilon, gamma, t;
```

```c
    REAL   relres, normu, r_normb;

    REAL   *c = NULL, *s = NULL, *rs = NULL, *norms = NULL, *r = NULL;
    REAL   **p = NULL, **hh = NULL, **z=NULL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart; // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    LONG worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    REAL *work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
    }

    p  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    z  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;
    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;
    for ( i = 0; i < Restart1; i++ ) z[i] = hh[Restart] + Restart + i*n;

    /* initialization */
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);
    norms[0] = r_norm;

    if ( PrtLvl >= PRINT_SOME) {
        ITS_PUTNORM("right-hand side", b_norm);
        ITS_PUTNORM("residual", r_norm);
    }

    if ( b_norm > 0.0 ) den_norm = b_norm;
    else                den_norm = r_norm;

    epsilon = tol*den_norm;

    // if initial residual is small, no need to iterate!
    if ( r_norm < epsilon || r_norm < 1e-12 * tol ) goto FINISHED;

    if ( b_norm > 0.0 ) {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm, norms[iter], 0);
    }
    else {
        fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter], 0);
    }

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;
        r_norm_old = r_norm;
        if ( r_norm == 0.0 ) {
            fasp_mem_free(work);  work  = NULL;
            fasp_mem_free(p);     p     = NULL;
            fasp_mem_free(hh);    hh    = NULL;
            fasp_mem_free(norms); norms = NULL;
            fasp_mem_free(z);     z     = NULL;
            return iter;
        }

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//

        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) Restart -= d;
            else Restart = restart_max;
        }

        // Enter the cycle at the first iteration for at least one iteration
        t = 1.0 / r_norm;
        fasp_blas_darray_ax(n, t, p[0]);
        i = 0;

        // RESTART CYCLE (right-preconditioning)
        while ( i < Restart && iter < MaxIt ) {

            i ++;  iter ++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], z[i-1]);
            else
                pc->fct(p[i-1], z[i-1], pc->data);

            fasp_blas_dblc_mxv(A, z[i-1], p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if ( t != 0.0 ) {
                t = 1.0 / t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for ( j = 1; j < i; ++j ) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] =  s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1]   = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t  = hh[i][i-1]   * hh[i][i-1];
            t += hh[i-1][i-1] * hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1] * rs[i-1];
            rs[i-1] =  c[i-1] * rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            r_norm = fabs(rs[i]);
            norms[iter] = r_norm;

            if ( b_norm > 0 ) {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
                            norms[iter], norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* Check:  Exit the restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */

        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j ++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, z[i-1], r);
        fasp_blas_darray_ax(n, rs[i-1], r);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], z[j], r);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if ( r_norm <= epsilon && iter >= min_iter ) {
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            switch (StopType) {
                case STOP_REL_RES:
                    relres  = r_norm/den_norm;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL ) fasp_darray_cp(n, r, p[0]);
                    else pc->fct(r, p[0], pc->data);
                    r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
                    relres  = r_normb/den_norm;
                    break;
                case STOP_MOD_REL_RES:
                    normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres  = r_norm/normu;
                    break;
                default:
                    printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
                    goto FINISHED;
            }

            if ( relres <= tol ) {
                break;
            }
            else {
                if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;
```

```

    } /* end of iteration while loop */

    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,r_norm/den_norm);

FINISHED:
    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);      p     = NULL;
    fasp_mem_free(hh);     hh    = NULL;
    fasp_mem_free(norms);  norms = NULL;
    fasp_mem_free(z);      z     = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT n                  = b->row;
    const INT min_iter           = 0;

    //---------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //---------------------------------------------//
    const REAL cr_max            = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min            = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter                  = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   epsmac                = SMALLREAL;
    REAL   r_norm, b_norm, den_norm;
    REAL   epsilon, gamma, t;

    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL;
    REAL  **p = NULL, **hh = NULL, **z=NULL;
    REAL  *work = NULL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart; // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    LONG worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VFGMRes solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n+Restart*n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vFGMRES restart number set to %d!\n", Restart);
    }

    p  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    z  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; rs = r + n; c = rs + Restart1; s = c + Restart;
    for (i = 0; i < Restart1; i ++) p[i] = s + Restart + i*n;
    for (i = 0; i < Restart1; i ++) hh[i] = p[Restart] + n + i*Restart;
    for (i = 0; i < Restart1; i ++) z[i] = hh[Restart] + Restart + i*n;

    /* initialization */
    mf->fct(mf->data, x->val, p[0]);
    fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);
    norms[0] = r_norm;

    if ( PrtLvl >= PRINT_SOME) {
        ITS_PUTNORM("right-hand side", b_norm);
        ITS_PUTNORM("residual", r_norm);
    }

    if (b_norm > 0.0)  den_norm = b_norm;
    else               den_norm = r_norm;

    epsilon = tol*den_norm;

    /* outer iteration cycle */
    while (iter < MaxIt) {
        rs[0] = r_norm;
        r_norm_old = r_norm;
        if (r_norm == 0.0) {
            fasp_mem_free(work);  work  = NULL;
            fasp_mem_free(p);     p     = NULL;
            fasp_mem_free(hh);    hh    = NULL;
            fasp_mem_free(norms); norms = NULL;
            fasp_mem_free(z);     z     = NULL;
            return iter;
        }

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//

        if (cr > cr_max || iter == 0) {
            Restart = restart_max;
        }
        else if (cr < cr_min) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) Restart -= d;
            else Restart = restart_max;
        }

        if (r_norm <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm <= epsilon) {
                break;
            }
            else {
                if (PrtLvl >= PRINT_SOME) ITS_FACONV;
            }
        }

        t = 1.0 / r_norm;
        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while (i < Restart && iter < MaxIt) {

            i ++;  iter ++;

            /* apply preconditioner */
            if (pc == NULL) fasp_darray_cp(n, p[i-1], z[i-1]);
            else pc->fct(p[i-1], z[i-1], pc->data);

            mf->fct(mf->data, z[i-1], p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            r_norm = fabs(rs[i]);
            norms[iter] = r_norm;

            if (b_norm > 0 ) {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter]/b_norm,
                            norms[iter], norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl, StopType, iter, norms[iter], norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* Check:  Exit restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */

        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, z[i-1], r);
        fasp_blas_darray_ax(n, rs[i-1], r);
        for (j = i-2; j >= 0; j --)  fasp_blas_darray_axpy(n, rs[j], z[j], r);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if (r_norm  <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
```

```
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm  <= epsilon) {
                break;
            }
            else {
                if (PrtLvl >= PRINT_SOME) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }
        } /* end of convergence check */


        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for (j = i-1 ; j > 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

    if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;
    fasp_mem_free(z);     z     = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, 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_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.

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,
            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.

Parameters

A	Pointer to dCSRmat: coefficient matrix

Parameters

b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             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.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             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.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             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.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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

mf	Pointer to mxv_matfree: spmv operation
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to precond: structure of precondition
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type – DOES not support this parameter
PrtLvl	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.

```
#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

```

```
INT fasp_solver_dcsr_pvgmres (dCSRmat      *A,
                              dvector      *b,
                              dvector      *x,
                              precond      *pc,
                              const REAL    tol,
                              const INT     MaxIt,
                              const SHORT   restart,
                              const SHORT   StopType,
                              const SHORT   PrtLvl)
{
    const INT   n           = b->row;
    const INT   MIN_ITER    = 0;
    const REAL  epsmac      = SMALLREAL;

    //-------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //-------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter              = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normu  = BIGREAL;

    REAL   cr           = 1.0;     // convergence rate
    REAL   r_norm_old   = 0.0;     // save residual norm of previous restart cycle
    INT    d            = 3;       // reduction for restart parameter
    INT    restart_max  = restart; // upper bound for restart in each restart cycle
    INT    restart_min  = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart;       // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dcsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute current residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT   n           = b->row;
    const INT   MIN_ITER    = 0;
    const REAL  epsmac      = SMALLREAL;

    //---------------------------------------------//
    //   Newly added parameters to monitor when    //
    //   to change the restart parameter           //
    //---------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o)  (experimental)

    // local variables
    INT    iter            = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normu  = BIGREAL;

    REAL   cr          = 1.0;      // convergence rate
    REAL   r_norm_old  = 0.0;      // save residual norm of previous restart cycle
    INT    d           = 3;        // reduction for restart parameter
    INT    restart_max = restart;  // upper bound for restart in each restart cycle
    INT    restart_min = 3;        // lower bound for restart in each restart cycle (should be small)

    INT  Restart   = restart;        // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dbsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dbsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute current residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dbsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

00757 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)
{
    const INT   n          = b->row;
    const INT   MIN_ITER   = 0;
    const REAL  epsmac     = SMALLREAL;

    //-----------------------------------------------//
    //   Newly added parameters to monitor when    //
    //   to change the restart parameter           //
    //-----------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)   (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o)  (experimental)

    // local variables
    INT    iter            = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normu  = BIGREAL;

    REAL   cr           = 1.0;     // convergence rate
    REAL   r_norm_old   = 0.0;     // save residual norm of previous restart cycle
    INT    d            = 3;       // reduction for restart parameter
    INT    restart_max  = restart; // upper bound for restart in each restart cycle
    INT    restart_min  = 3;       // lower bound for restart in each restart cycle (should be small)

    INT  Restart  = restart;       // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dblc_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute current residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
```

```
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);   work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
```

```
{
    const INT   n           = b->row;
    const INT   MIN_ITER    = 0;
    const REAL  epsmac      = SMALLREAL;

    //---------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter           //
    //---------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter            = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   absres0 = BIGREAL, absres = BIGREAL;
    REAL   relres  = BIGREAL, normu  = BIGREAL;

    REAL   cr           = 1.0;     // convergence rate
    REAL   r_norm_old   = 0.0;     // save residual norm of previous restart cycle
    INT    d            = 3;       // reduction for restart parameter
    INT    restart_max  = restart; // upper bound for restart in each restart cycle
    INT    restart_min  = 3;       // lower bound for restart in each restart cycle (should be small)

    INT  Restart  = restart;       // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    unsigned LONG worksize = (Restart+4)*(Restart+n)+1-n;

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
    }

    p     = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;

    for ( i = 0; i < Restart1; i++ ) p[i] = s + Restart + i*n;

    for ( i = 0; i < Restart1; i++ ) hh[i] = p[Restart] + n + i*Restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dstr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = MAX(SMALLREAL,r_norm);
            relres  = r_norm/absres0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            absres0 = MAX(SMALLREAL,r_normb);
            relres  = r_normb/absres0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            absres0 = r_norm;
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol || absres0 < 1e-12*tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,absres0,0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dstr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/absres0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres < tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // Check:  prevent false convergence
        if ( relres < tol && iter >= MIN_ITER ) {

            REAL computed_relres = relres;

            // compute current residual
            fasp_darray_cp(n, b->val, r);
            fasp_blas_dstr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/absres0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/absres0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres < tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_COMPRES(computed_relres); ITS_REALRES(relres);
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT n                  = b->row;
    const INT min_iter           = 0;

    //-------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //-------------------------------------------//
    const REAL cr_max           = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min           = 0.174;   // = cos(80^o)  (experimental)

    // local variables
    INT    iter                 = 0;
    int    i, j, k; // must be signed!  -zcs

    REAL   epsmac               = SMALLREAL;
    REAL   r_norm, b_norm, den_norm;
    REAL   epsilon, gamma, t;

    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  **p = NULL, **hh = NULL;
    REAL  *work = NULL;

    REAL   cr           = 1.0;     // convergence rate
    REAL   r_norm_old   = 0.0;     // save residual norm of previous restart cycle
    INT    d            = 3;       // reduction for restart parameter
    INT    restart_max  = restart; // upper bound for restart in each restart cycle
    INT    restart_min  = 3;       // lower bound for restart in each restart cycle

    INT  Restart  = restart;       // real restart in some fixed restarted cycle
    INT  Restart1 = Restart + 1;
    unsigned LONG worksize = (restart+4)*(restart+n)+1-n;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling VGMRes solver (MatFree) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (Restart > 5) ) {
        Restart = Restart - 5;
        worksize = (Restart+4)*(Restart+n)+1-n;
        work = (REAL *) fasp_mem_calloc(worksize, sizeof(REAL));
        Restart1 = Restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__ );
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    if ( PrtLvl > PRINT_MIN && Restart < restart ) {
        printf("### WARNING: vGMRES restart number set to %d!\n", Restart);
    }

    p  = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    hh = (REAL **)fasp_mem_calloc(Restart1, sizeof(REAL *));
    norms = (REAL *)fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + Restart1; s = c + Restart;
    for (i = 0; i < Restart1; i ++) p[i] = s + Restart + i*n;
    for (i = 0; i < Restart1; i ++) hh[i] = p[Restart] + n + i*Restart;

    /* initialization */
    mf->fct(mf->data, x->val, p[0]);
    fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, p[0]);

    b_norm = fasp_blas_darray_norm2(n, b->val);
    r_norm = fasp_blas_darray_norm2(n, p[0]);
    norms[0] = r_norm;

    if ( PrtLvl >= PRINT_SOME) {
        ITS_PUTNORM("right-hand side", b_norm);
        ITS_PUTNORM("residual", r_norm);
    }

    if (b_norm > 0.0)  den_norm = b_norm;
    else               den_norm = r_norm;

    epsilon = tol*den_norm;

    /* outer iteration cycle */
    while (iter < MaxIt) {
        rs[0] = r_norm;
        r_norm_old = r_norm;
        if (r_norm == 0.0) {
            fasp_mem_free(work);  work  = NULL;
            fasp_mem_free(p);     p     = NULL;
            fasp_mem_free(hh);    hh    = NULL;
            fasp_mem_free(norms); norms = NULL;
            return iter;
        }

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//

        if (cr > cr_max || iter == 0) {
            Restart = restart_max;
        }
        else if (cr < cr_min) {
            // Restart = Restart;
        }
        else {
            if (Restart - d > restart_min) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        if (r_norm <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm <= epsilon) {
                break;
            }
            else {
                if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
            }
        }

        t = 1.0 / r_norm;

        //for (j = 0; j < n; j ++) p[0][j] *= t;
        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while (i < Restart && iter < MaxIt) {

            i ++;  iter ++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            mf->fct(mf->data, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                //for (j = 0; j < n; j ++) p[i][j] *= t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];
            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];
            r_norm = fabs(rs[i]);

            norms[iter] = r_norm;

            if (b_norm > 0 ) {
                fasp_itinfo(PrtLvl,StopType,iter,norms[iter]/b_norm,
                            norms[iter],norms[iter]/norms[iter-1]);
            }
            else {
                fasp_itinfo(PrtLvl,StopType,iter,norms[iter],norms[iter],
                            norms[iter]/norms[iter-1]);
            }

            /* should we exit restart cycle?  */
            if (r_norm <= epsilon && iter >= min_iter) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */

        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }
        fasp_darray_cp(n, p[i-1], w);
        //for (j = 0; j < n; j ++) w[j] *= rs[i-1];
        fasp_blas_darray_ax(n, rs[i-1], w);
        for (j = i-2; j >= 0; j --)  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if (pc == NULL)
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        if (r_norm  <= epsilon && iter >= min_iter) {
            mf->fct(mf->data, x->val, r);
            fasp_blas_darray_axpby(n, 1.0, b->val, -1.0, r);
            r_norm = fasp_blas_darray_norm2(n, r);

            if (r_norm  <= epsilon) {
                break;
            }
            else {
                if ( PrtLvl >= PRINT_SOME ) ITS_FACONV;
                fasp_darray_cp(n, r, p[0]); i = 0;
            }
        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if (i) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for (j = i-1 ; j > 0; j --) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if (i) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //----------------------------------//
        //   compute the convergence rate    //
        //----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

    if (PrtLvl > PRINT_NONE) ITS_FINAL(iter,MaxIt,r_norm);

    /*-------------------------------------------
* Free some stuff
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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_spbcgs()

```
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.

Parameters

A	Pointer to dBLCmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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_spbcgs()

```
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.

Parameters

A	Pointer to dBSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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 KrySPbcgs.c.

9.125.2.3 fasp_solver_dcsr_spbcgs()

```
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.

Parameters

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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_spbcgs()

```
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.

Parameters

A	Pointer to dSTRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
    const REAL   TOL_s = tol*1e-2; // tolerance for norm(p)

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         alpha, beta, omega, temp1, temp2;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normd, tempr, normuinf;
    REAL         *uval = u->val, *bval = b->val;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 8*m REAL)
    REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
    REAL *p    = work,  *z  = work + m, *r = z  + m, *t  = r + m;
    REAL *rho  = t + m, *pp = rho  + m, *s = pp + m, *sp = s + m, *u_best = sp + m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,uval,r);
    absres0 = fasp_blas_darray_norm2(m,r);

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_RES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
            relres = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if (relres<tol) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // rho = r* := r
    fasp_darray_cp(m,r,rho);
    temp1 = fasp_blas_darray_dotprod(m,r,rho);

    // p = r
    fasp_darray_cp(m,r,p);

    // main BiCGstab loop
    while ( iter++ < MaxIt ) {

        // pp = precond(p)
        if ( pc != NULL )
            pc->fct(p,pp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,p,pp); /* No preconditioner */

        // z = A*pp
        fasp_blas_dcsr_mxv(A,pp,z);

        // alpha = (r,rho)/(A*p,rho)
        temp2 = fasp_blas_darray_dotprod(m,z,rho);
        if ( ABS(temp2) > SMALLREAL ) {
            alpha = temp1/temp2;
        }
        else {
            ITS_DIVZERO; goto FINISHED;
        }

        // s = r - alpha z
        fasp_darray_cp(m,r,s);
        fasp_blas_darray_axpy(m,-alpha,z,s);

        // sp = precond(s)
        if ( pc != NULL )
            pc->fct(s,sp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,s,sp); /* No preconditioner */

        // t = A*sp;
        fasp_blas_dcsr_mxv(A,sp,t);

        // omega = (t,s)/(t,t)
        tempr = fasp_blas_darray_dotprod(m,t,t);

        if ( ABS(tempr) > SMALLREAL ) {
            omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
        }
        else {
            omega = 0.0;
            if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
        }

        // delu = alpha pp + omega sp
        fasp_blas_darray_axpby(m,alpha,pp,omega,sp);
        // u = u + delu
        fasp_blas_darray_axpy(m,1.0,sp,uval);

        // r = s - omega t
        fasp_blas_darray_axpy(m,-omega,t,s);
        fasp_darray_cp(m,s,r);

        // beta = (r,rho)/(rp,rho)
        temp2 = temp1;
        temp1 = fasp_blas_darray_dotprod(m,r,rho);

        if ( ABS(temp2) > SMALLREAL ) {
            beta = (temp1*alpha)/(temp2*omega);
        }
        else {
            ITS_DIVZERO; goto RESTORE_BESTSOL;
        }

        // p = p - omega z
        fasp_blas_darray_axpy(m,-omega,z,p);

        // p = r + beta p
        fasp_blas_darray_axpby(m,1.0,r,beta,p);

        // compute difference
        normd   = fasp_blas_darray_norm2(m,sp);
        normu   = fasp_blas_darray_norm2(m,uval);
        reldiff = normd/normu;

        if ( normd < TOL_s ) {
            ITS_SMALLSP; goto FINISHED;
        }

        // compute residuals
        switch (StopType) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if ( pc == NULL )
                    fasp_darray_cp(m,r,z);
                else
                    pc->fct(r,z,pc->data);
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restrore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,uval,u_best);
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, uval);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            goto FINISHED;
        }

        // Check II: if staggenated, try to restart
        if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    goto FINISHED;
                }
                ++stag;
                ++restart_step;
            }

        } // end of stagnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = tempr/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                goto FINISHED;
            }
            else {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
            }

            ++more_step;
            ++restart_step;
        } // end if safe guard

        absres0 = absres;

    } // end of main BiCGstab loop

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED: // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
```

```
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
    const REAL   TOL_s = tol*1e-2; // tolerance for norm(p)

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         alpha, beta, omega, temp1, temp2;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normd, tempr, normuinf;
    REAL         *uval = u->val, *bval = b->val;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 8*m REAL)
    REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
    REAL *p    = work,  *z  = work + m, *r = z  + m, *t  = r + m;
    REAL *rho  = t + m, *pp = rho  + m, *s = pp + m, *sp = s + m, *u_best = sp + m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    fasp_blas_dbsr_aAxpy(-1.0,A,uval,r);
    absres0 = fasp_blas_darray_norm2(m,r);

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            normr0  = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
            relres = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if (relres<tol) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // rho = r* := r
    fasp_darray_cp(m,r,rho);
    temp1 = fasp_blas_darray_dotprod(m,r,rho);

    // p = r
    fasp_darray_cp(m,r,p);

    // main BiCGstab loop
    while ( iter++ < MaxIt ) {

        // pp = precond(p)
        if ( pc != NULL )
            pc->fct(p,pp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,p,pp); /* No preconditioner */

        // z = A*pp
        fasp_blas_dbsr_mxv(A,pp,z);

        // alpha = (r,rho)/(A*p,rho)
        temp2 = fasp_blas_darray_dotprod(m,z,rho);
        if ( ABS(temp2) > SMALLREAL ) {
            alpha = temp1/temp2;
        }
        else {
            ITS_DIVZERO; goto FINISHED;
        }

        // s = r - alpha z
        fasp_darray_cp(m,r,s);
        fasp_blas_darray_axpy(m,-alpha,z,s);

        // sp = precond(s)
        if ( pc != NULL )
            pc->fct(s,sp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,s,sp); /* No preconditioner */

        // t = A*sp;
        fasp_blas_dbsr_mxv(A,sp,t);

        // omega = (t,s)/(t,t)
        tempr = fasp_blas_darray_dotprod(m,t,t);

        if ( ABS(tempr) > SMALLREAL ) {
            omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
        }
        else {
            omega = 0.0;
            if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
        }

        // delu = alpha pp + omega sp
        fasp_blas_darray_axpby(m,alpha,pp,omega,sp);

        // u = u + delu
        fasp_blas_darray_axpy(m,1.0,sp,uval);

        // r = s - omega t
        fasp_blas_darray_axpy(m,-omega,t,s);
        fasp_darray_cp(m,s,r);

        // beta = (r,rho)/(rp,rho)
        temp2 = temp1;
        temp1 = fasp_blas_darray_dotprod(m,r,rho);

        if ( ABS(temp2) > SMALLREAL ) {
            beta = (temp1*alpha)/(temp2*omega);
        }
        else {
            ITS_DIVZERO; goto RESTORE_BESTSOL;
        }

        // p = p - omega z
        fasp_blas_darray_axpy(m,-omega,z,p);

        // p = r + beta p
        fasp_blas_darray_axpby(m,1.0,r,beta,p);

        // compute difference
        normd   = fasp_blas_darray_norm2(m,sp);
        normu   = fasp_blas_darray_norm2(m,uval);
        reldiff = normd/normu;

        if ( normd < TOL_s ) {
            ITS_SMALLSP; goto FINISHED;
        }

        // compute residuals
        switch (StopType) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if ( pc == NULL )
                    fasp_darray_cp(m,r,z);
                else
                    pc->fct(r,z,pc->data);
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restrore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,uval,u_best);
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, uval);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            goto FINISHED;
        }

        // Check II: if staggenated, try to restart
        if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dbsr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    goto FINISHED;
                }
                ++stag;
                ++restart_step;
            }

        } // end of stagnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dbsr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = tempr/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                goto FINISHED;
            }
            else {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
            }

            ++more_step;
            ++restart_step;
        } // end if safe guard

        absres0 = absres;

    } // end of main BiCGstab loop

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dbsr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED: // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
    const REAL   TOL_s = tol*1e-2; // tolerance for norm(p)

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         alpha, beta, omega, temp1, temp2;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normd, tempr, normuinf;
    REAL         *uval = u->val, *bval = b->val;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 8*m REAL)
    REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
    REAL *p    = work,  *z  = work + m, *r = z  + m, *t  = r + m;
    REAL *rho  = t + m, *pp = rho  + m, *s = pp + m, *sp = s + m, *u_best = sp + m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    fasp_blas_dblc_aAxpy(-1.0,A,uval,r);
    absres0 = fasp_blas_darray_norm2(m,r);

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_RES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
            relres = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if (relres<tol) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // rho = r* := r
    fasp_darray_cp(m,r,rho);
    temp1 = fasp_blas_darray_dotprod(m,r,rho);

    // p = r
    fasp_darray_cp(m,r,p);

    // main BiCGstab loop
    while ( iter++ < MaxIt ) {

        // pp = precond(p)
        if ( pc != NULL )
            pc->fct(p,pp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,p,pp); /* No preconditioner */

        // z = A*pp
        fasp_blas_dblc_mxv(A,pp,z);

        // alpha = (r,rho)/(A*p,rho)
        temp2 = fasp_blas_darray_dotprod(m,z,rho);
        if ( ABS(temp2) > SMALLREAL ) {
            alpha = temp1/temp2;
        }
        else {
            ITS_DIVZERO; goto FINISHED;
        }

        // s = r - alpha z
        fasp_darray_cp(m,r,s);
        fasp_blas_darray_axpy(m,-alpha,z,s);

        // sp = precond(s)
        if ( pc != NULL )
            pc->fct(s,sp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,s,sp); /* No preconditioner */

        // t = A*sp;
        fasp_blas_dblc_mxv(A,sp,t);

        // omega = (t,s)/(t,t)
        tempr = fasp_blas_darray_dotprod(m,t,t);

        if ( ABS(tempr) > SMALLREAL ) {
            omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
        }
        else {
            omega = 0.0;
            if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
        }

        // delu = alpha pp + omega sp
        fasp_blas_darray_axpby(m,alpha,pp,omega,sp);

        // u = u + delu
        fasp_blas_darray_axpy(m,1.0,sp,uval);

        // r = s - omega t
        fasp_blas_darray_axpy(m,-omega,t,s);
        fasp_darray_cp(m,s,r);

        // beta = (r,rho)/(rp,rho)
        temp2 = temp1;
        temp1 = fasp_blas_darray_dotprod(m,r,rho);

        if ( ABS(temp2) > SMALLREAL ) {
            beta = (temp1*alpha)/(temp2*omega);
        }
        else {
            ITS_DIVZERO; goto RESTORE_BESTSOL;
        }

        // p = p - omega z
        fasp_blas_darray_axpy(m,-omega,z,p);

        // p = r + beta p
        fasp_blas_darray_axpby(m,1.0,r,beta,p);

        // compute difference
        normd   = fasp_blas_darray_norm2(m,sp);
        normu   = fasp_blas_darray_norm2(m,uval);
        reldiff = normd/normu;

        if ( normd < TOL_s ) {
            ITS_SMALLSP; goto FINISHED;
        }

        // compute residuals
        switch (StopType) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if ( pc == NULL )
                    fasp_darray_cp(m,r,z);
                else
                    pc->fct(r,z,pc->data);
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restrore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,uval,u_best);
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, uval);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            goto FINISHED;
        }

        // Check II: if staggenated, try to restart
        if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dblc_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    goto FINISHED;
                }
                ++stag;
                ++restart_step;
            }

        } // end of stagnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dblc_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
```

```
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = tempr/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                goto FINISHED;
            }
            else {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
            }

            ++more_step;
            ++restart_step;
        } // end if safe guard

        absres0 = absres;

    } // end of main BiCGstab loop

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dblc_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

```

```
FINISHED: // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance
    const REAL   TOL_s = tol*1e-2; // tolerance for norm(p)

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         alpha, beta, omega, temp1, temp2;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normd, tempr, normuinf;
    REAL         *uval = u->val, *bval = b->val;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 8*m REAL)
    REAL *work = (REAL *)fasp_mem_calloc(9*m,sizeof(REAL));
    REAL *p    = work,  *z  = work + m, *r = z  + m, *t  = r + m;
    REAL *rho  = t + m, *pp = rho  + m, *s = pp + m, *sp = s + m, *u_best = sp + m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe BiCGstab solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,bval,r);
    fasp_blas_dstr_aAxpy(-1.0,A,uval,r);
    absres0 = fasp_blas_darray_norm2(m,r);

    // compute initial relative residual
    switch (StopType) {
        case STOP_REL_RES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            normr0 = MAX(SMALLREAL,absres0);
            relres = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,uval));
            relres = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if (relres<tol) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // rho = r* := r
    fasp_darray_cp(m,r,rho);
    temp1 = fasp_blas_darray_dotprod(m,r,rho);

    // p = r
    fasp_darray_cp(m,r,p);

    // main BiCGstab loop
    while ( iter++ < MaxIt ) {

        // pp = precond(p)
        if ( pc != NULL )
            pc->fct(p,pp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,p,pp); /* No preconditioner */

        // z = A*pp
        fasp_blas_dstr_mxv(A,pp,z);

        // alpha = (r,rho)/(A*p,rho)
        temp2 = fasp_blas_darray_dotprod(m,z,rho);
        if ( ABS(temp2) > SMALLREAL ) {
            alpha = temp1/temp2;
        }
        else {
            ITS_DIVZERO; goto FINISHED;
        }

        // s = r - alpha z
        fasp_darray_cp(m,r,s);
        fasp_blas_darray_axpy(m,-alpha,z,s);

        // sp = precond(s)
        if ( pc != NULL )
            pc->fct(s,sp,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,s,sp); /* No preconditioner */

        // t = A*sp;
        fasp_blas_dstr_mxv(A,sp,t);

        // omega = (t,s)/(t,t)
        tempr = fasp_blas_darray_dotprod(m,t,t);

        if ( ABS(tempr) > SMALLREAL ) {
            omega = fasp_blas_darray_dotprod(m,s,t)/tempr;
        }
        else {
            omega = 0.0;
            if ( PrtLvl >= PRINT_SOME ) ITS_DIVZERO;
        }

        // delu = alpha pp + omega sp
        fasp_blas_darray_axpby(m,alpha,pp,omega,sp);

        // u = u + delu
        fasp_blas_darray_axpy(m,1.0,sp,uval);

        // r = s - omega t
        fasp_blas_darray_axpy(m,-omega,t,s);
        fasp_darray_cp(m,s,r);

        // beta = (r,rho)/(rp,rho)
        temp2 = temp1;
        temp1 = fasp_blas_darray_dotprod(m,r,rho);

        if ( ABS(temp2) > SMALLREAL ) {
            beta = (temp1*alpha)/(temp2*omega);
        }
        else {
            ITS_DIVZERO; goto RESTORE_BESTSOL;
        }

        // p = p - omega z
        fasp_blas_darray_axpy(m,-omega,z,p);

        // p = r + beta p
        fasp_blas_darray_axpby(m,1.0,r,beta,p);

        // compute difference
        normd   = fasp_blas_darray_norm2(m,sp);
        normu   = fasp_blas_darray_norm2(m,uval);
        reldiff = normd/normu;

        if ( normd < TOL_s ) {
            ITS_SMALLSP; goto FINISHED;
        }

        // compute residuals
        switch (StopType) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if ( pc == NULL )
                    fasp_darray_cp(m,r,z);
                else
                    pc->fct(r,z,pc->data);
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restrore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,uval,u_best);
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, uval);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            goto FINISHED;
        }

        // Check II: if staggenated, try to restart
        if ( (stag <= MaxStag) && (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dstr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    goto FINISHED;
                }
                ++stag;
                ++restart_step;
            }

        } // end of stagnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {
            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // re-init iteration param
            fasp_darray_cp(m,bval,r);
            fasp_blas_dstr_aAxpy(-1.0,A,uval,r);

            // pp = precond(p)
            fasp_darray_cp(m,r,p);
            if ( pc != NULL )
                pc->fct(p,pp,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,p,pp); /* No preconditioner */

            // rho = r* := r
            fasp_darray_cp(m,r,rho);
            temp1 = fasp_blas_darray_dotprod(m,r,rho);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data);
                    else
                        fasp_darray_cp(m,r,z);
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,r,z)));
                    relres = tempr/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                goto FINISHED;
            }
            else {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTART;
            }

            ++more_step;
            ++restart_step;
        } // end if safe guard

        absres0 = absres;

    } // end of main BiCGstab loop

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dstr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED: // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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,
            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.

Parameters

A	Pointer to dBLCmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations

Parameters

StopType	Stopping criteria type
PrtLvl	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,
            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.

Parameters

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
pc	Pointer to the structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            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.

Parameters

A	Pointer to dSTRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
u	Pointer to dvector: the unknowns
MaxIt	Maximal number of iterations
tol	Tolerance for stopping
pc	Pointer to the structure of precondition (precond)
StopType	Stopping criteria type
PrtLvl	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_functs.h"
00028
00029 /*---------------------------------*/
00030 /*--  Declare Private Functions  --*/
00031 /*---------------------------------*/
```

```

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 5*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(5*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe CG solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

    if (pc != NULL)
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t=A*p
        fasp_blas_dcsr_mxv(A,p,t);

        // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            goto RESTORE_BESTSOL;
        }

        // u_k=u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k=r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // if the solution is NAN, restore the best solution
        if ( fasp_dvec_isnan(u) ) {
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        // safety net check:  save the best-so-far solution
        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,u->val,u_best);
        }

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, u->val);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        // Check II: if staggenated, try to restart
        normu   = fasp_blas_dvec_norm2(u);

        // compute relative difference
        reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
        if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                fasp_darray_set(m,p,0.0);
                ++stag;
                ++restart_step;
            }
        } // end of staggnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting the method
            fasp_darray_set(m,p,0.0);
            ++more_step;
            ++restart_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:  // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 5*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(5*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe CG solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

    if (pc != NULL)
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t=A*p
        fasp_blas_dblc_mxv(A,p,t);

        // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            goto RESTORE_BESTSOL;
        }

        // u_k=u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k=r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // if the solution is NAN, restore the best solution
        if ( fasp_dvec_isnan(u) ) {
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        // safety net check:  save the best-so-far solution
        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,u->val,u_best);
        }

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, u->val);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        // Check II: if staggenated, try to restart
        normu   = fasp_blas_dvec_norm2(u);

        // compute relative difference
        reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
        if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            fasp_darray_cp(m,b->val,r);
            fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                fasp_darray_set(m,p,0.0);
                ++stag;
                ++restart_step;
            }
        } // end of staggnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting the method
            fasp_darray_set(m,p,0.0);
            ++more_step;
            ++restart_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dblc_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:  // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         relres  = BIGREAL, normu  = BIGREAL, normr0 = BIGREAL;
    REAL         reldiff, factor, normuinf;
    REAL         alpha, beta, temp1, temp2;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 5*m REAL numbers)
    REAL *work = (REAL *)fasp_mem_calloc(5*m,sizeof(REAL));
    REAL *p = work, *z = work+m, *r = z+m, *t = r+m, *u_best = t+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe CG solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

    if (pc != NULL)
        pc->fct(r,z,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,z); /* No preconditioner */

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,z));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    fasp_darray_cp(m,z,p);
    temp1 = fasp_blas_darray_dotprod(m,z,r);

    // main PCG loop
    while ( iter++ < MaxIt ) {

        // t=A*p
        fasp_blas_dstr_mxv(A,p,t);

        // alpha_k=(z_{k-1},r_{k-1})/(A*p_{k-1},p_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,t,p);
        if ( ABS(temp2) > SMALLREAL2 ) {
            alpha = temp1/temp2;
        }
        else { // Possible breakdown
            goto RESTORE_BESTSOL;
        }

        // u_k=u_{k-1} + alpha_k*p_{k-1}
        fasp_blas_darray_axpy(m,alpha,p,u->val);

        // r_k=r_{k-1} - alpha_k*A*p_{k-1}
        fasp_blas_darray_axpy(m,-alpha,t,r);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres = fasp_blas_darray_norm2(m,r);
                relres = absres/normu;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // if the solution is NAN, restore the best solution
        if ( fasp_dvec_isnan(u) ) {
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        // safety net check:  save the best-so-far solution
        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,u->val,u_best);
        }

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, u->val);
        if ( normuinf <= sol_inf_tol ) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        // Check II: if staggenated, try to restart
        normu   = fasp_blas_dvec_norm2(u);

        // compute relative difference
        reldiff = ABS(alpha)*fasp_blas_darray_norm2(m,p)/normu;
        if ( (stag <= MaxStag) & (reldiff < maxdiff) ) {

            if ( PrtLvl >= PRINT_MORE ) {
                ITS_DIFFRES(reldiff,relres);
                ITS_RESTART;
            }

            fasp_darray_cp(m,b->val,r);
            fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                fasp_darray_set(m,p,0.0);
                ++stag;
                ++restart_step;
            }
        } // end of staggnation check!

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    // z = B(r)
                    if ( pc != NULL )
                        pc->fct(r,z,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,z); /* No preconditioner */
                    absres = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = fasp_blas_darray_norm2(m,r);
                    relres = absres/normu;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting the method
            fasp_darray_set(m,p,0.0);
            ++more_step;
            ++restart_step;

        } // end of safe-guard check!

        // save residual for next iteration
        absres0 = absres;

        // compute z_k = B(r_k)
        if ( StopType != STOP_REL_PRECRES ) {
            if ( pc != NULL )
                pc->fct(r,z,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,z); /* No preconditioner, B=I */
        }

        // compute beta_k = (z_k, r_k)/(z_{k-1}, r_{k-1})
        temp2 = fasp_blas_darray_dotprod(m,z,r);
        beta  = temp2/temp1;
        temp1 = temp2;

        // compute p_k = z_k + beta_k*p_{k-1}
        fasp_blas_darray_axpby(m,1.0,z,beta,p);

    } // end of main PCG loop.

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dstr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,z,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,z); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,z,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:  // finish the iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.129 KrySPgmres.c File Reference

Krylov subspace methods – Preconditioned GMRes with safety net.

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

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,
            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.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping

Parameters

MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
             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.

Parameters

A	Pointer to dBSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
            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.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
              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.

Parameters

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const INT  n          = b->row;
    const INT  MIN_ITER   = 0;
    const REAL maxdiff    = tol*STAG_RATIO; // staganation tolerance
    const REAL epsmac     = SMALLREAL;

    // local variables
    INT      iter = 0;
    INT      restart1 = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     normr0 = BIGREAL, absres = BIGREAL;
    REAL     relres = BIGREAL, normu  = BIGREAL;

    INT      iter_best = 0; // initial best known iteration
    REAL     absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL, *x_best = NULL;
    REAL    **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: GMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dcsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT  n          = b->row;
    const INT  MIN_ITER   = 0;
    const REAL maxdiff    = tol*STAG_RATIO; // staganation tolerance
    const REAL epsmac     = SMALLREAL;

    // local variables
    INT      iter = 0;
    INT      restart1 = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     normr0 = BIGREAL, absres = BIGREAL;
    REAL     relres = BIGREAL, normu  = BIGREAL;

    INT      iter_best = 0; // initial best known iteration
    REAL     absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL, *x_best = NULL;
    REAL    **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: GMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dbsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
```

```
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dbsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dbsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dbsr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT  n          = b->row;
    const INT  MIN_ITER   = 0;
    const REAL maxdiff    = tol*STAG_RATIO; // staganation tolerance
    const REAL epsmac     = SMALLREAL;

    // local variables
    INT      iter = 0;
    INT      restart1 = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     normr0 = BIGREAL, absres = BIGREAL;
    REAL     relres = BIGREAL, normu  = BIGREAL;

    INT      iter_best = 0; // initial best known iteration
    REAL     absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL    *c = NULL, *s = NULL, *rs = NULL;
    REAL    *norms = NULL, *r = NULL, *w = NULL;
    REAL    *work = NULL, *x_best = NULL;
    REAL    **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: GMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, p[0]);

    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dblc_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dblc_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Clean up workspace
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT  n         = b->row;
    const INT  MIN_ITER  = 0;
    const REAL maxdiff   = tol*STAG_RATIO; // staganation tolerance
    const REAL epsmac    = SMALLREAL;

    // local variables
    INT      iter = 0;
    INT      restart1 = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL     r_norm, r_normb, gamma, t;
    REAL     normr0 = BIGREAL, absres = BIGREAL;
    REAL     relres = BIGREAL, normu  = BIGREAL;

    INT      iter_best = 0; // initial best known iteration
    REAL     absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL     *c = NULL, *s = NULL, *rs = NULL;
    REAL     *norms = NULL, *r = NULL, *w = NULL;
    REAL     *work = NULL, *x_best = NULL;
    REAL     **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe GMRes solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: GMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dstr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm  = fasp_blas_darray_norm2(n,p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < restart && iter < MaxIt ) {

            i++; iter++;

            /* apply preconditioner */
            if ( pc == NULL )
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dstr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for ( j = 0; j < i; j++ ) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] =  c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for ( k = i-2; k >= 0; k-- ) {
            t = 0.0;
            for (j = k+1; j < i; j++) t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dstr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for (j = i; j > 0; j--) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j] = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

    } /* end of main while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dstr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
 * Clean up workspace
 *-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter >= MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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_spminres (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_spminres (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_spminres (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

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_spminres()

```
INT fasp_solver_dblc_spminres (
            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.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns

Parameters

pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
            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.

Parameters

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
StopType	Stopping criteria type
PrtLvl	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,
                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.

Parameters

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
u	Pointer to dvector: unknowns
MaxIt	Maximal number of iterations
tol	Tolerance for stopping
pc	Pointer to structure of precondition (precond)
StopType	Stopping criteria type
PrtLvl	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
```

```
#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dcsr_spminres (const dCSRmat  *A,
                               const dvector  *b,
                               dvector        *u,
                               precond        *pc,
                               const REAL      tol,
                               const INT       MaxIt,
                               const SHORT     StopType,
                               const SHORT     PrtLvl)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, normuinf, factor;
    REAL         alpha, alpha0, alpha1, temp2;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 12*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(12*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
    REAL *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m, *u_best = r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // p0 = 0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

    // p1 = B(r)
    if ( pc != NULL )
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu2;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // tp = A*p1
    fasp_blas_dcsr_mxv(A,p1,tp);

    // tz = B(tp)
    if ( pc != NULL )
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1 = p1/normp
    normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp = sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0 = A*p0 = 0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1 = tp/normp, z1 = tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    // main MinRes loop
    while ( iter++ < MaxIt ) {

        // alpha = <r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u = u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r = r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t = A*z1 alpha1 = <z1,t>
        fasp_blas_dcsr_mxv(A,z1,t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t = A*z0 alpha0 = <z1,t>
        fasp_blas_dcsr_mxv(A,z0,t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2 = z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp = A*p2
        fasp_blas_dcsr_mxv(A,p2,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2 = p2/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);
        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        normu2 = fasp_blas_darray_norm2(m,u->val);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normu2;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,u->val,u_best);
        }

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, u->val);
        if (normuinf <= sol_inf_tol) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
            break;
        }

        // Check II: if staggenated, try to restart
        normuu = fasp_blas_darray_norm2(m,p1);
        normuu = ABS(alpha)*(normuu/normu2);

        if ( normuu < maxdiff ) {

            if ( stag < MaxStag ) {
                if ( PrtLvl >= PRINT_MORE ) {
                    ITS_DIFFRES(normuu,relres);
                    ITS_RESTART;
                }
            }

            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            if ( relres < tol )
                break;
            else {
                if ( stag >= MaxStag ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                    iter = ERROR_SOLVER_STAG;
                    break;
                }
                fasp_darray_set(m,p0,0.0);
                ++stag;
                ++restart_step;

                // p1 = B(r)
                if ( pc != NULL )
                    pc->fct(r,p1,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,p1); /* No preconditioner */

                // tp = A*p1
                fasp_blas_dcsr_mxv(A,p1,tp);

                // tz = B(tp)
                if ( pc != NULL )
                    pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                else
                    fasp_darray_cp(m,tp,tz); /* No preconditioner */

                // p1 = p1/normp
                normp = fasp_blas_darray_dotprod(m,tz,tp);
                normp = sqrt(normp);
                fasp_darray_cp(m,p1,t);

                // t0 = A*p0=0
                fasp_darray_set(m,t0,0.0);
                fasp_darray_cp(m,t0,z0);
                fasp_darray_cp(m,t0,t1);
                fasp_darray_cp(m,t0,z1);
                fasp_darray_cp(m,t0,p1);

                fasp_blas_darray_axpy(m,1/normp,t,p1);

                // t1 = tp/normp, z1 = tz/normp
                fasp_blas_darray_axpy(m,1/normp,tp,t1);
                fasp_blas_darray_axpy(m,1/normp,tz,z1);
            }
        }

        // Check III: prevent false convergence
        if ( relres < tol ) {

            if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

            // compute residual r = b - Ax again
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dcsr_aAxpy(-1.0,A,u->val,r);

            // compute residuals
            switch (StopType) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

            // check convergence
            if ( relres < tol ) break;

            if ( more_step >= MaxRestartStep ) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                iter = ERROR_SOLVER_TOLSMALL;
                break;
            }

            // prepare for restarting method
            fasp_darray_set(m,p0,0.0);
            ++more_step;
            ++restart_step;

            // p1 = B(r)
            if ( pc != NULL )
                pc->fct(r,p1,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,r,p1); /* No preconditioner */

            // tp = A*p1
            fasp_blas_dcsr_mxv(A,p1,tp);

            // tz = B(tp)
            if ( pc != NULL )
                pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p1 = p1/normp
            normp = fasp_blas_darray_dotprod(m,tz,tp);
            normp = sqrt(normp);
            fasp_darray_cp(m,p1,t);

            // t0 = A*p0 = 0
            fasp_darray_set(m,t0,0.0);
            fasp_darray_cp(m,t0,z0);
            fasp_darray_cp(m,t0,t1);
            fasp_darray_cp(m,t0,z1);
            fasp_darray_cp(m,t0,p1);

            fasp_blas_darray_axpy(m,1/normp,t,p1);

            // t1=tp/normp,z1=tz/normp
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

        } // end of convergence check

        // update relative residual here
        absres0 = absres;

    } // end of the main loop

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,u_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
            case STOP_REL_PRECRES:
                if ( pc != NULL )
                    pc->fct(r,t,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,t); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(m,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(m,u_best,u->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:  // finish iterative method
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    // clean up temp memory
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

INT fasp_solver_dblc_spminres (const dBLCmat  *A,
                               const dvector  *b,
                               dvector        *u,
                               precond        *pc,
                               const REAL      tol,
                               const INT       MaxIt,
                               const SHORT     StopType,
                               const SHORT     PrtLvl)
{
    const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
    const INT    m = b->row;
    const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
    const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

    // local variables
    INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
    REAL         absres0 = BIGREAL, absres = BIGREAL;
    REAL         normr0  = BIGREAL, relres  = BIGREAL;
    REAL         normu2, normuu, normp, normuinf, factor;
    REAL         alpha, alpha0, alpha1, temp2;
    INT          iter_best = 0; // initial best known iteration
    REAL         absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need 12*m REAL)
    REAL *work=(REAL *)fasp_mem_calloc(12*m,sizeof(REAL));
    REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
    REAL *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m, *u_best = r+m;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    // p0 = 0
    fasp_darray_set(m,p0,0.0);

    // r = b-A*u
    fasp_darray_cp(m,b->val,r);
    fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

    // p1 = B(r)
    if ( pc != NULL )
        pc->fct(r,p1,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,r,p1); /* No preconditioner */

    // compute initial residuals
    switch ( StopType ) {
        case STOP_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_REL_PRECRES:
            absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
            normr0  = MAX(SMALLREAL,absres0);
            relres  = absres0/normr0;
            break;
        case STOP_MOD_REL_RES:
            absres0 = fasp_blas_darray_norm2(m,r);
            normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
            relres  = absres0/normu2;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

    // tp = A*p1
    fasp_blas_dblc_mxv(A,p1,tp);

    // tz = B(tp)
    if ( pc != NULL )
        pc->fct(tp,tz,pc->data); /* Apply preconditioner */
    else
        fasp_darray_cp(m,tp,tz); /* No preconditioner */

    // p1 = p1/normp
    normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
    normp = sqrt(normp);
    fasp_darray_cp(m,p1,t);
    fasp_darray_set(m,p1,0.0);
    fasp_blas_darray_axpy(m,1/normp,t,p1);

    // t0 = A*p0 = 0
    fasp_darray_set(m,t0,0.0);
    fasp_darray_cp(m,t0,z0);
    fasp_darray_cp(m,t0,t1);
    fasp_darray_cp(m,t0,z1);

    // t1 = tp/normp, z1 = tz/normp
    fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
    fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

    // main MinRes loop
    while ( iter++ < MaxIt ) {

        // alpha = <r,z1>
        alpha=fasp_blas_darray_dotprod(m,r,z1);

        // u = u+alpha*p1
        fasp_blas_darray_axpy(m,alpha,p1,u->val);

        // r = r-alpha*Ap1
        fasp_blas_darray_axpy(m,-alpha,t1,r);

        // compute t = A*z1 alpha1 = <z1,t>
        fasp_blas_dblc_mxv(A,z1,t);
        alpha1=fasp_blas_darray_dotprod(m,z1,t);

        // compute t = A*z0 alpha0 = <z1,t>
        fasp_blas_dblc_mxv(A,z0,t);
        alpha0=fasp_blas_darray_dotprod(m,z1,t);

        // p2 = z1-alpha1*p1-alpha0*p0
        fasp_darray_cp(m,z1,p2);
        fasp_blas_darray_axpy(m,-alpha1,p1,p2);
        fasp_blas_darray_axpy(m,-alpha0,p0,p2);

        // tp = A*p2
        fasp_blas_dblc_mxv(A,p2,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p2 = p2/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p2,t);
        fasp_darray_set(m,p2,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p2);

        // prepare for next iteration
        fasp_darray_cp(m,p1,p0);
        fasp_darray_cp(m,p2,p1);
        fasp_darray_cp(m,t1,t0);
        fasp_darray_cp(m,z1,z0);

        // t1=tp/normp,z1=tz/normp
        fasp_darray_set(m,t1,0.0);
        fasp_darray_cp(m,t1,z1);
        fasp_blas_darray_axpy(m,1/normp,tp,t1);
        fasp_blas_darray_axpy(m,1/normp,tz,z1);

        normu2 = fasp_blas_darray_norm2(m,u->val);

        // compute residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_REL_PRECRES:
                if (pc == NULL)
                    fasp_darray_cp(m,r,t);
                else
                    pc->fct(r,t,pc->data);
                temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                absres = sqrt(temp2);
                relres = absres/normr0;
                break;
            case STOP_MOD_REL_RES:
                temp2  = fasp_blas_darray_dotprod(m,r,r);
                absres = sqrt(temp2);
                relres = absres/normu2;
                break;
        }

        // compute reducation factor of residual ||r||
        factor = absres/absres0;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(u) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(m,u->val,u_best);
        }

        // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
        normuinf = fasp_blas_darray_norminf(m, u->val);
        if (normuinf <= sol_inf_tol) {
            if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
            iter = ERROR_SOLVER_SOLSTAG;
```

```
                break;
            }

            // Check II: if staggenated, try to restart
            normuu = fasp_blas_darray_norm2(m,p1);
            normuu = ABS(alpha)*(normuu/normu2);

            if ( normuu < maxdiff ) {

                if ( stag < MaxStag ) {
                    if ( PrtLvl >= PRINT_MORE ) {
                        ITS_DIFFRES(normuu,relres);
                        ITS_RESTART;
                    }
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p0,0.0);
                    ++stag;
                    ++restart_step;

                    // p1 = B(r)
                    if ( pc != NULL )
                        pc->fct(r,p1,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,p1); /* No preconditioner */

                    // tp = A*p1
                    fasp_blas_dblc_mxv(A,p1,tp);

                    // tz = B(tp)
                    if ( pc != NULL )
                        pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                    else
                        fasp_darray_cp(m,tp,tz); /* No preconditioner */

                    // p1 = p1/normp
                    normp = fasp_blas_darray_dotprod(m,tz,tp);
                    normp = sqrt(normp);
                    fasp_darray_cp(m,p1,t);

                    // t0 = A*p0=0
                    fasp_darray_set(m,t0,0.0);
                    fasp_darray_cp(m,t0,z0);
                    fasp_darray_cp(m,t0,t1);
                    fasp_darray_cp(m,t0,z1);
                    fasp_darray_cp(m,t0,p1);

                    fasp_blas_darray_axpy(m,1/normp,t,p1);

                    // t1 = tp/normp, z1 = tz/normp
                    fasp_blas_darray_axpy(m,1/normp,tp,t1);
                    fasp_blas_darray_axpy(m,1/normp,tz,z1);
                }
            }

            // Check III: prevent false convergence
            if ( relres < tol ) {

                if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

                // compute residual r = b - Ax again
                fasp_darray_cp(m,b->val,r);
                fasp_blas_dblc_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                // check convergence
                if ( relres < tol ) break;

                if ( more_step >= MaxRestartStep ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                    iter = ERROR_SOLVER_TOLSMALL;
                    break;
                }

                // prepare for restarting method
                fasp_darray_set(m,p0,0.0);
                ++more_step;
                ++restart_step;

                // p1 = B(r)
                if ( pc != NULL )
                    pc->fct(r,p1,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,p1); /* No preconditioner */

                // tp = A*p1
                fasp_blas_dblc_mxv(A,p1,tp);

                // tz = B(tp)
                if ( pc != NULL )
                    pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                else
                    fasp_darray_cp(m,tp,tz); /* No preconditioner */

                // p1 = p1/normp
                normp = fasp_blas_darray_dotprod(m,tz,tp);
                normp = sqrt(normp);
                fasp_darray_cp(m,p1,t);

                // t0 = A*p0 = 0
                fasp_darray_set(m,t0,0.0);
                fasp_darray_cp(m,t0,z0);
                fasp_darray_cp(m,t0,t1);
                fasp_darray_cp(m,t0,z1);
                fasp_darray_cp(m,t0,p1);

                fasp_blas_darray_axpy(m,1/normp,t,p1);

                // t1=tp/normp,z1=tz/normp
                fasp_blas_darray_axpy(m,1/normp,tp,t1);
                fasp_blas_darray_axpy(m,1/normp,tz,z1);

            } // end of convergence check

            // update relative residual here
            absres0 = absres;

        } // end of the main loop

    RESTORE_BESTSOL: // restore the best-so-far solution if necessary
        if ( iter != iter_best ) {

            // compute best residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dblc_aAxpy(-1.0,A,u_best,r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres_best = fasp_blas_darray_norm2(m,r);
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,t,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,t); /* No preconditioner */
                    absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
                    break;
                case STOP_MOD_REL_RES:
                    absres_best = fasp_blas_darray_norm2(m,r);
                    break;
            }

            if ( absres > absres_best + maxdiff || isnan(absres) ) {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
                fasp_darray_cp(m,u_best,u->val);
                relres = absres_best / normr0;
            }
        }

    FINISHED:  // finish iterative method
        if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

        // clean up temp memory
        fasp_mem_free(work); work = NULL;

    #if DEBUG_MODE > 0
        printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
    #endif

        if ( iter > MaxIt )
            return ERROR_SOLVER_MAXIT;
        else
            return iter;
    }

    INT fasp_solver_dstr_spminres (const dSTRmat  *A,
                                   const dvector  *b,
                                   dvector        *u,
                                   precond        *pc,
                                   const REAL      tol,
                                   const INT       MaxIt,
                                   const SHORT     StopType,
                                   const SHORT     PrtLvl)
    {
        const SHORT  MaxStag = MAX_STAG, MaxRestartStep = MAX_RESTART;
        const INT    m = b->row;
        const REAL   maxdiff = tol*STAG_RATIO; // staganation tolerance
        const REAL   sol_inf_tol = SMALLREAL; // infinity norm tolerance

        // local variables
        INT          iter = 0, stag = 1, more_step = 1, restart_step = 1;
        REAL         absres0 = BIGREAL, absres = BIGREAL;
        REAL         normr0  = BIGREAL, relres  = BIGREAL;
        REAL         normu2, normuu, normp, normuinf, factor;
        REAL         alpha, alpha0, alpha1, temp2;
        INT          iter_best = 0; // initial best known iteration
        REAL         absres_best = BIGREAL; // initial best known residual

        // allocate temp memory (need 12*m REAL)
        REAL *work=(REAL *)fasp_mem_calloc(12*m,sizeof(REAL));
        REAL *p0=work, *p1=work+m, *p2=p1+m, *z0=p2+m, *z1=z0+m, *t0=z1+m;
        REAL *t1=t0+m, *t=t1+m, *tp=t+m, *tz=tp+m, *r=tz+m, *u_best = r+m;

        // Output some info for debuging
        if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe MinRes solver (STR) ...\n");

    #if DEBUG_MODE > 0
        printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
        printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
    #endif

        // p0 = 0
        fasp_darray_set(m,p0,0.0);

        // r = b-A*u
        fasp_darray_cp(m,b->val,r);
        fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

        // p1 = B(r)
        if ( pc != NULL )
            pc->fct(r,p1,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,r,p1); /* No preconditioner */

        // compute initial residuals
        switch ( StopType ) {
            case STOP_REL_RES:
                absres0 = fasp_blas_darray_norm2(m,r);
                normr0  = MAX(SMALLREAL,absres0);
                relres  = absres0/normr0;
                break;
            case STOP_REL_PRECRES:
                absres0 = sqrt(fasp_blas_darray_dotprod(m,r,p1));
                normr0  = MAX(SMALLREAL,absres0);
                relres  = absres0/normr0;
                break;
            case STOP_MOD_REL_RES:
                absres0 = fasp_blas_darray_norm2(m,r);
                normu2  = MAX(SMALLREAL,fasp_blas_darray_norm2(m,u->val));
                relres  = absres0/normu2;
                break;
            default:
                printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
                goto FINISHED;
        }

        // if initial residual is small, no need to iterate!
        if ( relres < tol ) goto FINISHED;

        // output iteration information if needed
        fasp_itinfo(PrtLvl,StopType,iter,relres,absres0,0.0);

        // tp = A*p1
        fasp_blas_dstr_mxv(A,p1,tp);

        // tz = B(tp)
        if ( pc != NULL )
            pc->fct(tp,tz,pc->data); /* Apply preconditioner */
        else
            fasp_darray_cp(m,tp,tz); /* No preconditioner */

        // p1 = p1/normp
        normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
        normp = sqrt(normp);
        fasp_darray_cp(m,p1,t);
        fasp_darray_set(m,p1,0.0);
        fasp_blas_darray_axpy(m,1/normp,t,p1);

        // t0 = A*p0 = 0
        fasp_darray_set(m,t0,0.0);
        fasp_darray_cp(m,t0,z0);
        fasp_darray_cp(m,t0,t1);
        fasp_darray_cp(m,t0,z1);

        // t1 = tp/normp, z1 = tz/normp
        fasp_blas_darray_axpy(m,1.0/normp,tp,t1);
        fasp_blas_darray_axpy(m,1.0/normp,tz,z1);

        // main MinRes loop
        while ( iter++ < MaxIt ) {

            // alpha = <r,z1>
            alpha=fasp_blas_darray_dotprod(m,r,z1);

            // u = u+alpha*p1
            fasp_blas_darray_axpy(m,alpha,p1,u->val);

            // r = r-alpha*Ap1
            fasp_blas_darray_axpy(m,-alpha,t1,r);

            // compute t = A*z1 alpha1 = <z1,t>
            fasp_blas_dstr_mxv(A,z1,t);
            alpha1=fasp_blas_darray_dotprod(m,z1,t);

            // compute t = A*z0 alpha0 = <z1,t>
            fasp_blas_dstr_mxv(A,z0,t);
            alpha0=fasp_blas_darray_dotprod(m,z1,t);

            // p2 = z1-alpha1*p1-alpha0*p0
            fasp_darray_cp(m,z1,p2);
            fasp_blas_darray_axpy(m,-alpha1,p1,p2);
            fasp_blas_darray_axpy(m,-alpha0,p0,p2);

            // tp = A*p2
            fasp_blas_dstr_mxv(A,p2,tp);

            // tz = B(tp)
            if ( pc != NULL )
                pc->fct(tp,tz,pc->data); /* Apply preconditioner */
            else
                fasp_darray_cp(m,tp,tz); /* No preconditioner */

            // p2 = p2/normp
            normp = ABS(fasp_blas_darray_dotprod(m,tz,tp));
            normp = sqrt(normp);
            fasp_darray_cp(m,p2,t);
            fasp_darray_set(m,p2,0.0);
            fasp_blas_darray_axpy(m,1/normp,t,p2);

            // prepare for next iteration
            fasp_darray_cp(m,p1,p0);
            fasp_darray_cp(m,p2,p1);
            fasp_darray_cp(m,t1,t0);
            fasp_darray_cp(m,z1,z0);

            // t1=tp/normp,z1=tz/normp
            fasp_darray_set(m,t1,0.0);
            fasp_darray_cp(m,t1,z1);
            fasp_blas_darray_axpy(m,1/normp,tp,t1);
            fasp_blas_darray_axpy(m,1/normp,tz,z1);

            normu2 = fasp_blas_darray_norm2(m,u->val);

            // compute residuals
            switch ( StopType ) {
                case STOP_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if (pc == NULL)
                        fasp_darray_cp(m,r,t);
                    else
                        pc->fct(r,t,pc->data);
                    temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                    absres = sqrt(temp2);
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    temp2  = fasp_blas_darray_dotprod(m,r,r);
                    absres = sqrt(temp2);
                    relres = absres/normu2;
                    break;
            }

            // compute reducation factor of residual ||r||
            factor = absres/absres0;

            // output iteration information if needed
            fasp_itinfo(PrtLvl,StopType,iter,relres,absres,factor);

            // safety net check:  save the best-so-far solution
            if ( fasp_dvec_isnan(u) ) {
                // If the solution is NAN, restore the best solution
                absres = BIGREAL;
                goto RESTORE_BESTSOL;
            }

            if ( absres < absres_best - maxdiff) {
                absres_best = absres;
                iter_best   = iter;
                fasp_darray_cp(m,u->val,u_best);
            }

            // Check I: if soultion is close to zero, return ERROR_SOLVER_SOLSTAG
            normuinf = fasp_blas_darray_norminf(m, u->val);
            if (normuinf <= sol_inf_tol) {
                if ( PrtLvl > PRINT_MIN ) ITS_ZEROSOL;
                iter = ERROR_SOLVER_SOLSTAG;
                break;
            }

            // Check II: if staggenated, try to restart
            normuu = fasp_blas_darray_norm2(m,p1);
            normuu = ABS(alpha)*(normuu/normu2);

            if ( normuu < maxdiff ) {

                if ( stag < MaxStag ) {
                    if ( PrtLvl >= PRINT_MORE ) {
                        ITS_DIFFRES(normuu,relres);
                        ITS_RESTART;
                    }
                }

                fasp_darray_cp(m,b->val,r);
                fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                if ( relres < tol )
                    break;
                else {
                    if ( stag >= MaxStag ) {
                        if ( PrtLvl > PRINT_MIN ) ITS_STAGGED;
                        iter = ERROR_SOLVER_STAG;
                        break;
                    }
                    fasp_darray_set(m,p0,0.0);
                    ++stag;
                    ++restart_step;

                    // p1 = B(r)
                    if ( pc != NULL )
                        pc->fct(r,p1,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,p1); /* No preconditioner */

                    // tp = A*p1
                    fasp_blas_dstr_mxv(A,p1,tp);

                    // tz = B(tp)
                    if ( pc != NULL )
                        pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                    else
                        fasp_darray_cp(m,tp,tz); /* No preconditioner */

                    // p1 = p1/normp
                    normp = fasp_blas_darray_dotprod(m,tz,tp);
                    normp = sqrt(normp);
                    fasp_darray_cp(m,p1,t);

                    // t0 = A*p0=0
                    fasp_darray_set(m,t0,0.0);
                    fasp_darray_cp(m,t0,z0);
                    fasp_darray_cp(m,t0,t1);
                    fasp_darray_cp(m,t0,z1);
                    fasp_darray_cp(m,t0,p1);

                    fasp_blas_darray_axpy(m,1/normp,t,p1);

                    // t1 = tp/normp, z1 = tz/normp
                    fasp_blas_darray_axpy(m,1/normp,tp,t1);
                    fasp_blas_darray_axpy(m,1/normp,tz,z1);
                }
            }

            // Check III: prevent false convergence
            if ( relres < tol ) {

                if ( PrtLvl >= PRINT_MORE ) ITS_COMPRES(relres);

                // compute residual r = b - Ax again
                fasp_darray_cp(m,b->val,r);
                fasp_blas_dstr_aAxpy(-1.0,A,u->val,r);

                // compute residuals
                switch (StopType) {
                    case STOP_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_REL_PRECRES:
                        if (pc == NULL)
                            fasp_darray_cp(m,r,t);
                        else
                            pc->fct(r,t,pc->data);
                        temp2  = ABS(fasp_blas_darray_dotprod(m,r,t));
                        absres = sqrt(temp2);
                        relres = absres/normr0;
                        break;
                    case STOP_MOD_REL_RES:
                        temp2  = fasp_blas_darray_dotprod(m,r,r);
                        absres = sqrt(temp2);
                        relres = absres/normu2;
                        break;
                }

                if ( PrtLvl >= PRINT_MORE ) ITS_REALRES(relres);

                // check convergence
                if ( relres < tol ) break;

                if ( more_step >= MaxRestartStep ) {
                    if ( PrtLvl > PRINT_MIN ) ITS_ZEROTOL;
                    iter = ERROR_SOLVER_TOLSMALL;
                    break;
                }

                // prepare for restarting method
                fasp_darray_set(m,p0,0.0);
                ++more_step;
                ++restart_step;

                // p1 = B(r)
                if ( pc != NULL )
                    pc->fct(r,p1,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(m,r,p1); /* No preconditioner */

                // tp = A*p1
                fasp_blas_dstr_mxv(A,p1,tp);

                // tz = B(tp)
                if ( pc != NULL )
                    pc->fct(tp,tz,pc->data); /* Apply rreconditioner */
                else
                    fasp_darray_cp(m,tp,tz); /* No preconditioner */

                // p1 = p1/normp
                normp = fasp_blas_darray_dotprod(m,tz,tp);
                normp = sqrt(normp);
                fasp_darray_cp(m,p1,t);

                // t0 = A*p0 = 0
                fasp_darray_set(m,t0,0.0);
                fasp_darray_cp(m,t0,z0);
                fasp_darray_cp(m,t0,t1);
                fasp_darray_cp(m,t0,z1);
                fasp_darray_cp(m,t0,p1);

                fasp_blas_darray_axpy(m,1/normp,t,p1);

                // t1=tp/normp,z1=tz/normp
                fasp_blas_darray_axpy(m,1/normp,tp,t1);
                fasp_blas_darray_axpy(m,1/normp,tz,z1);

            } // end of convergence check

            // update relative residual here
            absres0 = absres;

        } // end of the main loop

    RESTORE_BESTSOL: // restore the best-so-far solution if necessary
        if ( iter != iter_best ) {

            // compute best residual
            fasp_darray_cp(m,b->val,r);
            fasp_blas_dstr_aAxpy(-1.0,A,u_best,r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres_best = fasp_blas_darray_norm2(m,r);
                    break;
                case STOP_REL_PRECRES:
                    if ( pc != NULL )
                        pc->fct(r,t,pc->data); /* Apply preconditioner */
                    else
                        fasp_darray_cp(m,r,t); /* No preconditioner */
                    absres_best = sqrt(ABS(fasp_blas_darray_dotprod(m,t,r)));
                    break;
                case STOP_MOD_REL_RES:
                    absres_best = fasp_blas_darray_norm2(m,r);
                    break;
            }

            if ( absres > absres_best + maxdiff || isnan(absres) ) {
                if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
                fasp_darray_cp(m,u_best,u->val);
                relres = absres_best / normr0;
            }
        }

    FINISHED:  // finish iterative method
        if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

        // clean up temp memory
        fasp_mem_free(work); work = NULL;

    #if DEBUG_MODE > 0
        printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
    #endif
```

```

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

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 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.

Parameters

A	Pointer to dBLCmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
              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

A	Pointer to dBSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
                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

A	Pointer to dCSRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns

Parameters

pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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,
            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

A	Pointer to dSTRmat: coefficient matrix
b	Pointer to dvector: right hand side
x	Pointer to dvector: unknowns
pc	Pointer to structure of precondition (precond)
tol	Tolerance for stopping
MaxIt	Maximal number of iterations
restart	Restarting steps
StopType	Stopping criteria type
PrtLvl	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public 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)
{
    const INT   n           = b->row;
    const INT   MIN_ITER    = 0;
    const REAL  maxdiff     = tol*STAG_RATIO; // staganation tolerance
    const REAL  epsmac      = SMALLREAL;

    //-----------------------------------------------//
    //   Newly added parameters to monitor when    //
    //   to change the restart parameter           //
    //-----------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter            = 0;
    INT    restart1    = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   normr0 = BIGREAL, absres = BIGREAL;
    REAL   relres = BIGREAL, normu  = BIGREAL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart (should be small)
    INT    Restart     = restart; // real restart in some fixed restarted cycle

    INT    iter_best = 0;         // initial best known iteration
    REAL   absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL, *x_best = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (CSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: vGMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dcsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dcsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;
        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dcsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

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)
{
    const INT   n          = b->row;
    const INT   MIN_ITER   = 0;
    const REAL  maxdiff    = tol*STAG_RATIO; // staganation tolerance
    const REAL  epsmac     = SMALLREAL;

    //-------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //-------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter            = 0;
    INT    restart1     = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   normr0 = BIGREAL, absres = BIGREAL;
    REAL   relres = BIGREAL, normu  = BIGREAL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart (should be small)
    INT    Restart     = restart; // real restart in some fixed restarted cycle

    INT    iter_best = 0;         // initial best known iteration
    REAL   absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL   *c = NULL, *s = NULL, *rs = NULL;
    REAL   *norms = NULL, *r = NULL, *w = NULL;
    REAL   *work = NULL, *x_best = NULL;
    REAL   **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (BSR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: vGMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dbsr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
            case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
            case STOP_REL_PRECRES:
            if ( pc == NULL )
            fasp_darray_cp(n, p[0], r);
            else
            pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
            case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
            default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
            fasp_darray_cp(n, p[i-1], r);
            else
            pc->fct(p[i-1], r, pc->data);

            fasp_blas_dbsr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
        fasp_darray_cp(n, w, r);
        else
        pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dbsr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                    case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                    case STOP_REL_PRECRES:
                    if ( pc == NULL )
                    fasp_darray_cp(n, r, w);
                    else
                    pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                    case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dbsr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
                case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
                case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
                case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
```

```c
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*-------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
    return ERROR_SOLVER_MAXIT;
    else
    return iter;
}

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)
{
    const INT   n           = b->row;
    const INT   MIN_ITER    = 0;
    const REAL  maxdiff     = tol*STAG_RATIO; // staganation tolerance
    const REAL  epsmac      = SMALLREAL;

    //-------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //-------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter            = 0;
    INT    restart1    = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   normr0 = BIGREAL, absres = BIGREAL;
    REAL   relres = BIGREAL, normu  = BIGREAL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart (should be small)
    INT    Restart     = restart; // real restart in some fixed restarted cycle

    INT    iter_best = 0;         // initial best known iteration
    REAL   absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL   *c = NULL, *s = NULL, *rs = NULL;
    REAL   *norms = NULL, *r = NULL, *w = NULL;
    REAL   *work = NULL, *x_best = NULL;
    REAL   **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (BLC) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: vGMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dblc_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0   = MAX(SMALLREAL,r_norm);
            relres   = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0   = MAX(SMALLREAL,r_normb);
            relres   = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu    = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0   = r_norm;
            relres   = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dblc_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dblc_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dblc_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

01210 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)
{
    const INT    n          = b->row;
    const INT    MIN_ITER   = 0;
    const REAL   maxdiff    = tol*STAG_RATIO; // staganation tolerance
    const REAL   epsmac     = SMALLREAL;

    //---------------------------------------------//
    //   Newly added parameters to monitor when   //
    //   to change the restart parameter          //
    //---------------------------------------------//
    const REAL cr_max      = 0.99;    // = cos(8^o)  (experimental)
    const REAL cr_min      = 0.174;   // = cos(80^o) (experimental)

    // local variables
    INT    iter            = 0;
    INT    restart1    = restart + 1;
    int      i, j, k; // must be signed!  -zcs

    REAL   r_norm, r_normb, gamma, t;
    REAL   normr0 = BIGREAL, absres = BIGREAL;
    REAL   relres = BIGREAL, normu  = BIGREAL;

    REAL   cr          = 1.0;     // convergence rate
    REAL   r_norm_old  = 0.0;     // save residual norm of previous restart cycle
    INT    d           = 3;       // reduction for restart parameter
    INT    restart_max = restart; // upper bound for restart in each restart cycle
    INT    restart_min = 3;       // lower bound for restart (should be small)
    INT    Restart     = restart; // real restart in some fixed restarted cycle

    INT    iter_best = 0;         // initial best known iteration
    REAL   absres_best = BIGREAL; // initial best known residual

    // allocate temp memory (need about (restart+4)*n REAL numbers)
    REAL  *c = NULL, *s = NULL, *rs = NULL;
    REAL  *norms = NULL, *r = NULL, *w = NULL;
    REAL  *work = NULL, *x_best = NULL;
    REAL  **p = NULL, **hh = NULL;

    // Output some info for debuging
    if ( PrtLvl > PRINT_NONE ) printf("\nCalling Safe VGMRes solver (STR) ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: maxit = %d, tol = %.4le\n", MaxIt, tol);
#endif

    /* allocate memory and setup temp work space */
    work  = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));

    /* check whether memory is enough for GMRES */
    while ( (work == NULL) && (restart > 5 ) ) {
        restart = restart - 5 ;
        work = (REAL *) fasp_mem_calloc((restart+4)*(restart+n)+1, sizeof(REAL));
        printf("### WARNING: vGMRES restart number set to %d!\n", restart);
        restart1 = restart + 1;
    }

    if ( work == NULL ) {
        printf("### ERROR: No enough memory!  [%s:%d]\n", __FILE__, __LINE__);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    p     = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    hh    = (REAL **)fasp_mem_calloc(restart1, sizeof(REAL *));
    norms = (REAL *) fasp_mem_calloc(MaxIt+1, sizeof(REAL));

    r = work; w = r + n; rs = w + n; c = rs + restart1;
    x_best = c + restart; s = x_best + n;

    for ( i = 0; i < restart1; i++ ) p[i] = s + restart + i*n;

    for ( i = 0; i < restart1; i++ ) hh[i] = p[restart] + n + i*restart;

    // r = b-A*x
    fasp_darray_cp(n, b->val, p[0]);
    fasp_blas_dstr_aAxpy(-1.0, A, x->val, p[0]);

    r_norm = fasp_blas_darray_norm2(n, p[0]);

    // compute initial residuals
    switch (StopType) {
        case STOP_REL_RES:
            normr0  = MAX(SMALLREAL,r_norm);
            relres  = r_norm/normr0;
            break;
        case STOP_REL_PRECRES:
            if ( pc == NULL )
                fasp_darray_cp(n, p[0], r);
            else
                pc->fct(p[0], r, pc->data);
            r_normb = sqrt(fasp_blas_darray_dotprod(n,p[0],r));
            normr0  = MAX(SMALLREAL,r_normb);
            relres  = r_normb/normr0;
            break;
        case STOP_MOD_REL_RES:
            normu   = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
            normr0  = r_norm;
            relres  = normr0/normu;
            break;
        default:
            printf("### ERROR: Unknown stopping type!  [%s]\n", __FUNCTION__);
            goto FINISHED;
    }

    // if initial residual is small, no need to iterate!
    if ( relres < tol ) goto FINISHED;

    // output iteration information if needed
    fasp_itinfo(PrtLvl,StopType,0,relres,normr0,0.0);

    // store initial residual
    norms[0] = relres;

    /* outer iteration cycle */
    while ( iter < MaxIt ) {

        rs[0] = r_norm_old = r_norm;

        t = 1.0 / r_norm;

        fasp_blas_darray_ax(n, t, p[0]);

        //-----------------------------------//
        //   adjust the restart parameter    //
        //-----------------------------------//
        if ( cr > cr_max || iter == 0 ) {
            Restart = restart_max;
        }
        else if ( cr < cr_min ) {
            // Restart = Restart;
        }
        else {
            if ( Restart - d > restart_min ) {
                Restart -= d;
            }
            else {
                Restart = restart_max;
            }
        }

        /* RESTART CYCLE (right-preconditioning) */
        i = 0;
        while ( i < Restart && iter < MaxIt ) {

            i++;  iter++;

            /* apply preconditioner */
            if (pc == NULL)
                fasp_darray_cp(n, p[i-1], r);
            else
                pc->fct(p[i-1], r, pc->data);

            fasp_blas_dstr_mxv(A, r, p[i]);

            /* modified Gram_Schmidt */
            for (j = 0; j < i; j ++) {
                hh[j][i-1] = fasp_blas_darray_dotprod(n, p[j], p[i]);
                fasp_blas_darray_axpy(n, -hh[j][i-1], p[j], p[i]);
            }
            t = fasp_blas_darray_norm2(n, p[i]);
            hh[i][i-1] = t;
            if (t != 0.0) {
                t = 1.0/t;
                fasp_blas_darray_ax(n, t, p[i]);
            }

            for (j = 1; j < i; ++j) {
                t = hh[j-1][i-1];
                hh[j-1][i-1] = s[j-1]*hh[j][i-1] + c[j-1]*t;
                hh[j][i-1] = -s[j-1]*t + c[j-1]*hh[j][i-1];
            }
            t= hh[i][i-1]*hh[i][i-1];
            t+= hh[i-1][i-1]*hh[i-1][i-1];

            gamma = sqrt(t);
            if (gamma == 0.0) gamma = epsmac;
            c[i-1]  = hh[i-1][i-1] / gamma;
            s[i-1]  = hh[i][i-1] / gamma;
            rs[i]   = -s[i-1]*rs[i-1];
            rs[i-1] = c[i-1]*rs[i-1];
            hh[i-1][i-1] = s[i-1]*hh[i][i-1] + c[i-1]*hh[i-1][i-1];

            absres = r_norm = fabs(rs[i]);

            relres = absres/normr0;

            norms[iter] = relres;

            // output iteration information if needed
            fasp_itinfo(PrtLvl, StopType, iter, relres, absres,
                        norms[iter]/norms[iter-1]);

            // should we exit restart cycle
            if ( relres <= tol && iter >= MIN_ITER ) break;

        } /* end of restart cycle */

        /* now compute solution, first solve upper triangular system */
        rs[i-1] = rs[i-1] / hh[i-1][i-1];
        for (k = i-2; k >= 0; k --) {
            t = 0.0;
            for (j = k+1; j < i; j ++)  t -= hh[k][j]*rs[j];

            t += rs[k];
            rs[k] = t / hh[k][k];
        }

        fasp_darray_cp(n, p[i-1], w);

        fasp_blas_darray_ax(n, rs[i-1], w);

        for ( j = i-2; j >= 0; j-- )  fasp_blas_darray_axpy(n, rs[j], p[j], w);

        /* apply preconditioner */
        if ( pc == NULL )
            fasp_darray_cp(n, w, r);
        else
            pc->fct(w, r, pc->data);

        fasp_blas_darray_axpy(n, 1.0, r, x->val);

        // safety net check:  save the best-so-far solution
        if ( fasp_dvec_isnan(x) ) {
            // If the solution is NAN, restore the best solution
            absres = BIGREAL;
            goto RESTORE_BESTSOL;
        }

        if ( absres < absres_best - maxdiff) {
            absres_best = absres;
            iter_best   = iter;
            fasp_darray_cp(n,x->val,x_best);
        }

        // Check:  prevent false convergence
        if ( relres <= tol && iter >= MIN_ITER ) {

            fasp_darray_cp(n, b->val, r);
            fasp_blas_dstr_aAxpy(-1.0, A, x->val, r);

            r_norm = fasp_blas_darray_norm2(n, r);

            switch ( StopType ) {
                case STOP_REL_RES:
                    absres = r_norm;
                    relres = absres/normr0;
                    break;
                case STOP_REL_PRECRES:
                    if ( pc == NULL )
                        fasp_darray_cp(n, r, w);
                    else
                        pc->fct(r, w, pc->data);
                    absres = sqrt(fasp_blas_darray_dotprod(n,w,r));
                    relres = absres/normr0;
                    break;
                case STOP_MOD_REL_RES:
                    absres = r_norm;
                    normu  = MAX(SMALLREAL,fasp_blas_darray_norm2(n,x->val));
                    relres = absres/normu;
                    break;
            }

            norms[iter] = relres;

            if ( relres <= tol ) {
                break;
            }
            else {
                // Need to restart
                fasp_darray_cp(n, r, p[0]); i = 0;
            }

        } /* end of convergence check */

        /* compute residual vector and continue loop */
        for ( j = i; j > 0; j-- ) {
            rs[j-1] = -s[j-1]*rs[j];
            rs[j]   = c[j-1]*rs[j];
        }

        if ( i ) fasp_blas_darray_axpy(n, rs[i]-1.0, p[i], p[i]);

        for ( j = i-1 ; j > 0; j-- ) fasp_blas_darray_axpy(n, rs[j], p[j], p[i]);

        if ( i ) {
            fasp_blas_darray_axpy(n, rs[0]-1.0, p[0], p[0]);
            fasp_blas_darray_axpy(n, 1.0, p[i], p[0]);
        }

        //-----------------------------------//
        //   compute the convergence rate    //
        //-----------------------------------//
        cr = r_norm / r_norm_old;

    } /* end of iteration while loop */

RESTORE_BESTSOL: // restore the best-so-far solution if necessary
    if ( iter != iter_best ) {

        // compute best residual
        fasp_darray_cp(n,b->val,r);
        fasp_blas_dstr_aAxpy(-1.0,A,x_best,r);

        switch ( StopType ) {
            case STOP_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
            case STOP_REL_PRECRES:
                // z = B(r)
                if ( pc != NULL )
                    pc->fct(r,w,pc->data); /* Apply preconditioner */
                else
                    fasp_darray_cp(n,r,w); /* No preconditioner */
                absres_best = sqrt(ABS(fasp_blas_darray_dotprod(n,w,r)));
                break;
            case STOP_MOD_REL_RES:
                absres_best = fasp_blas_darray_norm2(n,r);
                break;
        }

        if ( absres > absres_best + maxdiff || isnan(absres) ) {
            if ( PrtLvl > PRINT_NONE ) ITS_RESTORE(iter_best);
            fasp_darray_cp(n,x_best,x->val);
            relres = absres_best / normr0;
        }
    }

FINISHED:
    if ( PrtLvl > PRINT_NONE ) ITS_FINAL(iter,MaxIt,relres);

    /*-------------------------------------------
* Free some stuff
*------------------------------------------*/
    fasp_mem_free(work);  work  = NULL;
    fasp_mem_free(p);     p     = NULL;
    fasp_mem_free(hh);    hh    = NULL;
    fasp_mem_free(norms); norms = NULL;

```

```
#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (iter>=MaxIt)
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 ItrSmootherCSRcr.c

// 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_0	Starting index
i_n	Ending index
A	Pointer to dCSRmat: the coefficient matrix (index starts from 0)
vertices	Pointer to CF, 0: Fpt (current level) or 1: Cpt
param	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_functs.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 *);
```

```

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_amg_coarsening_cr (const INT   i_0,
                            const INT   i_n,
                            dCSRmat    *A,
                            ivector    *vertices,
                            AMG_param  *param)
{
    const SHORT prtlvl = param->print_level;

    // local variables
    INT   cand=0,cpt=-1,fpt=1;        // internal labeling
    INT   nc,ns=1;                    // # cpts, # stages
    INT   i,j,in1,nu=3,num1 = nu-1;   // nu is number of cr sweeps
    INT  *cf=NULL,*ia=NULL,*ja=NULL;

    REAL  temp0=0.0e0,temp1=0.0e0,rho=0.0e0,tg=8.0e-01;
    REAL *a=NULL;

    /* WORKING MEMORY -- b not needed, remove later */
    REAL *b=NULL,*u=NULL,*ma=NULL;

    ia = A->IA;
    ja = A->JA;
    a  = A->val;

    if (i_0 == 0) {
        in1 = i_n+1;
    } else {
        in1 = i_n;
    }

    /* CF, RHS, INITIAL GUESS, and MEAS. ARRAY */
    cf = (INT*)fasp_mem_calloc(in1,sizeof(INT));
    b  = (REAL*)fasp_mem_calloc(in1,sizeof(REAL));
    u  = (REAL*)fasp_mem_calloc(in1,sizeof(REAL));
    ma = (REAL*)fasp_mem_calloc(in1,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(i_n>OPENMP_HOLDS)
#endif
    for(i=i_0;i<=i_n;++i) {
        b[i] = 0.0e0; // ZERO RHS
        cf[i] = fpt;  // ALL FPTS
    }

    while (TRUE) {

        nc = 0;
#ifdef _OPENMP
#pragma omp parallel for if(i_n>OPENMP_HOLDS)
#endif
        for(i=i_0; i<=i_n; ++i) {
            if (cf[i] == cpt) {
                nc += 1;
                u[i] = 0.0e0;
            } else {
                u[i] = 1.0e0;
            }
        }

        for (i=i_0;i<=nu;++i) {

            if (i == num1)
                for (j = i_0; j<= i_n; ++j) {
                    if (cf[j] == fpt) {
                        temp0 += u[j]*u[j];
                    }
                }
            fasp_smoother_dcsr_gscr(fpt,i_n,u,ia,ja,a,b,1,cf);
        }

#ifdef _OPENMP
#pragma omp parallel for reduction(+:temp1) if(i_n>OPENMP_HOLDS)
#endif
        for (i = i_0; i<= i_n; ++i) {
            if (cf[i] == fpt) {
                temp1 += u[i]*u[i];
            }
        }
        rho = sqrt(temp1)/sqrt(temp0);

        if ( prtlvl > PRINT_MIN ) printf("rho=%2.13lf\n",rho);

        if ( rho > tg ) {
            /* FORM CAND. SET & COMPUTE IND SET */
            temp0 = 0.0e0;

            for (i = i_0; i<= i_n; ++i) {
                temp1 = fabs(u[i]);
                if (cf[i] == cpt && temp1 > 0.0e0) {
                    temp0 = temp1; // max.
                }
            }
            if (ns == 1) {
                temp1 = pow(0.3, nu);
            } else {
                temp1 = 0.5;
            }

#ifdef _OPENMP
#pragma omp parallel for if(i_n>OPENMP_HOLDS)
#endif
            for (i = i_0; i <= i_n; ++i) {
                if (cf[i] == fpt && fabs(u[i])/temp0 > temp1 && ia[i+1]-ia[i] > 1)
                    cf[i] = cand;
            }
            temp1 = 0.0e0;
            indset(cand,cpt,fpt,ia,ja,i_n,cf,ma);
            ns++;
        }
        else {
            /* back to fasp labeling */

#ifdef _OPENMP
#pragma omp parallel for if(i_n>OPENMP_HOLDS)
#endif
            for (i = i_0; i<= i_n; ++i) {
                if (cf[i] == cpt) {
                    cf[i] = 1; // cpt
                } else {
                    cf[i] = 0; // fpt
                }
                // printf("cf[%i] = %i\n",i,cf[i]);
            }
            vertices->row=i_n;
            if ( prtlvl >= PRINT_MORE ) printf("vertices = %i\n",vertices->row);
            vertices->val= cf;
            if ( prtlvl >= PRINT_MORE ) printf("nc=%i\n",nc);
            break;
        }
    }

    fasp_mem_free(u);   u  = NULL;
    fasp_mem_free(b);   b  = NULL;
    fasp_mem_free(ma);  ma = NULL;

    return nc;
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static INT GraphAdd (Link  *list,
                     INT   *head,
                     INT   *tail,
                     INT    index,
                     INT    istack)
{
    INT prev = tail[-istack];

    list[index].prev = prev;
    if (prev < 0)
        head[-istack] = index;
    else
        list[prev].next = index;
    list[index].next = -istack;
    tail[-istack] = index;

    return 0;
}

static INT GraphRemove (Link   *list,
                        INT    *head,
                        INT    *tail,
                        INT     index)
{
    INT prev = list[index].prev;
    INT next = list[index].next;

    if (prev < 0)
        head[prev] = next;
    else
        list[prev].next = next;
    if (next < 0)
        tail[next] = prev;
    else
        list[next].prev = prev;

    return 0;
}

static INT indset (INT   cand,
                   INT   cpt,
                   INT   fpt,
                   INT  *ia,
                   INT  *ja,
                   INT   n,
                   INT  *cf,
                   REAL *ma)
{
    /* ma:  candidates >= 1, cpts = -1, otherwise = 0
* Note:  graph contains candidates only */

    Link *list;
    INT  *head, *head_mem;
    INT  *tail, *tail_mem;

    INT i, ji, jj, jl, index, istack, stack_size;

    for (istack = i = 0; i < n; ++i) {

        if (cf[i] == cand) {
            ma[i] = 1;
            for (ji = ia[i]+1; ji < ia[i+1]; ++ji) {
                jj = ja[ji];
                if (cf[jj] != cpt) {
                    ma[i]++;
                }
            }

            if (ma[i] > istack) {
                istack = (INT) ma[i];
            }
        }
        else if (cf[i] == cpt) {
            ma[i] = -1;
        }
        else {
            ma[i] = 0;
        }
    }

    stack_size = 2*istack;

    /* INITIALIZE GRAPH */
    list = (Link*)fasp_mem_calloc(n,sizeof(Link));
    head_mem = (INT*)fasp_mem_calloc(stack_size,sizeof(INT));
    tail_mem = (INT*)fasp_mem_calloc(stack_size,sizeof(INT));
    head = head_mem + stack_size;
    tail = tail_mem + stack_size;

#ifdef _OPENMP
#pragma omp parallel for if(stack_size>OPENMP_HOLDS)
#endif
    for (i = -1; i >= -stack_size; i--) {
        head[i] = i;
        tail[i] = i;
    }
```

```

#ifdef _OPENMP
#pragma omp parallel for if(stack_size>OPENMP_HOLDS)
#endif
    for (i = 0; i < n; ++i) {
        if (ma[i] > 0) {
            GraphAdd(list, head, tail, i, (INT) ma[i]);
        }
    }

    while (istack > 0) {
        /* i with maximal measure is at the head of the stacks */
        i = head[-istack];
        /* make i a c-point */
        cf[i] = cpt;
        ma[i] = -1;
        /* remove i from graph */
        GraphRemove(list, head, tail, i);

        /* update neighbors and neighbors-of-neighbors */
        for (ji = ia[i]+1; ji < ia[i+1]; ++ji) {
            jj = ja[ji];
            /* if not "decided" c or f */
            if (ma[jj] > -1) {
                /* if a candidate, remove jj from graph */
                if (ma[jj] > 0) {
                    GraphRemove(list, head, tail, jj);
                }
                /* make jj an f-point and mark "decided" */
                cf[jj] = fpt;
                ma[jj] = -1;

                for (jl = ia[jj]+1; jl < ia[jj+1]; jl++) {
                    index = ja[jl];
                    /* if a candidate, increase likehood of being chosen */
                    if (ma[index] > 0) {
                        ma[index]++;
                        /* move index in graph */
                        GraphRemove(list, head, tail, index);
                        GraphAdd(list, head, tail, index, (INT) ma[index]);
                        if (ma[index] > istack) {
                            istack = (INT) ma[index];
                        }
                    }
                }
            }
        }

        /* reset istack to point to the biggest non-empty stack */
        for ( ; istack > 0; istack-- ) {
            /* if non-negative, break */
            if (head[-istack] > -1) {
                break;
            }
        }
    }

    fasp_mem_free(list);     list     = NULL;
    fasp_mem_free(head_mem); head_mem = NULL;
    fasp_mem_free(tail_mem); tail_mem = NULL;

    return 0;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.137 PreAMGCoarsenRS.c File Reference

Coarsening with a modified Ruge-Stuben strategy.

```
#include "fasp.h"
#include "fasp_functs.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.

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

A	Pointer to dCSRmat: Coefficient matrix (index starts from 0)
vertices	Indicator vector for the C/F splitting of the variables
P	Interpolation matrix (nonzero pattern only)
S	Strong connection matrix
param	Pointer to AMG_param: AMG parameters

Returns

FASP_SUCCESS if successed; 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
```

```
#if DEBUG_MODE > 1
    printf("### DEBUG: Step 2.  C/F splitting ......\n");
#endif

    switch ( coarse_type ) {

        case COARSE_RSP:  // Classical coarsening with positive connections
            col = cfsplitting_clsp(A, S, vertices); break;

        case COARSE_AC:  // Aggressive coarsening
            col = cfsplitting_agg(A, S, vertices, agg_path); break;

        case COARSE_CR:  // Compatible relaxation
            col = fasp_amg_coarsening_cr(0, row-1, A, vertices, param); break;

        case COARSE_MIS:  // Maximal independent set
        {
            ivector order = fasp_ivec_create(row);
            compress_S(S);
            ordering1(S, &order);
            col = cfsplitting_mis(S, vertices, &order);
            fasp_ivec_free(&order);
            break;
        }

        default:  // Classical coarsening
            col = cfsplitting_cls(A, S, vertices);

    }

#if DEBUG_MODE > 1
    printf("### DEBUG: col = %d\n", col);
#endif
    if ( col <= 0 ) return ERROR_UNKNOWN;

#if DEBUG_MODE > 1
    printf("### DEBUG: Step 3.  Find support of C points ......\n");
#endif

    switch ( interp_type ) {

        case INTERP_DIR:  // Direct interpolation or ...
        case INTERP_ENG:  // Energy-min interpolation
            col = clean_ff_couplings(S, vertices, row, col);
            form_P_pattern_dir(P, S, vertices, row, col);
            break;

        case INTERP_STD:  // Standard interpolation
        case INTERP_EXT:  // Extended interpolation
            form_P_pattern_std(P, S, vertices, row, col); break;

        default:
            fasp_chkerr(ERROR_AMG_INTERP_TYPE, __FUNCTION__);

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return FASP_SUCCESS;
}


/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void strong_couplings (dCSRmat    *A,
                              iCSRmat    *S,
                              AMG_param *param )
{
    const SHORT coarse_type = param->coarsening_type;
    const REAL  max_row_sum = param->max_row_sum;
    const REAL  epsilon_str = param->strong_threshold;
    const INT   row = A->row, col = A->col, row1 = row+1;
    const INT   nnz = A->nnz;

    INT  *ia = A->IA, *ja = A->JA;
    REAL *aj = A->val;

    // local variables
    INT   i, j, begin_row, end_row;
    REAL  row_scl, row_sum;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( row > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // get the diagonal entry of A: assume all connections are strong
    dvector diag; fasp_dcsr_getdiag(0, A, &diag);

    // copy the structure of A to S
    S->row = row; S->col = col; S->nnz = nnz; S->val = NULL;
    S->IA = (INT *)fasp_mem_calloc(row1, sizeof(INT));
    S->JA = (INT *)fasp_mem_calloc(nnz,  sizeof(INT));
    fasp_iarray_cp(row1, ia, S->IA);
    fasp_iarray_cp(nnz,  ja, S->JA);

    if ( use_openmp ) {

        // This part is still old!  Need to be updated.  --Chensong 09/18/2016

        INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin,myend,i,row_scl,row_sum,begin_row,end_row,j)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; i++) {

                // Compute most negative entry in each row and row sum
                row_scl = row_sum = 0.0;
                begin_row = ia[i]; end_row = ia[i+1];
                for ( j = begin_row; j < end_row; j++ ) {
                    row_scl  = MIN(row_scl, aj[j]);
                    row_sum += aj[j];
                }

                // Find diagonal entries of S and remove them later
                for ( j = begin_row; j < end_row; j++ ) {
                    if ( ja[j] == i ) { S->JA[j] = -1; break; }
                }

                // Mark entire row as weak couplings if strongly diagonal-dominant
                if ( ABS(row_sum) > max_row_sum * ABS(diag.val[i]) ) {
                    for ( j = begin_row; j < end_row; j++ ) S->JA[j] = -1;
                }
                else {
                    for ( j = begin_row; j < end_row; j++) {
                        // If a_{ij} >= \epsilon_{str} * \min a_{ij}, the connection
                        // j->i is set to be weak; positive entries result in weak
                        // connections
                        if ( A->val[j] >= epsilon_str*row_scl ) S->JA[j] = -1;
                    }
                }

            } // end for i
        } // end for myid

    }

    else {

        for ( i = 0; i < row; ++i ) {

            // Compute row scale and row sum
            row_scl = row_sum = 0.0;
            begin_row = ia[i]; end_row = ia[i+1];

            for ( j = begin_row; j < end_row; j++ ) {

                // Originally:  Not consider positive entries
                // row_sum += aj[j];
                // Now changed to --Chensong 05/17/2013
                row_sum += ABS(aj[j]);

                // Originally:  Not consider positive entries
                // row_scl = MAX(row_scl, -aj[j]); // smallest negative
                // Now changed to --Chensong 06/01/2013
                if ( ja[j] != i ) row_scl = MAX(row_scl, ABS(aj[j])); // largest abs

            }

            // Multiply by the strength threshold
            row_scl *= epsilon_str;

            // Find diagonal entries of S and remove them later
            for ( j = begin_row; j < end_row; j++ ) {
                if ( ja[j] == i ) { S->JA[j] = -1; break; }
            }

            // Mark entire row as weak couplings if strongly diagonal-dominant
            // Originally:  Not consider positive entries
            // if ( ABS(row_sum) > max_row_sum * ABS(diag.val[i]) ) {
            // Now changed to --Chensong 05/17/2013
            if ( row_sum < (2 - max_row_sum) * ABS(diag.val[i]) ) {

                for ( j = begin_row; j < end_row; j++ ) S->JA[j] = -1;

            }
            else {

                switch ( coarse_type ) {

                    case COARSE_RSP:  // consider positive off-diag as well
                        for ( j = begin_row; j < end_row; j++ ) {
                            if ( ABS(A->val[j]) <= row_scl ) S->JA[j] = -1;
                        }
                        break;

                    default:  // only consider n-couplings
                        for ( j = begin_row; j < end_row; j++ ) {
                            if ( -A->val[j] <= row_scl ) S->JA[j] = -1;
                        }
                        break;

                }
            }
        } // end for i

    } // end if openmp

    fasp_dvec_free(&diag);
}

static INT compress_S (iCSRmat *S)
{
    const INT   row = S->row;
    INT       * ia  = S->IA;

    // local variables
    INT         index, i, j, begin_row, end_row;

    // compress S: remove weak connections and form strong coupling matrix
    for ( index = i = 0; i < row; ++i ) {

        begin_row = ia[i]; end_row = ia[i+1];

        ia[i] = index;
        for ( j = begin_row; j < end_row; j++ ) {
            if ( S->JA[j] > -1 ) S->JA[index++] = S->JA[j]; // strong couplings
        }

    }

    S->nnz = S->IA[row] = index;

    if ( S->nnz <= 0 ) {
        return ERROR_UNKNOWN;
    }
    else {
        return FASP_SUCCESS;
    }
}

static void rem_positive_ff (dCSRmat   *A,
                             iCSRmat   *Stemp,
                              ivector    *vertices)
{
    const INT   row = A->row;
    INT        *ia  = A->IA, *vec = vertices->val;

    REAL        row_scl, max_entry;
    INT         i, j, ji, max_index;

    for ( i = 0; i < row; ++i ) {

        if ( vec[i] != FGPT ) continue; // skip non F-variables

        row_scl = 0.0;
        for ( ji = ia[i]; ji < ia[i+1]; ++ji ) {
            j = A->JA[ji];
            if ( j == i ) continue; // skip diagonal
            row_scl = MAX(row_scl, ABS(A->val[ji])); // max abs entry
        } // end for ji
        row_scl *= 0.75;

        // looking for strong F-F connections
        max_index = -1; max_entry = 0.0;
        for ( ji = ia[i]; ji < ia[i+1]; ++ji ) {
            j = A->JA[ji];
            if ( j == i ) continue; // skip diagonal
            if ( vec[j] != FGPT ) continue; // skip F-C connections
            if ( A->val[ji] > row_scl ) {
                Stemp->JA[ji] = j;
                if ( A->val[ji] > max_entry ) {
                    max_entry = A->val[ji];
                    max_index = j; // max positive entry
                }
            }
        } // end for ji

        // mark max positive entry as C-point
        if ( max_index != -1 ) vec[max_index] = CGPT;

    } // end for i

}

static INT cfsplitting_cls (dCSRmat    *A,
                            iCSRmat    *S,
                            ivector    *vertices)
{
    const INT   row = A->row;

    // local variables
    INT col = 0;
    INT maxmeas, maxnode, num_left = 0;
    INT measure, newmeas;
    INT i, j, k, l;
    INT myid, mybegin, myend;
    INT *vec = vertices->val;
    INT *work = (INT*)fasp_mem_calloc(3*row,sizeof(INT));
    INT *lists = work, *where = lists+row, *lambda = where+row;

#if RS_C1
    INT set_empty = 1;
    INT jkeep = 0, cnt, index;
    INT row_end_S, ji, row_end_S_nabor, jj;
    INT *graph_array = lambda;
#else
    INT *ia = A->IA;
#endif

    LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;

    SHORT nthreads = 1, use_openmp = FALSE;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

#ifdef _OPENMP
    if ( row > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif
```

```

    // 0.  Compress S and form S_transpose
    col = compress_S(S);
    if ( col < 0 ) goto FINISHED; // compression failed!!!

    iCSRmat ST; fasp_icsr_trans(S, &ST);

    // 1.  Initialize lambda
    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin, myend, i)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; i++ ) lambda[i] = ST.IA[i+1] - ST.IA[i];
        }
    }
    else {
        for ( i = 0; i < row; ++i ) lambda[i] = ST.IA[i+1] - ST.IA[i];
    }

    // 2.  Before C/F splitting algorithm starts, filter out the variables which
    //     have no connections at all and mark them as special F-variables.
    if ( use_openmp ) {

#ifdef _OPENMP
#pragma omp parallel for reduction(+:num_left) private(myid, mybegin, myend, i)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; i++ ) {
#if RS_C1 // Check C1 criteria or not
                if ( S->IA[i+1] == S->IA[i] )
#else
                    if ( (ia[i+1]-ia[i]) <= 1 )
#endif
                    {
                        vec[i] = ISPT; // set i as an ISOLATED fine node
                        lambda[i] = 0;
                    }
                    else {
                        vec[i] = UNPT; // set i as a undecided node
                        num_left++;
                    }
            }
        } // end for myid

    }

    else {

        for ( i = 0; i < row; ++i ) {

#if RS_C1
            if ( S->IA[i+1] == S->IA[i] )
#else
                if ( (ia[i+1]-ia[i]) <= 1 )
#endif
                {
                    vec[i] = ISPT; // set i as an ISOLATED fine node
                    lambda[i] = 0;
                }
                else {
                    vec[i] = UNPT; // set i as a undecided node
                    num_left++;
                }
        } // end for i

    }

    // 3.  Form linked list for lambda (max to min)
    for ( i = 0; i < row; ++i ) {

        if ( vec[i] == ISPT ) continue; // skip isolated variables

        measure = lambda[i];

        if ( measure > 0 ) {
            enter_list(&LoL_head, &LoL_tail, lambda[i], i, lists, where);
        }
        else {

            if ( measure < 0 ) printf("### WARNING: Negative lambda[%d]!\n", i);

            // Set variables with non-positive measure as F-variables
            vec[i] = FGPT; // no strong connections, set i as fine node
            --num_left;

            // Update lambda and linked list after i->F
            for ( k = S->IA[i]; k < S->IA[i+1]; ++k ) {
                j = S->JA[k];
                if ( vec[j] == ISPT ) continue; // skip isolate variables
                if ( j < i ) {
                    newmeas = lambda[j];
                    if ( newmeas > 0 ) {
                        remove_node(&LoL_head, &LoL_tail, newmeas, j, lists, where);
                    }
                    newmeas = ++(lambda[j]);
                    enter_list(&LoL_head, &LoL_tail, newmeas, j, lists, where);
                }
                else {
                    newmeas = ++(lambda[j]);
                }
            }

        } // end if measure

    } // end for i

    // 4.  Main loop
    while ( num_left > 0 ) {

        // pick $i\in U$ with $\max\lambda_i:  C:=C\cup\{i\}, U:=U\\{i\}$
        maxnode = LoL_head->head;
        maxmeas = lambda[maxnode];
        if ( maxmeas == 0 )
            printf("### WARNING: Head of the list has measure 0!\n");

        vec[maxnode] = CGPT; // set maxnode as coarse node
        lambda[maxnode] = 0;
        --num_left;
        remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
        col++;

        // for all $j\in S_i^T\cap U: F:=F\cup\{j\}, U:=U\backslash\{j\}$
        for ( i = ST.IA[maxnode]; i < ST.IA[maxnode+1]; ++i ) {

            j = ST.JA[i];

            if ( vec[j] != UNPT ) continue; // skip decided variables

            vec[j] = FGPT;  // set j as fine node
            remove_node(&LoL_head, &LoL_tail, lambda[j], j, lists, where);
            --num_left;

            // Update lambda and linked list after j->F
            for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
                k = S->JA[l];
                if ( vec[k] == UNPT ) { // k is unknown
                    remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
                    newmeas = ++(lambda[k]);
                    enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
                }
            }

        } // end for i

        // Update lambda and linked list after maxnode->C
        for ( i = S->IA[maxnode]; i < S->IA[maxnode+1]; ++i ) {

            j = S->JA[i];

            if ( vec[j] != UNPT ) continue; // skip decided variables

            measure = lambda[j];
            remove_node(&LoL_head, &LoL_tail, measure, j, lists, where);
            lambda[j] = --measure;

            if ( measure > 0 ) {
                enter_list(&LoL_head, &LoL_tail, measure, j, lists, where);
            }
            else { // j is the only point left, set as fine variable
```

```
                vec[j] = FGPT;
                --num_left;

                // Update lambda and linked list after j->F
                for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
                    k = S->JA[l];
                    if ( vec[k] == UNPT ) { // k is unknown
                        remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
                        newmeas = ++(lambda[k]);
                        enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
                    }
                } // end for l
            } // end if

        } // end for

    } // end while

#if RS_C1

    // C/F splitting of RS coarsening check C1 Criterion
    fasp_iarray_set(row, graph_array, -1);
    for (i = 0; i < row; i ++)
    {
        if (vec[i] == FGPT)
        {
            row_end_S = S->IA[i+1];
            for (ji = S->IA[i]; ji < row_end_S; ji ++)
            {
                j = S->JA[ji];
                if (vec[j] == CGPT)
                {
                    graph_array[j] = i;
                }
            }
            cnt = 0;
            for (ji = S->IA[i]; ji < row_end_S; ji ++)
            {
                j = S->JA[ji];
                if (vec[j] == FGPT)
                {
                    set_empty = 1;
                    row_end_S_nabor = S->IA[j+1];
                    for (jj = S->IA[j]; jj < row_end_S_nabor; jj ++)
                    {
                        index = S->JA[jj];
                        if (graph_array[index] == i)
                        {
                            set_empty = 0;
                            break;
                        }
                    }
                    if (set_empty)
                    {
                        if (cnt == 0)
                        {
                            vec[j] = CGPT;
                            col++;
                            graph_array[j] = i;
                            jkeep = j;
                            cnt = 1;
                        }
                        else
                        {
                            vec[i] = CGPT;
                            vec[jkeep] = FGPT;
                            break;
                        }
                    }
                }
            }
        }
    }

#endif

    fasp_icsr_free(&ST);

    if ( LoL_head ) {
        list_ptr = LoL_head;
        LoL_head->prev_node = NULL;
        LoL_head->next_node = NULL;
        LoL_head = list_ptr->next_node;
        fasp_mem_free(list_ptr); list_ptr = NULL;
    }

FINISHED:
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return col;
}

static INT cfsplitting_clsp (dCSRmat   *A,
                             iCSRmat   *S,
                             ivector   *vertices)
{
    const INT   row = A->row;

    // local variables
    INT col = 0;
    INT maxmeas, maxnode, num_left = 0;
    INT measure, newmeas;
    INT i, j, k, l;
    INT myid, mybegin, myend;

    INT *ia = A->IA, *vec = vertices->val;
    INT *work = (INT*)fasp_mem_calloc(3*row,sizeof(INT));
    INT *lists = work, *where = lists+row, *lambda = where+row;

    LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;

    SHORT nthreads = 1, use_openmp = FALSE;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

#ifdef _OPENMP
    if ( row > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // 0.  Compress S and form S_transpose (not complete, just IA and JA)
    iCSRmat Stemp;
    Stemp.row = S->row; Stemp.col = S->col; Stemp.nnz = S->nnz;
    Stemp.IA = (INT *)fasp_mem_calloc(S->row+1, sizeof(INT));
    fasp_iarray_cp (S->row+1, S->IA, Stemp.IA);
    Stemp.JA = (INT *)fasp_mem_calloc(S->nnz, sizeof(INT));
    fasp_iarray_cp (S->nnz, S->JA, Stemp.JA);

    if ( compress_S(S) < 0 ) goto FINISHED; // compression failed!!!

    iCSRmat ST; fasp_icsr_trans(S, &ST);

    // 1.  Initialize lambda
    if ( use_openmp ) {
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin,myend,i)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; i++ ) lambda[i] = ST.IA[i+1] - ST.IA[i];
        }
    }
    else {
        for ( i = 0; i < row; ++i ) lambda[i] = ST.IA[i+1] - ST.IA[i];
    }

    // 2.  Before C/F splitting algorithm starts, filter out the variables which
    //     have no connections at all and mark them as special F-variables.
    if ( use_openmp ) {

#ifdef _OPENMP
#pragma omp parallel for reduction(+:num_left) private(myid, mybegin, myend, i)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; i++ ) {
                if ( (ia[i+1]-ia[i]) <= 1 ) {
                    vec[i] = ISPT; // set i as an ISOLATED fine node
                    lambda[i] = 0;
                }
                else {
                    vec[i] = UNPT; // set i as a undecided node
                    num_left++;
                }
            }
        } // end for myid

    }
    else {

        for ( i = 0; i < row; ++i ) {
            if ( (ia[i+1]-ia[i]) <= 1 ) {
                vec[i] = ISPT; // set i as an ISOLATED fine node
                lambda[i] = 0;
            }
            else {
                vec[i] = UNPT; // set i as a undecided node
                num_left++;
            }
        } // end for i

    }

    // 3.  Form linked list for lambda (max to min)
    for ( i = 0; i < row; ++i ) {

        if ( vec[i] == ISPT ) continue; // skip isolated variables

        measure = lambda[i];

        if ( measure > 0 ) {
            enter_list(&LoL_head, &LoL_tail, lambda[i], i, lists, where);
        }
        else {

            if ( measure < 0 ) printf("### WARNING: Negative lambda[%d]!\n", i);

            // Set variables with non-positive measure as F-variables
            vec[i] = FGPT; // no strong connections, set i as fine node
            --num_left;

            // Update lambda and linked list after i->F
            for ( k = S->IA[i]; k < S->IA[i+1]; ++k ) {

                j = S->JA[k];
                if ( vec[j] == ISPT ) continue; // skip isolate variables

                if ( j < i ) { // only look at the previous points!!
                    newmeas = lambda[j];
                    if ( newmeas > 0 ) {
                        remove_node(&LoL_head, &LoL_tail, newmeas, j, lists, where);
                    }
                    newmeas = ++(lambda[j]);
                    enter_list(&LoL_head, &LoL_tail, newmeas, j, lists, where);
                }
                else { // will be checked later on
                    newmeas = ++(lambda[j]);
                } // end if

            } // end for k

        } // end if measure

    } // end for i

    // 4.  Main loop
    while ( num_left > 0 ) {

        // pick $i\in U$ with $\max\lambda_i:  C:=C\cup\{i\}, U:=U\\{i\}$
        maxnode = LoL_head->head;
        maxmeas = lambda[maxnode];
        if ( maxmeas == 0 )
            printf("### WARNING: Head of the list has measure 0!\n");

        vec[maxnode] = CGPT; // set maxnode as coarse node
        lambda[maxnode] = 0;
        --num_left;
        remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
        col++;

        // for all $j\in S_i^T\cap U: F:=F\cup\{j\}, U:=U\backslash\{j\}$
        for ( i = ST.IA[maxnode]; i < ST.IA[maxnode+1]; ++i ) {

            j = ST.JA[i];

            if ( vec[j] != UNPT ) continue; // skip decided variables

            vec[j] = FGPT;  // set j as fine node
            remove_node(&LoL_head, &LoL_tail, lambda[j], j, lists, where);
            --num_left;

            // Update lambda and linked list after j->F
            for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
                k = S->JA[l];
                if ( vec[k] == UNPT ) { // k is unknown
                    remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
                    newmeas = ++(lambda[k]);
                    enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
                }
            }

        } // end for i

        // Update lambda and linked list after maxnode->C
        for ( i = S->IA[maxnode]; i < S->IA[maxnode+1]; ++i ) {

            j = S->JA[i];

            if ( vec[j] != UNPT ) continue; // skip decided variables

            measure = lambda[j];
            remove_node(&LoL_head, &LoL_tail, measure, j, lists, where);
            lambda[j] = --measure;

            if ( measure > 0 ) {
                enter_list(&LoL_head, &LoL_tail, measure, j, lists, where);
            }
            else { // j is the only point left, set as fine variable
                vec[j] = FGPT;
                --num_left;

                // Update lambda and linked list after j->F
                for ( l = S->IA[j]; l < S->IA[j+1]; l++ ) {
                    k = S->JA[l];
                    if ( vec[k] == UNPT ) { // k is unknown
                        remove_node(&LoL_head, &LoL_tail, lambda[k], k, lists, where);
                        newmeas = ++(lambda[k]);
                        enter_list(&LoL_head, &LoL_tail, newmeas, k, lists, where);
                    }
                } // end for l
            } // end if

        } // end for

    } // end while

    fasp_icsr_free(&ST);

    if ( LoL_head ) {
        list_ptr = LoL_head;
        LoL_head->prev_node = NULL;
        LoL_head->next_node = NULL;
        LoL_head = list_ptr->next_node;
        fasp_mem_free(list_ptr); list_ptr = NULL;
    }

    // Enforce F-C connections.  Adding this step helps for the ExxonMobil test
    // problems!  Need more tests though --Chensong 06/08/2013
    // col = clean_ff_couplings(S, vertices, row, col);

    rem_positive_ff(A, &Stemp, vertices);

    if ( compress_S(&Stemp) < 0 ) goto FINISHED; // compression failed!!!

    S->row = Stemp.row;
    S->col = Stemp.col;
    S->nnz = Stemp.nnz;

    fasp_mem_free(S->IA); S->IA = Stemp.IA;
    fasp_mem_free(S->JA); S->JA = Stemp.JA;

FINISHED:
    fasp_mem_free(work);  work  = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return col;
}

static void strong_couplings_agg1 (dCSRmat   *A,
                                   iCSRmat   *S,
                                   iCSRmat   *Sh,
                                   ivector   *vertices,
                                   ivector   *CGPT_index,
                                   ivector   *CGPT_rindex)
{
    const INT row = A->row;

    // local variables
    INT       i, j, k;
    INT       num_c, count, ci, cj, ck, fj, cck;
    INT      *cp_index, *cp_rindex, *visited;
    INT      *vec = vertices->val;

    // count the number of coarse grid points
    for ( num_c = i = 0; i < row; i++ ) {
        if ( vec[i] == CGPT ) num_c++;
    }

    // for the reverse indexing of coarse grid points
    fasp_ivec_alloc(row, CGPT_rindex);
    cp_rindex = CGPT_rindex->val;

    // generate coarse grid point index
    fasp_ivec_alloc(num_c, CGPT_index);
    cp_index = CGPT_index->val;
    for ( j = i = 0; i < row; i++ ) {
        if ( vec[i] == CGPT ) {
            cp_index[j]  = i;
            cp_rindex[i] = j;
            j++;
        }
    }

    // allocate space for Sh
    Sh->row = Sh->col = num_c;
    Sh->val = Sh->JA = NULL;
    Sh->IA  = (INT*)fasp_mem_calloc(Sh->row+1, sizeof(INT));

    // record the number of times some coarse point is visited
    visited = (INT*)fasp_mem_calloc(num_c, sizeof(INT));
    fasp_iarray_set(num_c, visited, -1);

/**************************************************/
    /* step 1:  Find first the structure IA of Sh  */
/**************************************************/

    Sh->IA[0] = 0;

    for ( ci = 0; ci < Sh->row; ci++ ) {

        i = cp_index[ci]; // find the index of the ci-th coarse grid point

        // number of coarse point that i is strongly connected to w.r.t.  S(p,2)
        count = 0;

        // visit all the fine neighbors that ci is strongly connected to
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

            fj = S->JA[j];

            if ( vec[fj] == CGPT && fj != i ) {
                cj = cp_rindex[fj];
                if ( visited[cj] != ci ) {
                    visited[cj] = ci; // mark as strongly connected from ci
                    count++;
                }

            }

            else if ( vec[fj] == FGPT ) { // fine grid point,

                // find all the coarse neighbors that fj is strongly connected to
                for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
                    ck = S->JA[k];
                    if ( vec[ck] == CGPT && ck != i ) { // it is a coarse grid point
                        if ( cp_rindex[ck] >= num_c ) {
                            printf("### ERROR: ck=%d, num_c=%d, out of bound!\n",
                                   ck, num_c);
                            fasp_chkerr(ERROR_AMG_COARSEING, __FUNCTION__);
                        }
                        cck = cp_rindex[ck];

                        if ( visited[cck] != ci ) {
                            visited[cck] = ci; // mark as strongly connected from ci
                            count++;
                        }
                    } //end if
                } //end for k

            } //end if

        } //end for j

        Sh->IA[ci+1] = Sh->IA[ci] + count;

    } //end for i

/*************************/
    /* step 2:  Find JA of Sh */
/*************************/

    fasp_iarray_set(num_c, visited, -1); // reset visited

    Sh->nnz = Sh->IA[Sh->row];
    Sh->JA  = (INT*)fasp_mem_calloc(Sh->nnz, sizeof(INT));

    for ( ci = 0; ci < Sh->row; ci++ ) {

        i = cp_index[ci]; // find the index of the i-th coarse grid point
        count = Sh->IA[ci]; // count for coarse points

        // visit all the fine neighbors that ci is strongly connected to
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

            fj = S->JA[j];

            if ( vec[fj] == CGPT && fj != i ) {
                cj = cp_rindex[fj];
                if ( visited[cj] != ci ) { // not visited yet
                    visited[cj] = ci;
                    Sh->JA[count] = cj;
                    count++;
                }
            }
            else if ( vec[fj] == FGPT ) { // fine grid point,
                //find all the coarse neighbors that fj is strongly connected to
                for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {
                    ck = S->JA[k];
                    if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
                        cck = cp_rindex[ck];
                        if ( visited[cck] != ci ) { // not visited yet
                            visited[cck] = ci;
                            Sh->JA[count] = cck;
                            count++;
                        }
                    } // end if
                } // end for k
            } // end if

        } // end for j

        if ( count != Sh->IA[ci+1] ) {
            printf("### WARNING: Inconsistent numbers of nonzeros!\n ");
        }

    } // end for ci

    fasp_mem_free(visited); visited = NULL;
}

static void strong_couplings_agg2 (dCSRmat   *A,
                                   iCSRmat   *S,
                                   iCSRmat   *Sh,
                                   ivector   *vertices,
                                   ivector   *CGPT_index,
                                   ivector   *CGPT_rindex)
{
    const INT row = A->row;

    // local variables
    INT       i, j, k;
    INT       num_c, count, ci, cj, ck, fj, cck;
    INT      *cp_index, *cp_rindex, *visited;
    INT      *vec = vertices->val;

    // count the number of coarse grid points
    for ( num_c = i = 0; i < row; i++ ) {
        if ( vec[i] == CGPT ) num_c++;
    }

    // for the reverse indexing of coarse grid points
    fasp_ivec_alloc(row, CGPT_rindex);
    cp_rindex = CGPT_rindex->val;

    // generate coarse grid point index
    fasp_ivec_alloc(num_c, CGPT_index);
    cp_index = CGPT_index->val;
    for ( j = i = 0; i < row; i++ ) {
        if ( vec[i] == CGPT ) {
            cp_index[j]  = i;
            cp_rindex[i] = j;
            j++;
        }
    }

    // allocate space for Sh
    Sh->row = Sh->col = num_c;
    Sh->val = Sh->JA  = NULL;
    Sh->IA  = (INT*)fasp_mem_calloc(Sh->row+1, sizeof(INT));

    // record the number of times some coarse point is visited
    visited = (INT*)fasp_mem_calloc(num_c, sizeof(INT));
    memset(visited, 0, sizeof(INT)*num_c);

/**********************************************/
    /* step 1:  Find first the structure IA of Sh  */
/**********************************************/

    Sh->IA[0] = 0;

    for ( ci = 0; ci < Sh->row; ci++ ) {

        i = cp_index[ci]; // find the index of the ci-th coarse grid point

        // number of coarse point that i is strongly connected to w.r.t.  S(p,2)
        count = 0;

        // visit all the fine neighbors that ci is strongly connected to
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

            fj = S->JA[j];

            if ( vec[fj] == CGPT && fj != i ) {
                cj = cp_rindex[fj];
                if ( visited[cj] != ci+1 ) { // not visited yet
                    visited[cj] = ci+1; // mark as strongly connected from ci
                    count++;
                }
            }

            else if ( vec[fj] == FGPT ) { // fine grid point

                // find all the coarse neighbors that fj is strongly connected to
                for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {

                    ck = S->JA[k];

                    if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
                        if ( cp_rindex[ck] >= num_c ) {
                            printf("### ERROR: ck=%d, num_c=%d, out of bound!\n",
                                   ck, num_c);
                            fasp_chkerr(ERROR_AMG_COARSEING, __FUNCTION__);
                        }
                        cck = cp_rindex[ck];

                        if ( visited[cck] == ci+1 ) {
                            // visited already!
                        }
                        else if ( visited[cck] == -ci-1 ) {
                            visited[cck] = ci+1; // mark as strongly connected from ci
                            count++;
                        }
                        else {
                            visited[cck] = -ci-1; // mark as visited
                        }

                    } //end if vec[ck]

                } // end for k

            } // end if vec[fj]

        } // end for j

        Sh->IA[ci+1] = Sh->IA[ci] + count;

    } //end for i

/************************/
    /* step 2:  Find JA of Sh */
/************************/

    memset(visited, 0, sizeof(INT)*num_c); // reset visited

    Sh->nnz = Sh->IA[Sh->row];
    Sh->JA  = (INT*)fasp_mem_calloc(Sh->nnz,sizeof(INT));

    for ( ci = 0; ci < Sh->row; ci++ ) {

        i = cp_index[ci]; // find the index of the i-th coarse grid point
        count = Sh->IA[ci]; // count for coarse points

        // visit all the fine neighbors that ci is strongly connected to
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

            fj = S->JA[j];

            if ( vec[fj] == CGPT && fj != i ) {
                cj = cp_rindex[fj];
                if ( visited[cj] != ci+1 ) { // not visited yet
                    visited[cj] = ci+1;
                    Sh->JA[count] = cj;
                    count++;
                }
            }

            else if ( vec[fj] == FGPT ) { // fine grid point

                // find all the coarse neighbors that fj is strongly connected to
                for ( k = S->IA[fj]; k < S->IA[fj+1]; k++ ) {

                    ck = S->JA[k];

                    if ( vec[ck] == CGPT && ck != i ) { // coarse grid point
                        cck = cp_rindex[ck];
                        if ( visited[cck] == ci+1 ) {
                            // visited before
                        }
                        else if ( visited[cck] == -ci-1 ) {
                            visited[cck] = ci+1;
                            Sh->JA[count] = cck;
                            count++;
                        }
                        else {
                            visited[cck] = -ci-1;
                        }
                    } // end if vec[ck]

                } // end for k

            } // end if vec[fj]

        } // end for j

        if ( count != Sh->IA[ci+1] ) {
            printf("### WARNING: Inconsistent numbers of nonzeros!\n ");
        }

    } // end for ci

    fasp_mem_free(visited); visited = NULL;
}

static INT cfsplitting_agg (dCSRmat    *A,
                            iCSRmat    *S,
                            ivector    *vertices,
                            INT         aggressive_path)
{
    const INT    row = A->row;
    INT          col = 0; // initialize col(P): returning output

    // local variables
    INT    *vec = vertices->val, *cp_index;
    INT     maxmeas, maxnode, num_left = 0;
    INT     measure, newmeas;
    INT     i, j, k, l, m, ci, cj, ck, cl, num_c;
    SHORT   IS_CNEIGH;

    INT    *work = (INT*)fasp_mem_calloc(3*row,sizeof(INT));
    INT    *lists = work, *where = lists+row, *lambda = where+row;

    ivector  CGPT_index, CGPT_rindex;
    LinkList LoL_head = NULL, LoL_tail = NULL, list_ptr = NULL;

    // Sh is for the strong coupling matrix between temporary CGPTs
    // ShT is the transpose of Sh
    // Snew is for combining the information from S and Sh
    iCSRmat ST, Sh, ShT;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

/**************************************************************/
    /* Coarsening Phase ONE: find temporary coarse level points */
/**************************************************************/

    num_c = cfsplitting_cls(A, S, vertices);
    fasp_icsr_trans(S, &ST);

/**************************************************************/
    /* Coarsening Phase TWO: find real coarse level points      */
/**************************************************************/

    // find Sh, the strong coupling between coarse grid points S(path,2)
    if ( aggressive_path < 2 )
        strong_couplings_agg1(A, S, &Sh, vertices, &CGPT_index, &CGPT_rindex);
    else
        strong_couplings_agg2(A, S, &Sh, vertices, &CGPT_index, &CGPT_rindex);

    fasp_icsr_trans(&Sh, &ShT);

    CGPT_index.row  = num_c;
    CGPT_rindex.row = row;
    cp_index        = CGPT_index.val;

    // 1.  Initialize lambda
#ifdef _OPENMP
#pragma omp parallel for if(num_c>OPENMP_HOLDS)
#endif
    for ( ci = 0; ci < num_c; ++ci ) lambda[ci] = ShT.IA[ci+1]-ShT.IA[ci];

    // 2.  Form linked list for lambda (max to min)
    for ( ci = 0; ci < num_c; ++ci ) {

        i       = cp_index[ci];
        measure = lambda[ci];

        if ( vec[i] == ISPT ) continue; // skip isolated points

        if ( measure > 0 ) {
            enter_list(&LoL_head, &LoL_tail, lambda[ci], ci, lists, where);
            num_left++;
        }
        else {
            if ( measure < 0) printf("### WARNING: Negative lambda[%d]!\n", i);

            vec[i] = FGPT; // set i as fine node

            // update the lambda value in the CGPT neighbor of i
            for ( ck = Sh.IA[ci]; ck < Sh.IA[ci+1]; ++ck ) {

                cj = Sh.JA[ck];
                j  = cp_index[cj];

                if ( vec[j] == ISPT ) continue;

                if ( cj < ci ) {
                    newmeas = lambda[cj];
                    if ( newmeas > 0 ) {
                        remove_node(&LoL_head, &LoL_tail, newmeas, cj, lists, where);
                        num_left--;
                    }
                    newmeas = ++(lambda[cj]);
                    enter_list(&LoL_head, &LoL_tail,  newmeas, cj, lists, where);
                    num_left++;
                }
                else {
                    newmeas = ++(lambda[cj]);
                } // end if cj<ci

            } // end for ck

        } // end if

    } // end for ci

    // 3.  Main loop
    while ( num_left > 0 ) {

        // pick $i\in U$ with $\max\lambda_i:  C:=C\cup\{i\}, U:=U\\{i\}$
        maxnode = LoL_head->head;
        maxmeas = lambda[maxnode];
        if ( maxmeas == 0 ) printf("### WARNING: Head of the list has measure 0!\n");

        // mark maxnode as real coarse node, labelled as number 3
        vec[cp_index[maxnode]] = 3;
        --num_left;
        remove_node(&LoL_head, &LoL_tail, maxmeas, maxnode, lists, where);
        lambda[maxnode] = 0;
        col++; // count for the real coarse node after aggressive coarsening

        // for all $j\in S_i^T\cap U: F:=F\cup\{j\}, U:=U\backslash\{j\}$
        for ( ci = ShT.IA[maxnode]; ci < ShT.IA[maxnode+1]; ++ci ) {

            cj = ShT.JA[ci];
            j  = cp_index[cj];

            if ( vec[j] != CGPT ) continue; // skip if j is not C-point

            vec[j] = 4; // set j as 4--fake CGPT
            remove_node(&LoL_head, &LoL_tail, lambda[cj], cj, lists, where);
            --num_left;

            // update the measure for neighboring points
            for ( cl = Sh.IA[cj]; cl < Sh.IA[cj+1]; cl++ ) {
                ck = Sh.JA[cl];
                k  = cp_index[ck];
                if ( vec[k] == CGPT ) { // k is temporary CGPT
                    remove_node(&LoL_head, &LoL_tail, lambda[ck], ck, lists, where);
                    newmeas = ++(lambda[ck]);
                    enter_list(&LoL_head, &LoL_tail, newmeas, ck, lists, where);
                }
            }

        } // end for ci

        // Update lambda and linked list after maxnode->C
        for ( ci = Sh.IA[maxnode]; ci < Sh.IA[maxnode+1]; ++ci ) {
```

```c

            cj = Sh.JA[ci];
            j  = cp_index[cj];

            if ( vec[j] != CGPT ) continue; // skip if j is not C-point

            measure = lambda[cj];
            remove_node(&LoL_head, &LoL_tail, measure, cj, lists, where);
            lambda[cj] = --measure;

            if ( measure > 0 ) {
                enter_list(&LoL_head, &LoL_tail,measure, cj, lists, where);
            }
            else {
                vec[j] = 4; // set j as fake CGPT variable
                --num_left;
                for ( cl = Sh.IA[cj]; cl < Sh.IA[cj+1]; cl++ ) {
                    ck = Sh.JA[cl];
                    k  = cp_index[ck];
                    if ( vec[k] == CGPT ) { // k is temporary CGPT
                        remove_node(&LoL_head, &LoL_tail, lambda[ck], ck, lists, where);
                        newmeas = ++(lambda[ck]);
                        enter_list(&LoL_head, &LoL_tail, newmeas, ck, lists, where);
                    }
                } // end for l
            } // end if

        } // end for

    } // while

    // 4.  reorganize the variable type:  mark temporary CGPT--1 and fake CGPT--4 as
    //    FGPT; mark real CGPT--3 to be CGPT
#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
    for ( i = 0; i < row; i++ ) {
        if ( vec[i] == CGPT || vec[i] == 4 ) vec[i] = FGPT;
    }

#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
    for ( i = 0; i < row; i++ ) {
        if ( vec[i] == 3 ) vec[i] = CGPT;
    }

/************************************************************/
    /* Coarsening Phase THREE: all the FGPTs which have no CGPT */
    /* neighbors within distance 2.  Change them into CGPT such  */
    /* that the standard interpolation works!                                       */
/************************************************************/

    for ( i = 0; i < row; i++ ) {

        if ( vec[i] != FGPT ) continue;

        IS_CNEIGH = FALSE; // whether there exist CGPT neighbors within distance of 2

        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

            if ( IS_CNEIGH ) break;

            k = S->JA[j];

            if ( vec[k] == CGPT ) {
                IS_CNEIGH = TRUE;
            }
            else if ( vec[k] == FGPT ) {
                for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) {
                    m = S->JA[l];
                    if ( vec[m] == CGPT ) {
                        IS_CNEIGH = TRUE; break;
                    }
                } // end for l
            }

        } // end for j

        // no CGPT neighbors in distance <= 2, mark i as CGPT
        if ( !IS_CNEIGH ) {
            vec[i] = CGPT; col++;
        }

    } // end for i

    if ( LoL_head ) {
        list_ptr = LoL_head;
        LoL_head->prev_node = NULL;
        LoL_head->next_node = NULL;
        LoL_head = list_ptr->next_node;
        fasp_mem_free(list_ptr); list_ptr = NULL;
    }

    fasp_ivec_free(&CGPT_index);
    fasp_ivec_free(&CGPT_rindex);
    fasp_icsr_free(&Sh);
    fasp_icsr_free(&ST);
    fasp_icsr_free(&ShT);
    fasp_mem_free(work); work = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return col;
}

static INT clean_ff_couplings (iCSRmat   *S,
                               ivector   *vertices,
                               INT        row,
                               INT        col)
{
    // local variables
    INT  *vec         = vertices->val;
    INT  *cindex      = (INT *)fasp_mem_calloc(row, sizeof(INT));
    INT   set_empty   = TRUE, C_i_nonempty  = FALSE;
    INT   ci_tilde    = -1,   ci_tilde_mark = -1;
    INT   ji, jj, i, j, index;

    fasp_iarray_set(row, cindex, -1);

    for ( i = 0; i < row; ++i ) {

        if ( vec[i] != FGPT ) continue; // skip non F-variables

        for ( ji = S->IA[i]; ji < S->IA[i+1]; ++ji ) {
            j = S->JA[ji];
            if ( vec[j] == CGPT ) cindex[j] = i; // mark C-neighbors
            else cindex[j] = -1; // reset cindex --Chensong 06/02/2013
        }

        if ( ci_tilde_mark != i ) ci_tilde = -1;//???

        for ( ji = S->IA[i]; ji < S->IA[i+1]; ++ji ) {

            j = S->JA[ji];

            if ( vec[j] != FGPT ) continue; // skip non F-variables

            // check whether there is a C-connection
            set_empty = TRUE;
            for ( jj = S->IA[j]; jj < S->IA[j+1]; ++jj ) {
                index = S->JA[jj];
                if ( cindex[index] == i ) {
                    set_empty = FALSE; break;
                }
            } // end for jj

            // change the point i (if only F-F exists) to C
            if ( set_empty ) {
                if ( C_i_nonempty ) {
                    vec[i] = CGPT; col++;
                    if ( ci_tilde > -1 ) {
                        vec[ci_tilde] = FGPT; col--;
                        ci_tilde = -1;
                    }
                    C_i_nonempty = FALSE;
                    break;
                }
                else { // temporary set j->C and roll back
                    vec[j] = CGPT; col++;
                    ci_tilde = j;
                    ci_tilde_mark = i;
                    C_i_nonempty = TRUE;
                    i--; // roll back to check i-point again
                    break;
                } // end if C_i_nonempty
            } // end if set_empty

        } // end for ji

    } // end for i

    fasp_mem_free(cindex); cindex = NULL;

    return col;
}

static void form_P_pattern_dir (dCSRmat   *P,
                                iCSRmat   *S,
                                ivector   *vertices,
                                INT        row,
                                INT        col)
{
    // local variables
    INT i, j, k, index;
    INT *vec = vertices->val;

    SHORT nthreads = 1, use_openmp = FALSE;

#ifdef _OPENMP
    if ( row > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // Initialize P matrix
    P->row = row; P->col = col;
    P->IA  = (INT*)fasp_mem_calloc(row+1, sizeof(INT));

    // step 1:  Find the structure IA of P first:  using P as a counter
    if ( use_openmp ) {

        INT mybegin, myend, myid;
#ifdef _OPENMP
#pragma omp parallel for private(myid, mybegin,myend,i,j,k)
#endif
        for ( myid = 0; myid < nthreads; myid++ ) {
            fasp_get_start_end(myid, nthreads, row, &mybegin, &myend);
            for ( i = mybegin; i < myend; ++i ) {
                switch ( vec[i] ) {
                    case FGPT:  // fine grid points
                        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
                            k = S->JA[j];
                            if ( vec[k] == CGPT ) P->IA[i+1]++;
                        }
                        break;

                    case CGPT:  // coarse grid points
                        P->IA[i+1] = 1; break;

                    default:  // treat everything else as isolated
                        P->IA[i+1] = 0; break;
                }
            }
        }

    }

    else {

        for ( i = 0; i < row; ++i ) {
            switch ( vec[i] ) {
                case FGPT:  // fine grid points
                    for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
                        k = S->JA[j];
                        if ( vec[k] == CGPT ) P->IA[i+1]++;
                    }
                    break;

                case CGPT:  // coarse grid points
```

```
                    P->IA[i+1] = 1; break;

                default:  // treat everything else as isolated
                    P->IA[i+1] = 0; break;
            }
        } // end for i

    } // end if

    // Form P->IA from the counter P
    for ( i = 0; i < P->row; ++i ) P->IA[i+1] += P->IA[i];
    P->nnz = P->IA[P->row]-P->IA[0];

    // step 2:  Find the structure JA of P
    P->JA  = (INT*)fasp_mem_calloc(P->nnz,sizeof(INT));
    P->val = (REAL*)fasp_mem_calloc(P->nnz,sizeof(REAL));

    for ( index = i = 0; i < row; ++i ) {
        if ( vec[i] == FGPT ) { // fine grid point
            for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
                k = S->JA[j];
                if ( vec[k] == CGPT ) P->JA[index++] = k;
            } // end for j
        } // end if
        else if ( vec[i] == CGPT ) { // coarse grid point -- one entry only
            P->JA[index++] = i;
        }
    }

}

static void form_P_pattern_std (dCSRmat   *P,
                                iCSRmat   *S,
                                ivector   *vertices,
                                INT        row,
                                INT        col)
{
    // local variables
    INT i, j, k, l, h, index;
    INT *vec = vertices->val;

    // number of times a C-point is visited
    INT *visited = (INT*)fasp_mem_calloc(row,sizeof(INT));

    P->row = row; P->col = col;
    P->IA  = (INT*)fasp_mem_calloc(row+1, sizeof(INT));

    fasp_iarray_set(row, visited, -1);

    // Step 1:  Find the structure IA of P first:  use P as a counter
    for ( i = 0; i < row; ++i ) {

        if ( vec[i] == FGPT ) { // if node i is a F point
            for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

                k = S->JA[j];

                // if neighbor of i is a C point, good
                if ( (vec[k] == CGPT) && (visited[k] != i) ) {
                    visited[k] = i;
                    P->IA[i+1]++;
                }

                // if k is a F point and k is not i, look for indirect C neighbors
                else if ( (vec[k] == FGPT) && (k != i) ) {
                    for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) { // neighbors of k
                        h = S->JA[l];
                        if ( (vec[h] == CGPT) && (visited[h] != i) ) {
                            visited[h] = i;
                            P->IA[i+1]++;
                        }
                    }  // end for(l=S->IA[k];l<S->IA[k+1];l++)
                }  // end if (vec[k]==CGPT)

            } // end for (j=S->IA[i];j<S->IA[i+1];j++)
        }

        else if ( vec[i] == CGPT ) { // if node i is a C point
            P->IA[i+1] = 1;
        }

        else { // treat everything else as isolated points
            P->IA[i+1] = 0;
        } // end if (vec[i]==FGPT)

    } // end for (i=0;i<row;++i)

    // Form P->IA from the counter P
    for ( i = 0; i < P->row; ++i ) P->IA[i+1] += P->IA[i];
    P->nnz = P->IA[P->row]-P->IA[0];

    // Step 2:  Find the structure JA of P
    P->JA  = (INT*)fasp_mem_calloc(P->nnz,sizeof(INT));
    P->val = (REAL*)fasp_mem_calloc(P->nnz,sizeof(REAL));

    fasp_iarray_set(row, visited, -1); // re-init visited array

    for ( i = 0; i < row; ++i ) {

        if ( vec[i] == FGPT ) { // if node i is a F point

            index = 0;

            for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

                k = S->JA[j];

                // if neighbor k of i is a C point
                if ( (vec[k] == CGPT) && (visited[k] != i) ) {
                    visited[k] = i;
                    P->JA[P->IA[i]+index] = k;
                    index++;
                }

                // if neighbor k of i is a F point and k is not i
                else if ( (vec[k] == FGPT) && (k != i) ) {
                    for ( l = S->IA[k]; l < S->IA[k+1]; l++ ) { // neighbors of k
                        h = S->JA[l];
                        if ( (vec[h] == CGPT) && (visited[h] != i) ) {
                            visited[h] = i;
                            P->JA[P->IA[i]+index] = h;
                            index++;
                        }

                    }  // end for (l=S->IA[k];l<S->IA[k+1];l++)

                }  // end if (vec[k]==CGPT)

            } // end for (j=S->IA[i];j<S->IA[i+1];j++)
        }

        else if ( vec[i] == CGPT ) {
            P->JA[P->IA[i]] = i;
        }
    }

    // clean up
    fasp_mem_free(visited); visited = NULL;
}

static INT cfsplitting_mis (iCSRmat   *S,
                            ivector   *vertices,
                            ivector   *order)
{
    const INT n = S->row;

    INT  col = 0;
    INT *ord = order->val;
    INT *vec = vertices->val;
    INT *IS = S->IA;
    INT *JS = S->JA;

    INT i, j, ind;
    INT row_begin, row_end;

    fasp_ivec_set (n, vertices, UNPT);

    for (i=0; i<n ; i++)
    {
        ind = ord[i];
        if (vec[ind] == UNPT) {
            vec[ind] = CGPT;
            row_begin = IS[ind]; row_end = IS[ind+1];
            for (j = row_begin; j<row_end; j++)
            {
                if (vec[JS[j]] == CGPT ) {
                    vec[ind] = FGPT;
                    break;
                }
            }
            if (vec[ind] == CGPT) {
                col++;
                for (j = row_begin; j<row_end; j++)
                {
                    vec[JS[j]] = FGPT;
                }
            }
        }
    }
    return col;
}

static void ordering1 (iCSRmat   *S,
                       ivector   *order)
{
    const INT n = order->row;
    INT * IS = S->IA;
    INT * ord = order->val;
    INT maxind, maxdeg, degree;
    INT i;

    for (i = 0; i < n; i++) ord[i] = i;

    for (maxind = maxdeg = i = 0; i < n; i++)
    {
        degree = IS[i+1] - IS[i];
        if (degree > maxdeg)
        {
            maxind = i;
            maxdeg = degree;
        }
    }

    ord[0] = maxind;
    ord[maxind] = 0;

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

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

A	Pointer to dCSRmat coefficient matrix (index starts from 0)
vertices	Indicator vector for the C/F splitting of the variables
P	Prolongation (input: nonzero pattern, output: prolongation)
S	Strong connection matrix
param	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.

```

#include <math.h>
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
```

```
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

static void interp_DIR (dCSRmat *, ivector *, dCSRmat *, AMG_param *);
static void interp_STD (dCSRmat *, ivector *, dCSRmat *, iCSRmat *, AMG_param *);
static void interp_EXT (dCSRmat *, ivector *, dCSRmat *, iCSRmat *, AMG_param *);
static void amg_interp_trunc (dCSRmat *, AMG_param *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_amg_interp (dCSRmat    *A,
                      ivector    *vertices,
                      dCSRmat    *P,
                      iCSRmat    *S,
                      AMG_param  *param)
{
    const INT coarsening_type = param->coarsening_type;
    INT       interp_type     = param->interpolation_type;

    // make sure standard interpolation is used for aggressive coarsening
    if ( coarsening_type == COARSE_AC ) interp_type = INTERP_STD;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    switch ( interp_type ) {

        case INTERP_DIR:  // Direct interpolation
            interp_DIR(A, vertices, P, param); break;

        case INTERP_STD:  // Standard interpolation
            interp_STD(A, vertices, P, S, param); break;

        case INTERP_EXT:  // Extended interpolation
            interp_EXT(A, vertices, P, S, param); break;

        case INTERP_ENG:  // Energy-min interpolation defined in PreAMGInterpEM.c
            fasp_amg_interp_em(A, vertices, P, param); break;

        default:
            fasp_chkerr(ERROR_AMG_INTERP_TYPE, __FUNCTION__);

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void amg_interp_trunc (dCSRmat    *P,
                              AMG_param  *param)
{
    const INT   row    = P->row;
    const INT   nnzold = P->nnz;
    const INT   prtlvl = param->print_level;
    const REAL  eps_tr = param->truncation_threshold;

    // local variables
    INT  num_nonzero = 0;    // number of non zeros after truncation
    REAL Min_neg, Max_pos;   // min negative and max positive entries
    REAL Fac_neg, Fac_pos;   // factors for negative and positive entries
    REAL Sum_neg, TSum_neg;  // sum and truncated sum of negative entries
    REAL Sum_pos, TSum_pos;  // sum and truncated sum of positive entries

    INT  index1 = 0, index2 = 0, begin, end;
    INT  i, j;

#if DEBUG_MODE > 0
```

```
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    for ( i = 0; i < row; ++i ) {

        begin = P->IA[i]; end = P->IA[i+1];

        P->IA[i] = num_nonzero;
        Min_neg  = Max_pos  = 0;
        Sum_neg  = Sum_pos  = 0;
        TSum_neg = TSum_pos = 0;

        // 1.  Summations of positive and negative entries
        for ( j = begin; j < end; ++j ) {

            if ( P->val[j] > 0 ) {
                Sum_pos += P->val[j];
                Max_pos = MAX(Max_pos, P->val[j]);
            }

            else {
                Sum_neg += P->val[j];
                Min_neg = MIN(Min_neg, P->val[j]);
            }

        }

        // Truncate according to max and min values!!!
        Max_pos *= eps_tr; Min_neg *= eps_tr;

        // 2.  Set JA of truncated P
        for ( j = begin; j < end; ++j ) {

            if ( P->val[j] >= Max_pos ) {
                num_nonzero++;
                P->JA[index1++] = P->JA[j];
                TSum_pos += P->val[j];
            }

            else if ( P->val[j] <= Min_neg ) {
                num_nonzero++;
                P->JA[index1++] = P->JA[j];
                TSum_neg += P->val[j];
            }

        }

        // 3.  Compute factors and set values of truncated P
        if ( TSum_pos > SMALLREAL ) {
            Fac_pos = Sum_pos / TSum_pos; // factor for positive entries
        }
        else {
            Fac_pos = 1.0;
        }

        if ( TSum_neg < -SMALLREAL ) {
            Fac_neg = Sum_neg / TSum_neg; // factor for negative entries
        }
        else {
            Fac_neg = 1.0;
        }

        for ( j = begin; j < end; ++j ) {

            if ( P->val[j] >= Max_pos )
                P->val[index2++] = P->val[j] * Fac_pos;

            else if ( P->val[j] <= Min_neg )
                P->val[index2++] = P->val[j] * Fac_neg;
        }

    }

    // resize the truncated prolongation P
    P->nnz = P->IA[row] = num_nonzero;
    P->JA  = (INT  *)fasp_mem_realloc(P->JA,  num_nonzero*sizeof(INT));
    P->val = (REAL *)fasp_mem_realloc(P->val, num_nonzero*sizeof(REAL));

    if ( prtlvl >= PRINT_MOST ) {
        printf("NNZ in prolongator:  before truncation = %10d, after = %10d\n",
               nnzold, num_nonzero);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

}

static void interp_DIR (dCSRmat    *A,
                        ivector    *vertices,
                        dCSRmat    *P,
                        AMG_param  *param )
{
    INT     row = A->row;
    INT    *vec = vertices->val;

    // local variables
    SHORT   IS_STRONG;   // is the variable strong coupled to i?
    INT     num_pcouple; // number of positive strong couplings
    INT     begin_row, end_row;
    INT     i, j, k, l, index = 0, idiag;

    // a_minus and a_plus for Neighbors and Prolongation support
    REAL    amN, amP, apN, apP;
    REAL    alpha, beta, aii = 0.0;

    // indices of C-nodes
    INT    *cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));

    SHORT   use_openmp = FALSE;

#ifdef _OPENMP
    INT myid, mybegin, myend, stride_i, nthreads;
//    row = MIN(P->IA[P->row], row);
    if ( MIN(P->IA[P->row], row) > OPENMP_HOLDS ) {
        use_openmp = TRUE;
        nthreads = fasp_get_num_threads();
    }
#endif

    // Step 1.  Fill in values for interpolation operator P
    if (use_openmp) {
#ifdef _OPENMP
        stride_i = row/nthreads;
#pragma omp parallel private(myid,mybegin,myend,i,begin_row,end_row,idiag,aii, \
amN,amP,apN,apP,num_pcouple,j,k,alpha,beta,l)      \
num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            mybegin = myid*stride_i;
            if (myid < nthreads-1) myend = mybegin+stride_i;
            else myend = row;
            aii = 0.0;

            for (i=mybegin; i<myend; ++i) {
                begin_row=A->IA[i]; end_row=A->IA[i+1]-1;
                for (idiag=begin_row;idiag<=end_row;idiag++) {
                    if (A->JA[idiag]==i) {
                        aii=A->val[idiag];
                        break;
                    }
                }
                if (vec[i]==0){  // if node i is on fine grid
                    amN=0, amP=0, apN=0, apP=0,  num_pcouple=0;
                    for (j=begin_row;j<=end_row;++j) {
                        if(j==idiag) continue;
                        for (k=P->IA[i];k<P->IA[i+1];++k) {
                            if (P->JA[k]==A->JA[j]) break;
                        }
                        if (A->val[j]>0) {
                            apN+=A->val[j];
                            if (k<P->IA[i+1]) {
                                apP+=A->val[j];
                                num_pcouple++;
                            }
                        }
                        else {
                            amN+=A->val[j];
                            if (k<P->IA[i+1]) {
                                amP+=A->val[j];
                            }
                        }
                    } // j

                    alpha=amN/amP;
                    if (num_pcouple>0) {
                        beta=apN/apP;
                    }
                    else {
                        beta=0;
                        aii+=apN;
                    }
                    for (j=P->IA[i];j<P->IA[i+1];++j) {
                        k=P->JA[j];
                        for (l=A->IA[i];l<A->IA[i+1];l++) {
                            if (A->JA[l]==k) break;
                        }
                        if (A->val[l]>0) {
                            P->val[j]=-beta*A->val[l]/aii;
                        }
                        else {
                            P->val[j]=-alpha*A->val[l]/aii;
                        }
                    }
                }
                else if (vec[i]==2) // if node i is a special fine node
                {

                }
                else {// if node i is on coarse grid
                    P->val[P->IA[i]]=1;
                }
            }
        }
#endif
    }

    else {

        for ( i = 0; i < row; ++i ) {

            begin_row = A->IA[i]; end_row = A->IA[i+1];

            // find diagonal entry first!!!
            for ( idiag = begin_row; idiag < end_row; idiag++ ) {
                if ( A->JA[idiag] == i ) {
                    aii = A->val[idiag]; break;
                }
            }

            if ( vec[i] == FGPT ) { // fine grid nodes

                amN = amP = apN = apP = 0.0;

                num_pcouple = 0;

                for ( j = begin_row; j < end_row; ++j ) {

                    if ( j == idiag ) continue; // skip diagonal

                    // check a point strong-coupled to i or not
                    IS_STRONG = FALSE;
                    for ( k = P->IA[i]; k < P->IA[i+1]; ++k ) {
                        if ( P->JA[k] == A->JA[j] ) { IS_STRONG = TRUE; break; }
                    }

                    if ( A->val[j] > 0 ) {
                        apN += A->val[j]; // sum up positive entries
                        if ( IS_STRONG ) { apP += A->val[j]; num_pcouple++; }
                    }
                    else {
                        amN += A->val[j]; // sum up negative entries
                        if ( IS_STRONG ) { amP += A->val[j]; }
                    }
                } // end for j

                // set weight factors
                alpha = amN / amP;
                if ( num_pcouple > 0 ) {
                    beta = apN / apP;
                }
                else {
                    beta = 0.0; aii += apN;
                }

                // keep aii inside the loop to avoid floating pt error!  --Chensong
                for ( j = P->IA[i]; j < P->IA[i+1]; ++j ) {
                    k = P->JA[j];
                    for ( l = A->IA[i]; l < A->IA[i+1]; l++ ) {
                        if ( A->JA[l] == k ) break;
                    }
                    if ( A->val[l] > 0 ) {
                        P->val[j] = -beta  * A->val[l] / aii;
                    }
                    else {
                        P->val[j] = -alpha * A->val[l] / aii;
                    }
                }

            } // end if vec

            else if ( vec[i] == CGPT ) { // coarse grid nodes
                P->val[P->IA[i]] = 1.0;
            }
        }
    }

    // Step 2.  Generate coarse level indices and set values of P.JA
    for ( index = i = 0; i < row; ++i ) {
        if ( vec[i] == CGPT ) cindex[i] = index++;
    }
    P->col = index;

    if (use_openmp) {
#ifdef _OPENMP
        stride_i = P->IA[P->row]/nthreads;
#pragma omp parallel private(myid,mybegin,myend,i,j) num_threads(nthreads)
        {
            myid = omp_get_thread_num();
            mybegin = myid*stride_i;
            if ( myid < nthreads-1 ) myend = mybegin+stride_i;
            else myend = P->IA[P->row];
            for ( i = mybegin; i < myend; ++i ) {
                j = P->JA[i];
                P->JA[i] = cindex[j];
            }
        }
#endif
    }
    else {
        for ( i = 0; i < P->nnz; ++i ) {
            j = P->JA[i];
            P->JA[i] = cindex[j];
        }
    }

    // clean up
    fasp_mem_free(cindex); cindex = NULL;

    // Step 3.  Truncate the prolongation operator to reduce cost
    amg_interp_trunc(P, param);
}

static void interp_STD (dCSRmat    *A,
                        ivector    *vertices,
                        dCSRmat    *P,
                        iCSRmat    *S,
                        AMG_param  *param)
{
    const INT   row    = A->row;
    INT        *vec    = vertices->val;

    // local variables
    INT    i, j, k, l, m, index;
    REAL   alpha = 1.0, factor, alN, alP;
    REAL   akk, akl, aik, aki;

    // indices for coarse neighbor node for every node
    INT  * cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));

    // indices from column number to index in nonzeros in i-th row
    INT  * rindi  = (INT *)fasp_mem_calloc(2*row, sizeof(INT));

    // indices from column number to index in nonzeros in k-th row
    INT  * rindk  = (INT *)fasp_mem_calloc(2*row, sizeof(INT));

    // sums of strongly connected C neighbors
    REAL * csum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

#if RS_C1
    // sums of all neighbors except ISPT
    REAL * psum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
#endif

    // sums of all neighbors
    REAL * nsum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // diagonal entries
    REAL * diag   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // coefficients hat a_ij for relevant CGPT of the i-th node
    REAL * Ahat   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // Step 0.  Prepare diagonal, Cs-sum, and N-sum
    fasp_iarray_set(row, cindex, -1);
    fasp_darray_set(row, csum, 0.0);
    fasp_darray_set(row, nsum, 0.0);

    for ( i = 0; i < row; i++ ) {

        // set flags for strong-connected C nodes
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
            k = S->JA[j];
            if ( vec[k] == CGPT ) cindex[k] = i;
        }

        for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
            k = A->JA[j];

            if ( cindex[k] == i ) csum[i] += A->val[j]; // strong C-couplings

            if ( k == i ) diag[i]  = A->val[j];
#if RS_C1
            else {
                nsum[i] += A->val[j];
                if ( vec[k] != ISPT ) {
                    psum[i] += A->val[j];
                }
            }
#else
            else nsum[i] += A->val[j];
#endif
        }

    }

    // Step 1.  Fill in values for interpolation operator P
    for ( i = 0; i < row; i++ ) {

        if ( vec[i] == FGPT ) {
#if RS_C1
            alN = psum[i];
#else
            alN = nsum[i];
#endif
            alP = csum[i];

            // form the reverse indices for i-th row
            for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) rindi[A->JA[j]] = j;

            // clean up Ahat for relevant nodes only
            for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) Ahat[P->JA[j]] = 0.0;

            // set values of Ahat
            Ahat[i] = diag[i];

            for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

                k = S->JA[j]; aik = A->val[rindi[k]];

                if ( vec[k] == CGPT ) Ahat[k] += aik;

                else if ( vec[k] == FGPT ) {

                    akk = diag[k];

                    // form the reverse indices for k-th row
                    for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) rindk[A->JA[m]] = m;

                    factor = aik / akk;

                    // visit the strong-connected C neighbors of k, compute
                    // Ahat in the i-th row, set aki if found
                    aki = 0.0;
#if 0               // modified by Xiaoqiang Yue 12/25/2013
                    for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
                        l   = S->JA[m];
                        akl = A->val[rindk[l]];
                        if ( vec[l] == CGPT ) Ahat[l] -= factor * akl;
                        else if ( l == i ) {
                            aki = akl; Ahat[l] -= factor * aki;
                        }
                    } // end for m
#else
                    for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) {
                        if ( A->JA[m] == i ) {
                            aki = A->val[m];
                            Ahat[i] -= factor * aki;
                        }
                    } // end for m
#endif
                    for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
                        l   = S->JA[m];
                        akl = A->val[rindk[l]];
                        if ( vec[l] == CGPT ) Ahat[l] -= factor * akl;
                    } // end for m

                    // compute Cs-sum and N-sum for Ahat
                    alN -= factor * (nsum[k]-aki+akk);
                    alP -= factor *  csum[k];

                } // end if vec[k]

            } // end for j

            // Originally:  alpha = alN/alP, do this only if P is not empty!
            if ( P->IA[i] < P->IA[i+1] ) alpha = alN/alP;

            // How about positive entries?  --Chensong
            for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) {
                k = P->JA[j];
                P->val[j] = -alpha*Ahat[k]/Ahat[i];
            }

        }

        else if ( vec[i] == CGPT ) {
            P->val[P->IA[i]] = 1.0;
        }

    } // end for i

    // Step 2.  Generate coarse level indices and set values of P.JA
    for ( index = i = 0; i < row; ++i ) {
        if ( vec[i] == CGPT ) cindex[i] = index++;
    }
    P->col = index;

#ifdef _OPENMP
#pragma omp parallel for private(i,j) if(P->IA[P->row]>OPENMP_HOLDS)
#endif
    for ( i = 0; i < P->IA[P->row]; ++i ) {
        j = P->JA[i];
        P->JA[i] = cindex[j];
    }

    // clean up
    fasp_mem_free(cindex); cindex = NULL;
    fasp_mem_free(rindi);  rindi  = NULL;
    fasp_mem_free(rindk);  rindk  = NULL;
    fasp_mem_free(nsum);   nsum   = NULL;
    fasp_mem_free(csum);   csum   = NULL;
    fasp_mem_free(diag);   diag   = NULL;
    fasp_mem_free(Ahat);   Ahat   = NULL;

```

```
#if RS_C1
    fasp_mem_free(psum);   psum   = NULL;
#endif

    // Step 3.  Truncate the prolongation operator to reduce cost
    amg_interp_trunc(P, param);
}

static void interp_EXT (dCSRmat    *A,
                        ivector    *vertices,
                        dCSRmat    *P,
                        iCSRmat    *S,
                        AMG_param  *param)
{
    const INT   row    = A->row;
    INT        *vec    = vertices->val;

    // local variables
    INT    i, j, k, l, m, index;
    REAL   alpha = 1.0, factor, alN, alP;
    REAL   akk, akl, aik, aki;

    // indices for coarse neighbor node for every node
    INT  * cindex = (INT *)fasp_mem_calloc(row, sizeof(INT));

    // indices from column number to index in nonzeros in i-th row
    INT  * rindi  = (INT *)fasp_mem_calloc(2*row, sizeof(INT));

    // indices from column number to index in nonzeros in k-th row
    INT  * rindk  = (INT *)fasp_mem_calloc(2*row, sizeof(INT));

    // sums of strongly connected C neighbors
    REAL * csum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

#if RS_C1
    // sums of all neighbors except ISPT
    REAL * psum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));
#endif
    // sums of all neighbors
    REAL * nsum   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // diagonal entries
    REAL * diag   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // coefficients hat a_ij for relevant CGPT of the i-th node
    REAL * Ahat   = (REAL *)fasp_mem_calloc(row, sizeof(REAL));

    // Step 0.  Prepare diagonal, Cs-sum, and N-sum
    fasp_iarray_set(row, cindex, -1);
    fasp_darray_set(row, csum, 0.0);
    fasp_darray_set(row, nsum, 0.0);

    for ( i = 0; i < row; i++ ) {

        // set flags for strong-connected C nodes
        for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {
            k = S->JA[j];
            if ( vec[k] == CGPT ) cindex[k] = i;
        }

        for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) {
            k = A->JA[j];

            if ( cindex[k] == i ) csum[i] += A->val[j]; // strong C-couplings

            if ( k == i ) diag[i]  = A->val[j];
#if RS_C1
            else {
                nsum[i] += A->val[j];
                if ( vec[k] != ISPT ) {
                    psum[i] += A->val[j];
                }
            }
#else
            else          nsum[i] += A->val[j];
#endif
        }

    }

    // Step 1.  Fill in values for interpolation operator P
    for ( i = 0; i < row; i++ ) {

        if ( vec[i] == FGPT ) {
#if RS_C1
            alN = psum[i];
#else
            alN = nsum[i];
#endif
            alP = csum[i];

            // form the reverse indices for i-th row
            for ( j = A->IA[i]; j < A->IA[i+1]; j++ ) rindi[A->JA[j]] = j;

            // clean up Ahat for relevant nodes only
            for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) Ahat[P->JA[j]] = 0.0;

            // set values of Ahat
            Ahat[i] = diag[i];

            for ( j = S->IA[i]; j < S->IA[i+1]; j++ ) {

                k = S->JA[j]; aik = A->val[rindi[k]];

                if ( vec[k] == CGPT ) Ahat[k] += aik;

                else if ( vec[k] == FGPT ) {

                    akk = diag[k];

                    // form the reverse indices for k-th row
                    for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) rindk[A->JA[m]] = m;

                    factor = aik / akk;

                    // visit the strong-connected C neighbors of k, compute
                    // Ahat in the i-th row, set aki if found
                    aki = 0.0;
#if 0               // modified by Xiaoqiang Yue 12/25/2013
                    for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
                        l   = S->JA[m];
                        akl = A->val[rindk[l]];
                        if ( vec[l] == CGPT ) Ahat[l] -= factor * akl;
                        else if ( l == i ) {
                            aki = akl; Ahat[l] -= factor * aki;
                        }
                    } // end for m
#else
                    for ( m = A->IA[k]; m < A->IA[k+1]; m++ ) {
                        if ( A->JA[m] == i ) {
                            aki = A->val[m];
                            Ahat[i] -= factor * aki;
                        }
                    } // end for m
#endif
                    for ( m = S->IA[k]; m < S->IA[k+1]; m++ ) {
                        l   = S->JA[m];
                        akl = A->val[rindk[l]];
                        if ( vec[l] == CGPT ) Ahat[l] -= factor * akl;
                    } // end for m

                    // compute Cs-sum and N-sum for Ahat
                    alN -= factor * (nsum[k]-aki+akk);
                    alP -= factor *  csum[k];

                } // end if vec[k]

            } // end for j

            // Originally:  alpha = alN/alP, do this only if P is not empty!
            if ( P->IA[i] < P->IA[i+1] ) alpha = alN/alP;

            // How about positive entries?  --Chensong
            for ( j = P->IA[i]; j < P->IA[i+1]; j++ ) {
                k = P->JA[j];
                P->val[j] = -alpha*Ahat[k]/Ahat[i];
            }

        }

        else if ( vec[i] == CGPT ) {
            P->val[P->IA[i]] = 1.0;
        }

    } // end for i

    // Step 2.  Generate coarse level indices and set values of P.JA
    for ( index = i = 0; i < row; ++i ) {
        if ( vec[i] == CGPT ) cindex[i] = index++;
    }
    P->col = index;

#ifdef _OPENMP
#pragma omp parallel for private(i,j) if(P->IA[P->row]>OPENMP_HOLDS)
#endif
    for ( i = 0; i < P->IA[P->row]; ++i ) {
        j = P->JA[i];
        P->JA[i] = cindex[j];
    }

    // clean up
    fasp_mem_free(cindex); cindex = NULL;
    fasp_mem_free(rindi);  rindi  = NULL;
    fasp_mem_free(rindk);  rindk  = NULL;
    fasp_mem_free(nsum);   nsum   = NULL;
    fasp_mem_free(csum);   csum   = NULL;
    fasp_mem_free(diag);   diag   = NULL;
    fasp_mem_free(Ahat);   Ahat   = NULL;

#if RS_C1
    fasp_mem_free(psum);   psum   = NULL;
#endif

    // Step 3.  Truncate the prolongation operator to reduce cost
    amg_interp_trunc(P, param);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

A	Pointer to dCSRmat: the coefficient matrix (index starts from 0)
vertices	Pointer to the indicator of CF splitting on fine or coarse grid
P	Pointer to the dCSRmat matrix of resulted interpolation
param	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);
```

```

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_amg_interp_em (dCSRmat    *A,
                         ivector    *vertices,
                         dCSRmat    *P,
                         AMG_param  *param)
{
    INT    *vec = vertices->val;
    INT    *CoarseIndex = (INT *)fasp_mem_calloc(vertices->row, sizeof(INT));
    INT     i, j, index;

    // generate indices for C-points
    for ( index = i = 0; i < vertices->row; ++i ) {
        if ( vec[i] == 1 ) {
            CoarseIndex[i] = index;
            index++;
        }
    }

#ifdef _OPENMP
#pragma omp parallel for private(i,j) if(P->nnz>OPENMP_HOLDS)
#endif
    for ( i = 0; i < P->nnz; ++i ) {
        j        = P->JA[i];
        P->JA[i] = CoarseIndex[j];
    }

    // clean up memory
    fasp_mem_free(CoarseIndex); CoarseIndex = NULL;

    // main part
    getiteval(A, P);
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static SHORT invden (INT    nn,
                     REAL   *mat,
                     REAL   *invmat)
{
    INT    i,j;
    SHORT  status = FASP_SUCCESS;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    INT  *pivot=(INT *)fasp_mem_calloc(nn,sizeof(INT));
    REAL *rhs=(REAL *)fasp_mem_calloc(nn,sizeof(REAL));
    REAL *sol=(REAL *)fasp_mem_calloc(nn,sizeof(REAL));

    fasp_smat_lu_decomp(mat,pivot,nn);

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,j) if(nn>OPENMP_HOLDS)
    for (myid=0; myid<nthreads; ++myid) {
        fasp_get_start_end(myid, nthreads, nn, &mybegin, &myend);
        for (i=mybegin; i<myend; ++i) {
#else
            for (i=0;i<nn;++i) {
#endif
                for (j=0;j<nn;++j) rhs[j]=0.;
                rhs[i]=1.;
                fasp_smat_lu_solve(mat,rhs,pivot,sol,nn);
                for (j=0;j<nn;++j) invmat[i*nn+j]=sol[j];
#ifdef _OPENMP
            }
        }
#else
    }
#endif

    fasp_mem_free(pivot); pivot = NULL;
    fasp_mem_free(rhs);   rhs   = NULL;
    fasp_mem_free(sol);   sol   = NULL;

    return status;
}

static SHORT get_block (dCSRmat  *A,
                        INT       m,
                        INT       n,
                        INT      *rows,
                        INT      *cols,
                        REAL     *Aloc,
                        INT      *mask)
{
    INT i, j, k, iloc;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    memset(Aloc, 0x0, sizeof(REAL)*m*n);

#ifdef _OPENMP
#pragma omp parallel for if(n>OPENMP_HOLDS) private(j)
#endif
    for ( j=0; j<n; ++j ) {
        mask[cols[j]] = j; // initialize mask, mask stores C indices 0,1,...
    }

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,j,k,iloc) if(m>OPENMP_HOLDS)
    for ( myid=0; myid<nthreads; ++myid ) {
        fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
        for ( i=mybegin; i<myend; ++i ) {
#else
            for ( i=0; i<m; ++i ) {
#endif
                iloc=rows[i];
                for ( k=A->IA[iloc]; k<A->IA[iloc+1]; ++k ) {
                    j = A->JA[k];
                    if (mask[j]>=0) Aloc[i*n+mask[j]]=A->val[k];
                } /* end for k */
#ifdef _OPENMP
            }
        }
#else
    } /* enf for i */
#endif

#ifdef _OPENMP
#pragma omp parallel for if(n>OPENMP_HOLDS) private(j)
#endif
    for ( j=0; j<n; ++j ) mask[cols[j]] = -1; // re-initialize mask

    return FASP_SUCCESS;
}

static SHORT gentisquare_nomass (dCSRmat  *A,
                                 INT       mm,
                                 INT      *Ii,
                                 REAL     *ima,
                                 INT      *mask)
{
    SHORT status = FASP_SUCCESS;

    REAL *ms = (REAL *)fasp_mem_calloc(mm*mm,sizeof(REAL));

    get_block(A,mm,mm,Ii,Ii,ms,mask);

    status = invden(mm,ms,ima);

    fasp_mem_free(ms); ms = NULL;

    return status;
}

static SHORT getinonefull (INT   **mat,
                           REAL  **matval,
                           INT    *lengths,
                           INT     mm,
                             INT     *Ii,
                             REAL    *ima)
{
    INT tniz,i,j;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif

    tniz=lengths[1];

#ifdef _OPENMP
#pragma omp parallel for private(myid,mybegin,myend,i,j) if(mm>OPENMP_HOLDS)
    for (myid=0; myid<nthreads; ++myid) {
        fasp_get_start_end(myid, nthreads, mm, &mybegin, &myend);
        for (i=mybegin; i<myend; ++i) {
#else
            for (i=0;i<mm;++i) {
#endif
                for (j=0;j<mm;++j) {
                    mat[0][tniz+i*mm+j]=Ii[i];
                    mat[1][tniz+i*mm+j]=Ii[j];
                    matval[0][tniz+i*mm+j]=ima[i*mm+j];
                }
#ifdef _OPENMP
            }
        }
#else
    }
#endif
    lengths[1]=tniz+mm*mm;

    return FASP_SUCCESS;
}

static SHORT orderone (INT    **mat,
                       REAL   **matval,
                       INT     *lengths)
//    lengths[0] for the number of rows
//    lengths[1] for the number of cols
//    lengths[2] for the number of nonzeros
{
    INT *rows[2],*cols[2],nns[2],tnizs[2];
    REAL *vals[2];
    SHORT status = FASP_SUCCESS;
    INT tniz,i;

    nns[0]=lengths[0];
    nns[1]=lengths[1];
    tnizs[0]=lengths[2];
    tniz=lengths[2];

    rows[0]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
    cols[0]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
    vals[0]=(REAL *)fasp_mem_calloc(tniz,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        rows[0][i]=mat[0][i];
        cols[0][i]=mat[1][i];
        vals[0][i]=matval[0][i];
    }

    rows[1]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
    cols[1]=(INT *)fasp_mem_calloc(tniz,sizeof(INT));
    vals[1]=(REAL *)fasp_mem_calloc(tniz,sizeof(REAL));

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

    // all the nonzeros with same col are gathering together

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        rows[0][i]=rows[1][i];
        cols[0][i]=cols[1][i];
        vals[0][i]=vals[1][i];
    }
    tnizs[1]=nns[0];
    nns[0]=nns[1];
    nns[1]=tnizs[1];
    tnizs[1]=tnizs[0];
    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

    // all the nonzeros with same col and row are gathering together
#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        rows[0][i]=rows[1][i];
        cols[0][i]=cols[1][i];
        vals[0][i]=vals[1][i];
    }
    tnizs[1]=nns[0];
    nns[0]=nns[1];
    nns[1]=tnizs[1];
    tnizs[1]=tnizs[0];

    tniz=tnizs[0];
    for (i=0;i<tniz-1;++i) {
        if (rows[0][i]==rows[0][i+1]&&cols[0][i]==cols[0][i+1]) {
            vals[0][i+1]+=vals[0][i];
            rows[0][i]=nns[0];
            cols[0][i]=nns[1];
        }
    }
    nns[0]=nns[0]+1;
    nns[1]=nns[1]+1;

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        rows[0][i]=rows[1][i];
        cols[0][i]=cols[1][i];
        vals[0][i]=vals[1][i];
    }
    tnizs[1]=nns[0];
    nns[0]=nns[1];
    nns[1]=tnizs[1];
    tnizs[1]=tnizs[0];

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        rows[0][i]=rows[1][i];
        cols[0][i]=cols[1][i];
        vals[0][i]=vals[1][i];
    }
    tnizs[1]=nns[0];
    nns[0]=nns[1];
    nns[1]=tnizs[1];
    tnizs[1]=tnizs[0];

    tniz=0;
    for (i=0;i<tnizs[0];++i)
        if (rows[0][i]<nns[0]-1) tniz++;

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) {
        mat[0][i]=rows[0][i];
        mat[1][i]=cols[0][i];
        matval[0][i]=vals[0][i];
    }
    nns[0]=nns[0]-1;
    nns[1]=nns[1]-1;
    lengths[0]=nns[0];
    lengths[1]=nns[1];
    lengths[2]=tniz;

    fasp_mem_free(rows[0]); rows[0] = NULL;
    fasp_mem_free(rows[1]); rows[1] = NULL;
    fasp_mem_free(cols[0]); cols[0] = NULL;
    fasp_mem_free(cols[1]); cols[1] = NULL;
    fasp_mem_free(vals[0]); vals[0] = NULL;
    fasp_mem_free(vals[1]); vals[1] = NULL;

    return(status);
}

static SHORT genintval (dCSRmat  *A,
                        INT     **itmat,
                        REAL    **itmatval,
                        INT       ittniz,
                        INT      *isol,
                        INT       numiso,
                        INT       nf,
                        INT       nc)
{
    INT  *Ii=NULL, *mask=NULL;
    REAL *ima=NULL, *pex=NULL, **imas=NULL;
    INT **mat=NULL;
    REAL **matval;
    INT lengths[3];
    dCSRmat T;
    INT tniz;
    dvector sol, rhs;

    INT mm,sum,i,j,k;
    INT *iz,*izs,*izt,*izts;
    SHORT status=FASP_SUCCESS;

    mask=(INT *)fasp_mem_calloc(nf,sizeof(INT));
    iz=(INT *)fasp_mem_calloc(nc,sizeof(INT));
    izs=(INT *)fasp_mem_calloc(nc,sizeof(INT));
    izt=(INT *)fasp_mem_calloc(nf,sizeof(INT));
    izts=(INT *)fasp_mem_calloc(nf,sizeof(INT));

    fasp_iarray_set(nf, mask, -1);

    memset(iz, 0, sizeof(INT)*nc);

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<ittniz;++i) iz[itmat[0][i]]++;

    izs[0]=0;
    for (i=1;i<nc;++i) izs[i]=izs[i-1]+iz[i-1];

    sum = 0;
#ifdef _OPENMP
#pragma omp parallel for reduction(+:sum) if(nc>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<nc;++i) sum+=iz[i]*iz[i];

    imas=(REAL **)fasp_mem_calloc(nc,sizeof(REAL *));

    for (i=0;i<nc;++i) {
        imas[i]=(REAL *)fasp_mem_calloc(iz[i]*iz[i],sizeof(REAL));
    }

    mat=(INT **)fasp_mem_calloc(2,sizeof(INT *));
    mat[0]=(INT *)fasp_mem_calloc((sum+numiso),sizeof(INT));
    mat[1]=(INT *)fasp_mem_calloc((sum+numiso),sizeof(INT));
    matval=(REAL **)fasp_mem_calloc(1,sizeof(REAL *));
    matval[0]=(REAL *)fasp_mem_calloc(sum+numiso,sizeof(REAL));

    lengths[1]=0;

    for (i=0;i<nc;++i) {

        mm=iz[i];
        Ii=(INT *)fasp_mem_realloc(Ii,mm*sizeof(INT));

#ifdef _OPENMP
#pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
#endif
        for (j=0;j<mm;++j) Ii[j]=itmat[1][izs[i]+j];

        ima=(REAL *)fasp_mem_realloc(ima,mm*mm*sizeof(REAL));

        gentisquare_nomass(A,mm,Ii,ima,mask);

        getinonefull(mat,matval,lengths,mm,Ii,ima);

#ifdef _OPENMP
#pragma omp parallel for if(mm*mm>OPENMP_HOLDS) private(j)
#endif
        for (j=0;j<mm*mm;++j) imas[i][j]=ima[j];
    }

#ifdef _OPENMP
#pragma omp parallel for if(numiso>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<numiso;++i) {
        mat[0][sum+i]=isol[i];
        mat[1][sum+i]=isol[i];
        matval[0][sum+i]=1.0;
    }

    lengths[0]=nf;
    lengths[2]=lengths[1]+numiso;
    lengths[1]=nf;
    orderone(mat,matval,lengths);
    tniz=lengths[2];

    sol.row=nf;
    sol.val=(REAL*)fasp_mem_calloc(nf,sizeof(REAL));

    memset(izt, 0, sizeof(INT)*nf);

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<tniz;++i) izt[mat[0][i]]++;

    T.IA=(INT*)fasp_mem_calloc((nf+1),sizeof(INT));

    T.row=nf;
    T.col=nf;
    T.nnz=tniz;
    T.IA[0]=0;
    for (i=1;i<nf+1;++i) T.IA[i]=T.IA[i-1]+izt[i-1];

    T.JA=(INT*)fasp_mem_calloc(tniz,sizeof(INT));

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(j)
#endif
    for (j=0;j<tniz;++j) T.JA[j]=mat[1][j];

    T.val=(REAL*)fasp_mem_calloc(tniz,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(tniz>OPENMP_HOLDS) private(j)
#endif
    for (j=0;j<tniz;++j) T.val[j]=matval[0][j];

    rhs.val=(REAL*)fasp_mem_calloc(nf,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(nf>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<nf;++i) rhs.val[i]=1.0;
    rhs.row=nf;

    // setup preconditioner
    dvector diag; fasp_dcsr_getdiag(0,&T,&diag);

    precond pc;
    pc.data = &diag;
    pc.fct  = fasp_precond_diag;

    status = fasp_solver_dcsr_pcg(&T,&rhs,&sol,&pc,1e-3,100,STOP_REL_RES,PRINT_NONE);

    for (i=0;i<nc;++i) {
        mm=iz[i];

        ima=(REAL *)fasp_mem_realloc(ima,mm*mm*sizeof(REAL));

        pex=(REAL *)fasp_mem_realloc(pex,mm*sizeof(REAL));

        Ii=(INT *)fasp_mem_realloc(Ii,mm*sizeof(INT));

#ifdef _OPENMP
#pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
#endif
        for (j=0;j<mm;++j) Ii[j]=itmat[1][izs[i]+j];

#ifdef _OPENMP
#pragma omp parallel for if(mm*mm>OPENMP_HOLDS) private(j)
#endif
        for (j=0;j<mm*mm;++j) ima[j]=imas[i][j];

#ifdef _OPENMP
#pragma omp parallel for if(mm>OPENMP_HOLDS) private(k,j)
#endif
        for (k=0;k<mm;++k) {
            for (pex[k]=j=0;j<mm;++j) pex[k]+=ima[k*mm+j]*sol.val[Ii[j]];
        }
#ifdef _OPENMP
#pragma omp parallel for if(mm>OPENMP_HOLDS) private(j)
#endif
        for (j=0;j<mm;++j) itmatval[0][izs[i]+j]=pex[j];

    }

    fasp_mem_free(ima); ima = NULL;
    fasp_mem_free(pex); pex = NULL;
    fasp_mem_free(Ii); Ii = NULL;
    fasp_mem_free(mask); mask = NULL;
    fasp_mem_free(iz); iz = NULL;
    fasp_mem_free(izs); izs = NULL;
    fasp_mem_free(izt); izt = NULL;
    fasp_mem_free(izts); izts = NULL;
    fasp_mem_free(mat[0]); mat[0] = NULL;
    fasp_mem_free(mat[1]); mat[1] = NULL;
    fasp_mem_free(mat); mat = NULL;
    fasp_mem_free(matval[0]); matval[0] = NULL;
    fasp_mem_free(matval); matval = NULL;
    for ( i=0; i<nc; ++i ) {fasp_mem_free(imas[i]); imas[i] = NULL;}
    fasp_mem_free(imas); imas = NULL;

    fasp_dcsr_free(&T);
    fasp_dvec_free(&rhs);
    fasp_dvec_free(&sol);
    fasp_dvec_free(&diag);

    return status;
}

static SHORT getiteval (dCSRmat  *A,
                        dCSRmat  *it)
{
    INT nf,nc,ittniz;
    INT *itmat[2];
    REAL **itmatval;
    INT *rows[2],*cols[2];
    REAL *vals[2];
    INT nns[2],tnizs[2];
    INT i,j,numiso;
    INT *isol;
    SHORT status = FASP_SUCCESS;

    nf=A->row;
    nc=it->col;
    ittniz=it->IA[nf];

    itmat[0]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    itmat[1]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    itmatval=(REAL **)fasp_mem_calloc(1,sizeof(REAL *));
    itmatval[0]=(REAL *)fasp_mem_calloc(ittniz,sizeof(REAL));
    isol=(INT *)fasp_mem_calloc(nf,sizeof(INT));

    numiso=0;
    for (i=0;i<nf;++i) {
        if (it->IA[i]==it->IA[i+1]) {
            isol[numiso]=i;
            numiso++;
        }
    }

```

```
#ifdef _OPENMP
#pragma omp parallel for if(nf>OPENMP_HOLDS) private(i,j)
#endif
    for (i=0;i<nf;++i) {
        for (j=it->IA[i];j<it->IA[i+1];++j) itmat[0][j]=i;
    }

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(j)
#endif
    for (j=0;j<ittniz;++j) {
        itmat[1][j]=it->JA[j];
        itmatval[0][j]=it->val[j];
    }

    rows[0]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    cols[0]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    vals[0]=(REAL *)fasp_mem_calloc(ittniz,sizeof(REAL));

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<ittniz;++i) {
        rows[0][i]=itmat[0][i];
        cols[0][i]=itmat[1][i];
        vals[0][i]=itmat[0][i];
    }

    nns[0]=nf;
    nns[1]=nc;
    tnizs[0]=ittniz;

    rows[1]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    cols[1]=(INT *)fasp_mem_calloc(ittniz,sizeof(INT));
    vals[1]=(REAL *)fasp_mem_calloc(ittniz,sizeof(REAL));

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<ittniz;++i) {
        itmat[0][i]=rows[1][i];
        itmat[1][i]=cols[1][i];
        itmatval[0][i]=vals[1][i];
    }
    genintval(A,itmat,itmatval,ittniz,isol,numiso,nf,nc);

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<ittniz;++i) {
        rows[0][i]=itmat[0][i];
        cols[0][i]=itmat[1][i];
        vals[0][i]=itmatval[0][i];
    }
    nns[0]=nc;
    nns[1]=nf;
    tnizs[0]=ittniz;

    fasp_dcsr_transpose(rows,cols,vals,nns,tnizs);

#ifdef _OPENMP
#pragma omp parallel for if(ittniz>OPENMP_HOLDS) private(i)
#endif
    for (i=0;i<ittniz;++i) it->val[i]=vals[1][i];

    fasp_mem_free(isol); isol = NULL;
    fasp_mem_free(itmat[0]); itmat[0] = NULL;
    fasp_mem_free(itmat[1]); itmat[1] = NULL;
    fasp_mem_free(itmatval[0]); itmatval[0] = NULL;
    fasp_mem_free(itmatval); itmatval = NULL;
    fasp_mem_free(rows[0]); rows[0] = NULL;
    fasp_mem_free(rows[1]); rows[1] = NULL;
    fasp_mem_free(cols[0]); cols[0] = NULL;
    fasp_mem_free(cols[1]); cols[1] = NULL;
    fasp_mem_free(vals[0]); vals[0] = NULL;
    fasp_mem_free(vals[1]); vals[1] = NULL;

    return status;
}
```

```

/*----------------------------------*/
/*--        End of File           --*/
/*----------------------------------*/
```

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

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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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_functs.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 --*/
```

```
        // fasp_dcsr_trans(mgl[level].P, mgl[level].R);

        /*-- Form coarse level stiffness matrix --*/
        //fasp_blas_dcsr_rap(mgl[level].R, mgl[level].A, mgl[level].P, mgl[level+1].A);

        ++level;

#if DIAGONAL_PREF
        fasp_dcsr_diagpref(&mgl[level].A); // reorder each row to make diagonal appear first
#endif
    }

    // setup total level number and current level
    mgl[0].num_levels = max_levels = level+1;
    mgl[0].w          = fasp_dvec_create(m);

    for ( level = 1; level < max_levels; ++level ) {
        INT mm = mgl[level].A.row;
        mgl[level].num_levels = max_levels;
        mgl[level].b = fasp_dvec_create(mm);
        mgl[level].x = fasp_dvec_create(mm);
        mgl[level].w = fasp_dvec_create(mm);
    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity(mgl,prtlvl);
        fasp_cputime("Compatible relaxation setup", setup_end - setup_start);
    }

    fasp_ivec_free(&vertices);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, BlaILUSetupCSR.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.

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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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_functs.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);
```

```
    const INT   m          = mgl[0].A.row;

    // local variables
    SHORT      status = FASP_SUCCESS;
    INT        lvl = 0, max_lvls = param->max_levels;
    REAL       setup_start, setup_end;
    ILU_param  iluparam;
    SWZ_param  swzparam;
    iCSRmat    Scouple; // strong n-couplings

    // level info (fine:  0; coarse:  1)
    ivector    vertices = fasp_ivec_create(m);

    // Output some info for debugging
    if ( prtlvl > PRINT_NONE ) printf("\nSetting up Classical AMG ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n = %d, nnz = %d\n", mgl[0].A.row, mgl[0].A.nnz);
#endif

    fasp_gettime(&setup_start);

    // Make sure classical AMG will not call fasp_blas_dcsr_mxv_agg!
    param->tentative_smooth = 1.0;

    // If user want to use aggressive coarsening but did not specify number of
    // levels use aggressive coarsening, make sure apply aggressive coarsening
    // on the finest level only !!!
    if ( param->coarsening_type == COARSE_AC ) {
        param->aggressive_level = MAX(param->aggressive_level, 1);
    }

    // Initialize AMLI coefficients
    if ( cycle_type == AMLI_CYCLE ) {
        const INT amlideg = param->amli_degree;
        param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1,sizeof(REAL));
        fasp_amg_amli_coef(2.0, 0.5, amlideg, param->amli_coef);
    }

    // Initialize ILU parameters
    mgl->ILU_levels = param->ILU_levels;
    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil    = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax   = param->ILU_relax;
        iluparam.ILU_type    = param->ILU_type;
    }

    // Initialize Schwarz parameters
    mgl->SWZ_levels = param->SWZ_levels;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_mmsize = param->SWZ_mmsize;
        swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
        swzparam.SWZ_type   = param->SWZ_type;
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

#if DIAGONAL_PREF
    // Reorder each row to keep the diagonal entries appear first !!!
    fasp_dcsr_diagpref(&mgl[0].A);
#endif

    // Main AMG setup loop
    while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_lvls-1) ) {

#if DEBUG_MODE > 1
        printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
               lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);
#endif

        /*-- Setup ILU decomposition if needed --*/
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /*-- Setup Schwarz smoother if needed --*/
        if ( lvl < param->SWZ_levels ) {
            mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
            fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
            status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
                }
                param->SWZ_levels = lvl;
            }
        }

        /*-- Coarsening and form the structure of interpolation --*/
        status = fasp_amg_coarsening_rs(&mgl[lvl].A, &vertices, &mgl[lvl].P,
                                        &Scouple, param);

        // Check 1:  Did coarsening step succeeded?
        if ( status < 0 ) {
            /*-- Clean up Scouple generated in coarsening --*/
            fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
            fasp_mem_free(Scouple.JA); Scouple.JA = NULL;

            // When error happens, stop at the current multigrid level!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Could not find any C-variables!\n");
                printf("### WARNING: Stop coarsening on level=%d!\n", lvl);
            }
            status = FASP_SUCCESS; break;
        }

        // Check 2:  Is coarse sparse too small?
        if ( mgl[lvl].P.col < MIN_CDOF ) {
            /*-- Clean up Scouple generated in coarsening --*/
            fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
            fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
            break;
        }

        // Check 3:  Does this coarsening step too aggressive?
        if ( mgl[lvl].P.row > mgl[lvl].P.col * 10.0 ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening might be too aggressive!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }

            /*-- Clean up Scouple generated in coarsening --*/
            fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
            fasp_mem_free(Scouple.JA); Scouple.JA = NULL;
            break;
        }

        /*-- Perform aggressive coarsening only up to the specified level --*/
        if ( mgl[lvl].P.col*1.5 > mgl[lvl].A.row ) param->coarsening_type = COARSE_RS;
        if ( lvl == param->aggressive_level ) param->coarsening_type = COARSE_RS;

        /*-- Store the C/F marker --*/
        {
            INT size = mgl[lvl].A.row;
            mgl[lvl].cfmark = fasp_ivec_create(size);
            memcpy(mgl[lvl].cfmark.val, vertices.val, size*sizeof(INT));
        }

        /*-- Form interpolation --*/
        fasp_amg_interp(&mgl[lvl].A, &vertices, &mgl[lvl].P, &Scouple, param);

        /*-- Form coarse level matrix:  two RAP routines available!  --*/
        fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);

        fasp_blas_dcsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);

        /*-- Clean up Scouple generated in coarsening --*/
        fasp_mem_free(Scouple.IA); Scouple.IA = NULL;
        fasp_mem_free(Scouple.JA); Scouple.JA = NULL;

        ++lvl;
```

```

#if DIAGONAL_PREF
        // reorder each row to make diagonal appear first
        fasp_dcsr_diagpref(&mgl[lvl].A);
#endif

        // Check 4:  Is the coarse matrix too dense?
        if ( mgl[lvl].A.nnz / mgl[lvl].A.row > mgl[lvl].A.col * 0.2 ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarse matrix is too dense!\n");
                printf("### WARNING: m = n = %d, nnz = %d!\n",
                       mgl[lvl].A.col, mgl[lvl].A.nnz);
            }

            break;
        }

    } // end of the main while loop

    // Setup coarse level systems for direct solvers
    switch (csolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].mumps.job = 1;
            fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            dCSRmat Ac_tran;
            Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
            fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
            // It is equivalent to do transpose and then sort
            //     fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
            //     fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
            break;
        }
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
             fasp_dcsr_sort(&mgl[lvl].A);
             fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
             break;
         }
#endif

        default:
            // Do nothing!
            break;
    }

    // setup total level number and current level
    mgl[0].num_levels = max_lvls = lvl+1;
    mgl[0].w          = fasp_dvec_create(m);

    for ( lvl = 1; lvl < max_lvls; ++lvl ) {
        const INT mm        = mgl[lvl].A.row;

        mgl[lvl].num_levels = max_lvls;
        mgl[lvl].b          = fasp_dvec_create(mm);
        mgl[lvl].x          = fasp_dvec_create(mm);

        mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
        mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
        mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!

        // allocate work arrays for the solve phase
        if ( cycle_type == NL_AMLI_CYCLE )
            mgl[lvl].w = fasp_dvec_create(3*mm);
        else
            mgl[lvl].w = fasp_dvec_create(2*mm);
    }

    fasp_ivec_free(&vertices);

#if MULTI_COLOR_ORDER
    INT Colors,rowmax;
#ifdef _OPENMP
    int threads = fasp_get_num_threads();
    if (threads  > max_lvls-1  ) threads = max_lvls-1;
#pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
#endif
    for (lvl=0; lvl<max_lvls-1; lvl++){

#if 1
        dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
#else
        dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
#endif
        if ( prtlvl > 1 )
            printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %le.\n",
            lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
            mgl[lvl].A.color, mgl[lvl].GS_Theta);
    }
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity(mgl, prtlvl);
        fasp_cputime("Classical AMG setup", setup_end - setup_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/

```

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, BlaILUSetupCSR.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

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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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_functs.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 *);
```

```c
static SHORT amg_setup_smoothP_unsmoothR (AMG_data *, AMG_param *);
static void smooth_agg (dCSRmat *, dCSRmat *, dCSRmat *, AMG_param *, dCSRmat *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_amg_setup_sa (AMG_data   *mgl,
                         AMG_param  *param)
{
    const SHORT prtlvl     = param->print_level;
    const SHORT smoothR    = param->smooth_restriction;
    SHORT status           = FASP_SUCCESS;

    // Output some info for debuging
    if ( prtlvl > PRINT_NONE ) printf("\nSetting up SA AMG ...\n");

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    if ( smoothR ) { // Default:  smoothed P, smoothed R
        status = amg_setup_smoothP_smoothR(mgl, param);
    }
    else { // smoothed P, unsmoothed R
        status = amg_setup_smoothP_unsmoothR(mgl, param);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void smooth_agg (dCSRmat    *A,
                        dCSRmat    *tentp,
                        dCSRmat    *P,
                        AMG_param  *param,
                        dCSRmat    *N)
{
    const SHORT filter = param->smooth_filter;
    const INT   row = A->row, col= A->col;
    const REAL  smooth_factor = param->tentative_smooth;

    dCSRmat S;
    dvector diag;  // diagonal entries

    REAL row_sum_A, row_sum_N;
    INT i,j;

    /* Step 1.  Form smoother */

    /* Without filter:  Using A for damped Jacobian smoother */
    if ( filter != ON ) {

        // copy structure from A
        S = fasp_dcsr_create(row, col, A->IA[row]);

#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
        for ( i=0; i<=row; ++i ) S.IA[i] = A->IA[i];
        for ( i=0; i<S.IA[S.row]; ++i ) S.JA[i] = A->JA[i];

        fasp_dcsr_getdiag(0, A, &diag);  // get the diagonal entries of A

        // check the diagonal entries.
        // if it is too small, use Richardson smoother for the corresponding row
#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
        for (i=0; i<row; ++i) {
            if (ABS(diag.val[i]) < 1e-6) diag.val[i] = 1.0;
        }

#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS) private(j)
#endif
        for (i=0; i<row; ++i) {
            for (j=S.IA[i]; j<S.IA[i+1]; ++j) {
                if (S.JA[j] == i) {
                    S.val[j] = 1 - smooth_factor * A->val[j] / diag.val[i];
                }
                else {
                    S.val[j] = - smooth_factor * A->val[j] / diag.val[i];
                }
            }
        }
    }

    /* Using filtered A for damped Jacobian smoother */
    else {
        /* Form filtered A and store in N */
#ifdef _OPENMP
#pragma omp parallel for private(j, row_sum_A, row_sum_N) if (row>OPENMP_HOLDS)
#endif
        for (i=0; i<row; ++i) {
            for (row_sum_A = 0.0, j=A->IA[i]; j<A->IA[i+1]; ++j) {
                if (A->JA[j] != i) row_sum_A += A->val[j];
            }

            for (row_sum_N = 0.0, j=N->IA[i]; j<N->IA[i+1]; ++j) {
                if (N->JA[j] != i) row_sum_N += N->val[j];
            }

            for (j=N->IA[i]; j<N->IA[i+1]; ++j) {
                if (N->JA[j] == i) {
                    // The original paper has a wrong sign!!!  --Chensong
                    N->val[j] += row_sum_A - row_sum_N;
                }
            }
        }

        // copy structure from N (filtered A)
        S = fasp_dcsr_create(row, col, N->IA[row]);

#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
        for (i=0; i<=row; ++i) S.IA[i] = N->IA[i];

        for (i=0; i<S.IA[S.row]; ++i) S.JA[i] = N->JA[i];

        fasp_dcsr_getdiag(0, N, &diag);  // get the diagonal entries of N (filtered A)

        // check the diagonal entries.
        // if it is too small, use Richardson smoother for the corresponding row
#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS)
#endif
        for (i=0;i<row;++i) {
            if (ABS(diag.val[i]) < 1e-6) diag.val[i] = 1.0;
        }

#ifdef _OPENMP
#pragma omp parallel for if(row>OPENMP_HOLDS) private(i,j)
#endif
        for (i=0;i<row;++i) {
            for (j=S.IA[i]; j<S.IA[i+1]; ++j) {
                if (S.JA[j] == i) {
                    S.val[j] = 1 - smooth_factor * N->val[j] / diag.val[i];
                }
                else {
                    S.val[j] = - smooth_factor * N->val[j] / diag.val[i];
                }
            }
        }

    }

    fasp_dvec_free(&diag);

    /* Step 2.  Smooth the tentative prolongation P = S*tenp */
    fasp_blas_dcsr_mxm(&S, tentp, P); // Note:  think twice about this.
    P->nnz = P->IA[P->row];
    fasp_dcsr_free(&S);
}

static SHORT amg_setup_smoothP_smoothR (AMG_data   *mgl,
                                        AMG_param  *param)
{
    const SHORT prtlvl     = param->print_level;
    const SHORT cycle_type = param->cycle_type;
    const SHORT csolver    = param->coarse_solver;
    const SHORT min_cdof   = MAX(param->coarse_dof,50);
    const INT   m          = mgl[0].A.row;

    // local variables
    SHORT       max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
    INT         i, j;
    REAL        setup_start, setup_end;
    ILU_param   iluparam;
    SWZ_param   swzparam;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    fasp_gettime(&setup_start);

    // level info (fine:  0; coarse:  1)
    ivector *vertices = (ivector *)fasp_mem_calloc(max_levels,sizeof(ivector));

    // each elvel stores the information of the number of aggregations
    INT *num_aggs = (INT *)fasp_mem_calloc(max_levels,sizeof(INT));

    // each level stores the information of the strongly coupled neighbourhood
    dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));

    // each level stores the information of the tentative prolongations
    dCSRmat *tentp = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));

    // Initialize level information
    for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;

    mgl[0].near_kernel_dim   = 1;
    mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim,sizeof(REAL*));

    for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
        mgl[0].near_kernel_basis[i] = (REAL *)fasp_mem_calloc(m,sizeof(REAL));
        for ( j = 0; j < m; ++j ) mgl[0].near_kernel_basis[i][j] = 1.0;
    }

    // Initialize ILU parameters
    mgl->ILU_levels = param->ILU_levels;
    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil    = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax   = param->ILU_relax;
        iluparam.ILU_type    = param->ILU_type;
    }

    // Initialize Schwarz parameters
    mgl->SWZ_levels = param->SWZ_levels;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_mmsize = param->SWZ_mmsize;
        swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
        swzparam.SWZ_type   = param->SWZ_type;
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

    // Initialize AMLI coefficients
    if ( cycle_type == AMLI_CYCLE ) {
        const INT amlideg = param->amli_degree;
        param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1,sizeof(REAL));
        REAL lambda_max = 2.0, lambda_min = lambda_max/4;
        fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
    }

#if DIAGONAL_PREF
    fasp_dcsr_diagpref(&mgl[0].A); // reorder each row to make diagonal appear first
#endif

    /*----------------------------*/
    /*--- checking aggregation ---*/
    /*----------------------------*/
    if ( param->aggregation_type == PAIRWISE )
        param->pair_number = MIN(param->pair_number, max_levels);

    // Main AMG setup loop
    while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_levels-1) ) {

#if DEBUG_MODE > 2
        printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
               lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);
#endif

        /*-- setup ILU decomposition if necessary */
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /* -- setup Schwarz smoother if necessary */
        if ( lvl < param->SWZ_levels ) {
            mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
            fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
            status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
                }
                param->SWZ_levels = lvl;
            }
        }

        /*-- Aggregation --*/
        status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl+1,
                                 &Neighbor[lvl], &num_aggs[lvl]);

        // Check 1:  Did coarsening step succeed?
        if ( status < 0 ) {
            // When error happens, stop at the current multigrid level!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Forming aggregates on level-%d failed!\n", lvl);
            }
            status = FASP_SUCCESS;
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            break;
        }

        /* -- Form Tentative prolongation --*/
        form_tentative_p(&vertices[lvl], &tentp[lvl], mgl[0].near_kernel_basis,
                         num_aggs[lvl]);

        /* -- Form smoothed prolongation -- */
        smooth_agg(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);

        // Check 2:  Is coarse sparse too small?
        if ( mgl[lvl].P.col < MIN_CDOF ) {
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            fasp_dcsr_free(&tentp[lvl]);
            break;
        }

        // Check 3:  Does this coarsening step too aggressive?
        if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening might be too aggressive!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            fasp_dcsr_free(&tentp[lvl]);
            break;
        }

        /*-- Form restriction --*/
        fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);

        /*-- Form coarse level stiffness matrix --*/
        fasp_blas_dcsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);

        fasp_dcsr_free(&Neighbor[lvl]);
        fasp_dcsr_free(&tentp[lvl]);
        fasp_ivec_free(&vertices[lvl]);

        ++lvl;

#if DIAGONAL_PREF
        // reorder each row to make diagonal appear first
        fasp_dcsr_diagpref(&mgl[lvl].A);
#endif

        // Check 4:  Is this coarsening ratio too small?
        if ( (REAL)mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening rate is too small!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }

            break;
        }

    } // end of the main while loop

    // Setup coarse level systems for direct solvers
    switch (csolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].mumps.job = 1;
            fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            dCSRmat Ac_tran;
            Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
            fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
            // It is equivalent to do transpose and then sort
            //     fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
            //     fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
            break;
        }
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
             fasp_dcsr_sort(&mgl[lvl].A);
             fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
             break;
         }
#endif

        default:
            // Do nothing!
            break;
    }

    // setup total level number and current level
    mgl[0].num_levels = max_levels = lvl+1;
    mgl[0].w          = fasp_dvec_create(m);

    for ( lvl = 1; lvl < max_levels; ++lvl) {
        INT mm = mgl[lvl].A.row;
        mgl[lvl].num_levels = max_levels;
        mgl[lvl].b          = fasp_dvec_create(mm);
        mgl[lvl].x          = fasp_dvec_create(mm);

        mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
        mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
        mgl[lvl].SWZ_levels = param->SWZ_levels -lvl; // initialize Schwarz!

        if ( cycle_type == NL_AMLI_CYCLE )
            mgl[lvl].w = fasp_dvec_create(3*mm);
        else
            mgl[lvl].w = fasp_dvec_create(2*mm);
    }

#if MULTI_COLOR_ORDER
    INT Colors,rowmax;
#ifdef _OPENMP
    int threads = fasp_get_num_threads();
    if (threads  > max_levels-1  ) threads = max_levels-1;
#pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
#endif
    for (lvl=0; lvl<max_levels-1; lvl++){

#if 1
        dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
#else
        dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
#endif
        if ( prtlvl > 1 )
            printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %le.\n",
            lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
            mgl[lvl].A.color, mgl[lvl].GS_Theta);
    }
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity(mgl,prtlvl);
        fasp_cputime("Smoothed aggregation setup", setup_end - setup_start);
    }

    fasp_mem_free(vertices); vertices = NULL;
    fasp_mem_free(num_aggs); num_aggs = NULL;
    fasp_mem_free(Neighbor); Neighbor = NULL;
    fasp_mem_free(tentp);    tentp    = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

static SHORT amg_setup_smoothP_unsmoothR (AMG_data   *mgl,
                                          AMG_param  *param)
{
    const SHORT prtlvl     = param->print_level;
    const SHORT cycle_type = param->cycle_type;
    const SHORT csolver    = param->coarse_solver;
    const SHORT min_cdof   = MAX(param->coarse_dof,50);
    const INT   m          = mgl[0].A.row;

    // local variables
    SHORT       max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
    INT         i, j;
    REAL        setup_start, setup_end;
    ILU_param   iluparam;
    SWZ_param swzparam;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    fasp_gettime(&setup_start);

    // level info (fine:  0; coarse:  1)
    ivector *vertices = (ivector *)fasp_mem_calloc(max_levels,sizeof(ivector));

    // each level stores the information of the number of aggregations
    INT *num_aggs = (INT *)fasp_mem_calloc(max_levels,sizeof(INT));

    // each level stores the information of the strongly coupled neighbourhood
    dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));

    // each level stores the information of the tentative prolongations
    dCSRmat *tentp = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));
    dCSRmat *tentr = (dCSRmat *)fasp_mem_calloc(max_levels,sizeof(dCSRmat));

    for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;

    mgl[0].near_kernel_dim   = 1;

    mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim,sizeof(REAL*));

    for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
        mgl[0].near_kernel_basis[i] = (REAL *)fasp_mem_calloc(m,sizeof(REAL));
        for ( j = 0; j < m; ++j ) mgl[0].near_kernel_basis[i][j] = 1.0;
    }

    // Initialize ILU parameters
    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil    = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax   = param->ILU_relax;
        iluparam.ILU_type    = param->ILU_type;
    }

    // Initialize Schwarz parameters
    mgl->SWZ_levels = param->SWZ_levels;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_mmsize = param->SWZ_mmsize;
        swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
        swzparam.SWZ_type   = param->SWZ_type;
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

    // Initialize AMLI coefficients
    if ( cycle_type == AMLI_CYCLE ) {
        const INT amlideg = param->amli_degree;
        param->amli_coef = (REAL *)fasp_mem_calloc(amlideg+1,sizeof(REAL));
        REAL lambda_max = 2.0, lambda_min = lambda_max/4;
        fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
    }

    // Main AMG setup loop
    while ( (mgl[lvl].A.row > min_cdof) && (lvl < max_levels-1) ) {

        /*-- setup ILU decomposition if necessary */
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /* -- setup Schwarz smoother if necessary */
        if ( lvl < param->SWZ_levels ) {
            mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
            fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
            status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
                }
                param->SWZ_levels = lvl;
            }
        }

        /*-- Aggregation --*/
        status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl+1,
                                 &Neighbor[lvl], &num_aggs[lvl]);

        // Check 1:  Did coarsening step succeeded?
        if ( status < 0 ) {
            // When error happens, stop at the current multigrid level!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Stop coarsening on level=%d!\n", lvl);
            }
            status = FASP_SUCCESS; break;
        }

        /* -- Form Tentative prolongation --*/
        form_tentative_p(&vertices[lvl], &tentp[lvl], mgl[0].near_kernel_basis,
                         num_aggs[lvl]);

        /* -- Form smoothed prolongation -- */
        smooth_agg(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);

        // Check 2:  Is coarse sparse too small?
        if ( mgl[lvl].P.col < MIN_CDOF ) break;

        // Check 3:  Does this coarsening step too aggressive?
        if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening might be too aggressive!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }
            break;
        }

        // Check 4:  Is this coarsening ratio too small?
        if ( (REAL)mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening rate is too small!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }
            break;
        }

        /*-- Form restriction --*/
        fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);
        fasp_dcsr_trans(&tentp[lvl], &tentr[lvl]);

        /*-- Form coarse level stiffness matrix --*/
        fasp_blas_dcsr_rap_agg(&tentr[lvl], &mgl[lvl].A, &tentp[lvl], &mgl[lvl+1].A);

        fasp_dcsr_free(&Neighbor[lvl]);
        fasp_dcsr_free(&tentp[lvl]);
        fasp_ivec_free(&vertices[lvl]);

        ++lvl;
    }

    // Setup coarse level systems for direct solvers
    switch (csolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].mumps.job = 1;
            fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            dCSRmat Ac_tran;
            Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
            fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
            // It is equivalent to do transpose and then sort
            //     fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
            //     fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
            break;
        }
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
             fasp_dcsr_sort(&mgl[lvl].A);
             fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
             break;
         }
#endif
```

```

        default:
            // Do nothing!
            break;
    }

    // setup total level number and current level
    mgl[0].num_levels = max_levels = lvl+1;
    mgl[0].w          = fasp_dvec_create(m);

    for ( lvl = 1; lvl < max_levels; ++lvl) {
        INT mm = mgl[lvl].A.row;
        mgl[lvl].num_levels = max_levels;
        mgl[lvl].b          = fasp_dvec_create(mm);
        mgl[lvl].x          = fasp_dvec_create(mm);

        mgl[lvl].cycle_type     = cycle_type; // initialize cycle type!
        mgl[lvl].ILU_levels     = param->ILU_levels - lvl; // initialize ILU levels!
        mgl[lvl].SWZ_levels = param->SWZ_levels -lvl; // initialize Schwarz!

        if ( cycle_type == NL_AMLI_CYCLE )
            mgl[lvl].w = fasp_dvec_create(3*mm);
        else
            mgl[lvl].w = fasp_dvec_create(2*mm);
    }

#if MULTI_COLOR_ORDER
    INT Colors,rowmax;
#ifdef _OPENMP
    int threads = fasp_get_num_threads();
    if (threads  > max_levels-1  ) threads = max_levels-1;
#pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
#endif
    for (lvl=0; lvl<max_levels-1; lvl++){

#if 1
        dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
#else
        dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
#endif
        if ( prtlvl > 1 )
            printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %le.\n",
            lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
            mgl[lvl].A.color, mgl[lvl].GS_Theta);
    }
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity(mgl,prtlvl);
        fasp_cputime("Smoothed aggregation 1/2 setup", setup_end - setup_start);
    }

    fasp_mem_free(vertices); vertices = NULL;
    fasp_mem_free(num_aggs); num_aggs = NULL;
    fasp_mem_free(Neighbor); Neighbor = NULL;
    fasp_mem_free(tentp);    tentp    = NULL;
    fasp_mem_free(tentr);    tentr    = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

mgl	Pointer to AMG data: AMG_data_bsr
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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.

```c

#include <math.h>
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreAMGAggregation.inl"
#include "PreAMGAggregationBSR.inl"
#include "PreAMGAggregationUA.inl"

static SHORT amg_setup_smoothP_smoothR_bsr (AMG_data_bsr *, AMG_param *);
static void smooth_agg_bsr (const dBSRmat *, dBSRmat *, dBSRmat *, const AMG_param *,
                            const dCSRmat *);

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

SHORT fasp_amg_setup_sa_bsr (AMG_data_bsr  *mgl,
                             AMG_param     *param)
{
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    SHORT status = amg_setup_smoothP_smoothR_bsr(mgl, param);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--      Private Functions      --*/
/*---------------------------------*/

static void smooth_agg_bsr (const dBSRmat    *A,
                            dBSRmat          *tentp,
                            dBSRmat          *P,
                            const AMG_param  *param,
                            const dCSRmat    *N)
{
    const INT   row = A->ROW, col= A->COL, nnz = A->NNZ;
    const INT   nb = A->nb, nb2 = nb*nb;
    const REAL  smooth_factor = param->tentative_smooth;

    // local variables
    dBSRmat S;
    dvector diaginv;  // diagonal block inv

    INT i, j;

    REAL *Id   = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));
    REAL *temp = (REAL *)fasp_mem_calloc(nb2, sizeof(REAL));

    fasp_smat_identity(Id, nb, nb2);

    /* Step 1.  Form smoother */

    // copy structure from A
    S = fasp_dbsr_create(row, col, nnz, nb, 0);

    for ( i=0; i<=row; ++i ) S.IA[i] = A->IA[i];
    for ( i=0; i<nnz;  ++i ) S.JA[i] = A->JA[i];

    diaginv = fasp_dbsr_getdiaginv(A);

    // for S
    for (i=0; i<row; ++i) {

        for (j=S.IA[i]; j<S.IA[i+1]; ++j) {

            if (S.JA[j] == i) {

                fasp_blas_smat_mul(diaginv.val+(i*nb2), A->val+(j*nb2), temp, nb);
                fasp_blas_smat_add(Id, temp, nb, 1.0, (-1.0)*smooth_factor, S.val+(j*nb2));

            }
            else {

                fasp_blas_smat_mul(diaginv.val+(i*nb2), A->val+(j*nb2), S.val+(j*nb2), nb);
                fasp_blas_smat_axm(S.val+(j*nb2), nb, (-1.0)*smooth_factor);

            }

        }

    }
    fasp_dvec_free(&diaginv);

    fasp_mem_free(Id);   Id   = NULL;
    fasp_mem_free(temp); temp = NULL;

    /* Step 2.  Smooth the tentative prolongation P = S*tenp */
    fasp_blas_dbsr_mxm(&S, tentp, P); // Note:  think twice about this.

    P->NNZ = P->IA[P->ROW];

    fasp_dbsr_free(&S);
}

static SHORT amg_setup_smoothP_smoothR_bsr (AMG_data_bsr *mgl,
                                            AMG_param *param)
{
    const SHORT CondType = 1; // Condensation method used for AMG

    const SHORT prtlvl     = param->print_level;
    const SHORT csolver    = param->coarse_solver;
    const SHORT min_cdof   = MAX(param->coarse_dof,50);
    const INT   m          = mgl[0].A.ROW;
    const INT   nb         = mgl[0].A.nb;

    ILU_param iluparam;
    SHORT     max_levels=param->max_levels;
    SHORT     i, lvl=0, status=FASP_SUCCESS;
    REAL      setup_start, setup_end;

    AMG_data *mgl_csr = fasp_amg_data_create(max_levels);

    dCSRmat temp1, temp2;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
           mgl[0].A.ROW, mgl[0].A.COL, mgl[0].A.NNZ);
#endif

    fasp_gettime(&setup_start);

    /*-----------------------*/
    /*--local working array--*/
    /*-----------------------*/

    // level info (fine:  0; coarse:  1)
    ivector *vertices = (ivector *)fasp_mem_calloc(max_levels, sizeof(ivector));

    // each level stores the information of the number of aggregations
    INT *num_aggs = (INT *)fasp_mem_calloc(max_levels, sizeof(INT));

    // each level stores the information of the strongly coupled neighbourhood
    dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));

    // each level stores the information of the tentative prolongations
    dBSRmat *tentp = (dBSRmat *)fasp_mem_calloc(max_levels,sizeof(dBSRmat));

    for ( i=0; i<max_levels; ++i ) num_aggs[i] = 0;

    /*-----------------------*/
    /*-- setup null spaces --*/
    /*-----------------------*/

    // null space for whole Jacobian
    //mgl[0].near_kernel_dim   = 1;
    //mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));

    //for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = NULL;

    /*-----------------------*/
    /*-- setup ILU param   --*/
    /*-----------------------*/

    // initialize ILU parameters
    mgl->ILU_levels = param->ILU_levels;
    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil    = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax   = param->ILU_relax;
        iluparam.ILU_type    = param->ILU_type;
    }

    /*----------------------------*/
    /*--- checking aggregation ---*/
    /*----------------------------*/

    if (param->aggregation_type == PAIRWISE)
        param->pair_number = MIN(param->pair_number, max_levels);

    // Main AMG setup loop
    while ( (mgl[lvl].A.ROW > min_cdof) && (lvl < max_levels-1) ) {

        /*-- setup ILU decomposition if necessary */
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dbsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /*-- get the diagonal inverse --*/
        mgl[lvl].diaginv = fasp_dbsr_getdiaginv(&mgl[lvl].A);

        switch ( CondType ) {
            case 2:
                mgl[lvl].PP = condenseBSR(&mgl[lvl].A); break;
            default:
                mgl[lvl].PP = condenseBSRLinf(&mgl[lvl].A); break;
        }

        /*-- Aggregation --*/
        switch ( param->aggregation_type ) {

            case NPAIR:  // unsymmetric pairwise matching aggregation

                mgl_csr[lvl].A = mgl[lvl].PP;
                status = aggregation_nsympair (mgl_csr, param, lvl, vertices,
                                               &num_aggs[lvl]);

                break;

            default:  // symmetric pairwise matching aggregation

                mgl_csr[lvl].A = mgl[lvl].PP;
                status = aggregation_symmpair (mgl_csr, param, lvl, vertices,
                                               &num_aggs[lvl]);

                // TODO: Need to design better algorithm for pairwise BSR -- Xiaozhe
                // TODO: Check why this fails for BSR --Chensong

                break;
        }

        if ( status < 0 ) {
            // When error happens, force solver to use the current multigrid levels!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Aggregation on level-%d failed!\n", lvl);
            }
            status = FASP_SUCCESS; break;
        }

        /* -- Form Tentative prolongation --*/
        if (lvl == 0 && mgl[0].near_kernel_dim >0 ){
            form_tentative_p_bsr1(&vertices[lvl], &tentp[lvl], &mgl[0],
                                  num_aggs[lvl], mgl[0].near_kernel_dim,
                                  mgl[0].near_kernel_basis);
        }
        else{
            form_boolean_p_bsr(&vertices[lvl], &tentp[lvl], &mgl[0], num_aggs[lvl]);
        }

        /* -- Smoothing -- */
        smooth_agg_bsr(&mgl[lvl].A, &tentp[lvl], &mgl[lvl].P, param, &Neighbor[lvl]);

        /*-- Form restriction --*/
        fasp_dbsr_trans(&mgl[lvl].P, &mgl[lvl].R);

        /*-- Form coarse level stiffness matrix --*/
        fasp_blas_dbsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);

        /*-- Form extra near kernel space if needed --*/
        if (mgl[lvl].A_nk != NULL){

            mgl[lvl+1].A_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));
            mgl[lvl+1].P_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));
            mgl[lvl+1].R_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));

            temp1 = fasp_format_dbsr_dcsr(&mgl[lvl].R);
            fasp_blas_dcsr_mxm(&temp1, mgl[lvl].P_nk, mgl[lvl+1].P_nk);
            fasp_dcsr_trans(mgl[lvl+1].P_nk, mgl[lvl+1].R_nk);
            temp2 = fasp_format_dbsr_dcsr(&mgl[lvl+1].A);
            fasp_blas_dcsr_rap(mgl[lvl+1].R_nk, &temp2, mgl[lvl+1].P_nk, mgl[lvl+1].A_nk);
            fasp_dcsr_free(&temp1);
            fasp_dcsr_free(&temp2);

        }

        fasp_dcsr_free(&Neighbor[lvl]);
        fasp_ivec_free(&vertices[lvl]);
        fasp_dbsr_free(&tentp[lvl]);

        ++lvl;
    }

    // Setup coarse level systems for direct solvers (BSR version)
    switch (csolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].mumps.job = 1;
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            fasp_solver_mumps_steps(&mgl[lvl].Ac, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            dCSRmat Ac_tran;
            fasp_dcsr_trans(&mgl[lvl].Ac, &Ac_tran);
            fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].Ac);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].Ac, 0);
            break;
        }
#endif

#if WITH_SuperLU
        case SOLVER_SUPERLU:  {
            /* Setup SuperLU direct solver on the coarsest level */
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
        }
```

```
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            fasp_dcsr_sort(&mgl[lvl].Ac);
            fasp_pardiso_factorize(&mgl[lvl].Ac, &mgl[lvl].pdata, prtlvl);
            break;
        }
#endif

        default:
            // Do nothing!
            break;
    }

    // setup total level number and current level
    mgl[0].num_levels = max_levels = lvl+1;
    mgl[0].w = fasp_dvec_create(3*m*nb);

    if (mgl[0].A_nk != NULL){

#if WITH_UMFPACK
        // Need to sort the matrix A_nk for UMFPACK
        fasp_dcsr_trans(mgl[0].A_nk, &temp1);
        fasp_dcsr_sort(&temp1);
        fasp_dcsr_cp(&temp1, mgl[0].A_nk);
        fasp_dcsr_free(&temp1);
        mgl[0].Numeric = fasp_umfpack_factorize(mgl[0].A_nk, 0);
#endif

    }

    for ( lvl = 1; lvl < max_levels; lvl++ ) {
        const INT mm = mgl[lvl].A.ROW*nb;
        mgl[lvl].num_levels = max_levels;
        mgl[lvl].b          = fasp_dvec_create(mm);
        mgl[lvl].x          = fasp_dvec_create(mm);
        mgl[lvl].w          = fasp_dvec_create(3*mm);
        mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!

        if (mgl[lvl].A_nk != NULL){

#if WITH_UMFPACK
            // Need to sort the matrix A_nk for UMFPACK
            fasp_dcsr_trans(mgl[lvl].A_nk, &temp1);
            fasp_dcsr_sort(&temp1);
            fasp_dcsr_cp(&temp1, mgl[lvl].A_nk);
            fasp_dcsr_free(&temp1);
            mgl[lvl].Numeric = fasp_umfpack_factorize(mgl[lvl].A_nk, 0);
#endif

        }

    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity_bsr(mgl,prtlvl);
        fasp_cputime("Smoothed aggregation (BSR) setup", setup_end - setup_start);
    }

    fasp_mem_free(vertices); vertices = NULL;
    fasp_mem_free(num_aggs); num_aggs = NULL;
    fasp_mem_free(Neighbor); Neighbor = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, BlaILUSetupCSR.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

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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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_functs.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 debuging
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
```

```
    SHORT max_levels = param->max_levels, lvl = 0, status = FASP_SUCCESS;
    INT i;
    REAL setup_start, setup_end;
    ILU_param iluparam;
    SWZ_param swzparam;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nr=%d, nc=%d, nnz=%d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    fasp_gettime(&setup_start);

    // level info (fine:  0; coarse:  1)
    ivector *vertices = (ivector *) fasp_mem_calloc(max_levels, sizeof(ivector));

    // each level stores the information of the number of aggregations
    INT *num_aggs = (INT *) fasp_mem_calloc(max_levels, sizeof(INT));

    // each level stores the information of the strongly coupled neighborhoods
    dCSRmat *Neighbor = (dCSRmat *) fasp_mem_calloc(max_levels, sizeof(dCSRmat));

    // Initialize level information
    for ( i = 0; i < max_levels; ++i ) num_aggs[i] = 0;

    mgl[0].near_kernel_dim = 1;
    mgl[0].near_kernel_basis = (REAL **) fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL * ));

    for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
        mgl[0].near_kernel_basis[i] = (REAL *) fasp_mem_calloc(m, sizeof(REAL));
        fasp_darray_set(m, mgl[0].near_kernel_basis[i], 1.0);
    }

    // Initialize ILU parameters
    mgl->ILU_levels = param->ILU_levels;
    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax = param->ILU_relax;
        iluparam.ILU_type = param->ILU_type;
    }

    // Initialize Schwarz parameters
    mgl->SWZ_levels = param->SWZ_levels;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_mmsize = param->SWZ_mmsize;
        swzparam.SWZ_maxlvl = param->SWZ_maxlvl;
        swzparam.SWZ_type = param->SWZ_type;
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

    // Initialize AMLI coefficients
    if ( cycle_type == AMLI_CYCLE ) {
        const INT amlideg = param->amli_degree;
        param->amli_coef = (REAL *) fasp_mem_calloc(amlideg + 1, sizeof(REAL));
        REAL lambda_max = 2.0, lambda_min = lambda_max / 4;
        fasp_amg_amli_coef(lambda_max, lambda_min, amlideg, param->amli_coef);
    }

#if DIAGONAL_PREF
    fasp_dcsr_diagpref(&mgl[0].A); // reorder each row to make diagonal appear first
#endif

    /*----------------------------*/
    /*--- checking aggregation ---*/
    /*----------------------------*/

    // Pairwise matching algorithm requires diagonal preference ordering
    if ( param->aggregation_type == PAIRWISE ) {
        param->pair_number = MIN(param->pair_number, max_levels);
        fasp_dcsr_diagpref(&mgl[0].A);
    }

    // Main AMG setup loop
    while ((mgl[lvl].A.row > min_cdof) && (lvl < max_levels - 1)) {

#if DEBUG_MODE > 1
        printf("### DEBUG: level = %d, row = %d, nnz = %d\n",
               lvl, mgl[lvl].A.row, mgl[lvl].A.nnz);
```

```
#endif

        /*-- Setup ILU decomposition if necessary */
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dcsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /*-- Setup Schwarz smoother if necessary */
        if ( lvl < param->SWZ_levels ) {
            mgl[lvl].Schwarz.A = fasp_dcsr_sympart(&mgl[lvl].A);
            fasp_dcsr_shift(&(mgl[lvl].Schwarz.A), 1);
            status = fasp_swz_dcsr_setup(&mgl[lvl].Schwarz, &swzparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: Schwarz on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable Schwarz for level >= %d.\n", lvl);
                }
                param->SWZ_levels = lvl;
            }
        }

        /*-- Aggregation --*/
        switch ( param->aggregation_type ) {

            case VMB:  // VMB aggregation
                status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl + 1,
                                         &Neighbor[lvl], &num_aggs[lvl]);

                /*-- Choose strength threshold adaptively --*/
                if ( num_aggs[lvl] * 4.0 > mgl[lvl].A.row )
                    param->strong_coupled /= 2.0;
                else if ( num_aggs[lvl] * 1.25 < mgl[lvl].A.row )
                    param->strong_coupled *= 2.0;

                break;

            case NPAIR:  // non-symmetric pairwise matching aggregation
                status = aggregation_nsympair(mgl, param, lvl, vertices, &num_aggs[lvl]);
                /*-- Modified by Chunsheng Feng on 10/17/2020, ZCS on 01/15/2021:
if NPAIR fail, switch aggregation type to VBM. --*/
                if ( status != FASP_SUCCESS || num_aggs[lvl] * 2.0 > mgl[lvl].A.row ) {
                    if ( prtlvl > PRINT_MORE ) {
                        printf("### WARNING: Non-symmetric pairwise matching failed on level %d!\n", lvl);
                        printf("### WARNING: Switch to VMB aggregation on level %d!\n", lvl);
                    }
                    param->aggregation_type = VMB;
                    status = aggregation_vmb(&mgl[lvl].A, &vertices[lvl], param, lvl + 1,
                                             &Neighbor[lvl], &num_aggs[lvl]);
                }

                break;

            default:  // symmetric pairwise matching aggregation
                status = aggregation_symmpair(mgl, param, lvl, vertices, &num_aggs[lvl]);

        }

        // Check 1:  Did coarsening step succeed?
        if ( status < 0 ) {
            // When error happens, stop at the current multigrid level!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Stop coarsening on level %d!\n", lvl);
            }
            status = FASP_SUCCESS;
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            break;
        }

        /*-- Form Prolongation --*/
        form_tentative_p(&vertices[lvl], &mgl[lvl].P, mgl[0].near_kernel_basis,
                         num_aggs[lvl]);

        // Check 2:  Is coarse sparse too small?
        if ( mgl[lvl].P.col < MIN_CDOF ) {
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            break;
        }

        // Check 3:  Does this coarsening step too aggressive?
        if ( mgl[lvl].P.row > mgl[lvl].P.col * MAX_CRATE ) {
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Coarsening might be too aggressive!\n");
                printf("### WARNING: Fine level = %d, coarse level = %d.  Discard!\n",
                       mgl[lvl].P.row, mgl[lvl].P.col);
            }
            fasp_ivec_free(&vertices[lvl]);
            fasp_dcsr_free(&Neighbor[lvl]);
            break;
        }

        /*-- Form restriction --*/
        fasp_dcsr_trans(&mgl[lvl].P, &mgl[lvl].R);

        /*-- Form coarse level stiffness matrix --*/
        fasp_blas_dcsr_rap_agg(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P,
                               &mgl[lvl + 1].A);

        fasp_dcsr_free(&Neighbor[lvl]);
        fasp_ivec_free(&vertices[lvl]);

        ++lvl;

#if DIAGONAL_PREF
        fasp_dcsr_diagpref(&mgl[lvl].A); // reorder each row to make diagonal appear first
#endif

        // Check 4:  Is this coarsening ratio too small?
        if ((REAL) mgl[lvl].P.col > mgl[lvl].P.row * MIN_CRATE ) {
            param->quality_bound *= 2.0;
        }

    } // end of the main while loop

    // Setup coarse level systems for direct solvers
    switch ( csolver ) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].mumps.job = 1;
            fasp_solver_mumps_steps(&mgl[lvl].A, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            dCSRmat Ac_tran;
            Ac_tran = fasp_dcsr_create(mgl[lvl].A.row, mgl[lvl].A.col, mgl[lvl].A.nnz);
            fasp_dcsr_transz(&mgl[lvl].A, NULL, &Ac_tran);
            // It is equivalent to do transpose and then sort
            //     fasp_dcsr_trans(&mgl[lvl].A, &Ac_tran);
            //     fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].A);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].A, 0);
            break;
        }
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
             fasp_dcsr_sort(&mgl[lvl].A);
             fasp_pardiso_factorize(&mgl[lvl].A, &mgl[lvl].pdata, prtlvl);
             break;
         }
#endif

        default:  // Do nothing!
            break;
    }

    // setup total level number and current level
    mgl[0].num_levels = max_levels = lvl + 1;
    mgl[0].w = fasp_dvec_create(m);

    for ( lvl = 1; lvl < max_levels; ++lvl ) {
        INT mm = mgl[lvl].A.row;
        mgl[lvl].num_levels = max_levels;
        mgl[lvl].b = fasp_dvec_create(mm);
        mgl[lvl].x = fasp_dvec_create(mm);

        mgl[lvl].cycle_type = cycle_type; // initialize cycle type!
        mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!
        mgl[lvl].SWZ_levels = param->SWZ_levels - lvl; // initialize Schwarz!

        if ( cycle_type == NL_AMLI_CYCLE )
            mgl[lvl].w = fasp_dvec_create(3 * mm);
        else
            mgl[lvl].w = fasp_dvec_create(2 * mm);
    }

    // setup for cycle type of unsmoothed aggregation
    eta = xsi / ((1 - xsi) * (cplxmax - 1));
    mgl[0].cycle_type = 1;
    mgl[max_levels - 1].cycle_type = 0;

    for ( lvl = 1; lvl < max_levels - 1; ++lvl ) {
        fracratio = (REAL) mgl[lvl].A.nnz / mgl[0].A.nnz;
        mgl[lvl].cycle_type = (INT)(pow((REAL) xsi, (REAL) lvl) / (eta * fracratio * icum));
        // safe-guard step:  make cycle type >= 1 and <= 2
        mgl[lvl].cycle_type = MAX(1, MIN(2, mgl[lvl].cycle_type));
        icum = icum * mgl[lvl].cycle_type;
    }

#if MULTI_COLOR_ORDER
    INT Colors,rowmax;
#ifdef _OPENMP
    int threads = fasp_get_num_threads();
    if (threads  > max_levels-1  ) threads = max_levels-1;
#pragma omp parallel for private(lvl,rowmax,Colors) schedule(static, 1) num_threads(threads)
#endif
    for (lvl=0; lvl<max_levels-1; lvl++){

#if 1
        dCSRmat_Multicoloring(&mgl[lvl].A, &rowmax, &Colors);
#else
        dCSRmat_Multicoloring_Theta(&mgl[lvl].A, mgl[lvl].GS_Theta, &rowmax, &Colors);
#endif
        if ( prtlvl > 1 )
            printf("mgl[%3d].A.row = %12d, rowmax = %5d, rowavg = %7.2lf, colors = %5d, Theta = %le.\n",
            lvl, mgl[lvl].A.row, rowmax, (double)mgl[lvl].A.nnz/mgl[lvl].A.row,
            mgl[lvl].A.color, mgl[lvl].GS_Theta);
    }
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity(mgl, prtlvl);
        fasp_cputime("Unsmoothed aggregation setup", setup_end - setup_start);
    }

    fasp_mem_free(Neighbor);
    Neighbor = NULL;
    fasp_mem_free(vertices);
    vertices = NULL;
    fasp_mem_free(num_aggs);
    num_aggs = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

mgl	Pointer to AMG data: AMG_data_bsr
param	Pointer to AMG parameters: AMG_param

Returns

FASP_SUCCESS if successed; 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",
               mgl[0].A.ROW, mgl[0].A.COL, mgl[0].A.NNZ);
#endif

    fasp_gettime(&setup_start);

    /*-----------------------*/
    /*--local working array--*/
    /*-----------------------*/

    // level info (fine:  0; coarse:  1)
    ivector *vertices = (ivector *)fasp_mem_calloc(max_levels, sizeof(ivector));

    //each elvel stores the information of the number of aggregations
    INT *num_aggs = (INT *)fasp_mem_calloc(max_levels, sizeof(INT));

    // each level stores the information of the strongly coupled neighborhoods
    dCSRmat *Neighbor = (dCSRmat *)fasp_mem_calloc(max_levels, sizeof(dCSRmat));

    for ( i=0; i<max_levels; ++i ) num_aggs[i] = 0;

    /*-----------------------------------------*/
    /*-- setup null spaces for whole Jacobian --*/
    /*-----------------------------------------*/

    /*
mgl[0].near_kernel_dim   = 1;
mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));

for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = NULL;
*/

    /*-----------------------*/
    /*-- setup ILU param   --*/
    /*-----------------------*/

    // initialize ILU parameters
    mgl->ILU_levels = param->ILU_levels;
    ILU_param iluparam;

    if ( param->ILU_levels > 0 ) {
        iluparam.print_level = param->print_level;
        iluparam.ILU_lfil    = param->ILU_lfil;
        iluparam.ILU_droptol = param->ILU_droptol;
        iluparam.ILU_relax   = param->ILU_relax;
        iluparam.ILU_type    = param->ILU_type;
    }

    /*----------------------------*/
    /*--- checking aggregation ---*/
    /*----------------------------*/
    if (param->aggregation_type == PAIRWISE)
        param->pair_number = MIN(param->pair_number, max_levels);

    // Main AMG setup loop
    while ( (mgl[lvl].A.ROW > min_cdof) && (lvl < max_levels-1) ) {

        /*-- setup ILU decomposition if necessary */
        if ( lvl < param->ILU_levels ) {
            status = fasp_ilu_dbsr_setup(&mgl[lvl].A, &mgl[lvl].LU, &iluparam);
            if ( status < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: ILU setup on level-%d failed!\n", lvl);
                    printf("### WARNING: Disable ILU for level >= %d.\n", lvl);
                }
                param->ILU_levels = lvl;
            }
        }

        /*-- get the diagonal inverse --*/
        mgl[lvl].diaginv = fasp_dbsr_getdiaginv(&mgl[lvl].A);

        switch ( CondType ) {
            case 2:
                mgl[lvl].PP = condenseBSR(&mgl[lvl].A); break;
            default:
                mgl[lvl].PP = condenseBSRLinf(&mgl[lvl].A); break;
        }

        /*-- Aggregation --*/
        switch ( param->aggregation_type ) {

            case VMB:  // VMB aggregation

                status = aggregation_vmb (&mgl[lvl].PP, &vertices[lvl], param,
                                          lvl+1, &Neighbor[lvl], &num_aggs[lvl]);

                /*-- Choose strength threshold adaptively --*/
                if ( num_aggs[lvl]*4 > mgl[lvl].PP.row )
                    param->strong_coupled /= 4;
                else if ( num_aggs[lvl]*1.25 < mgl[lvl].PP.row )
                    param->strong_coupled *= 1.5;

                break;

            case NPAIR:  // non-symmetric pairwise matching aggregation

                mgl_csr[lvl].A = mgl[lvl].PP;
                status = aggregation_nsympair (mgl_csr, param, lvl, vertices,
                                               &num_aggs[lvl]);

                break;

            default:  // symmetric pairwise matching aggregation

                mgl_csr[lvl].A = mgl[lvl].PP;
                status = aggregation_symmpair (mgl_csr, param, lvl, vertices,
                                              &num_aggs[lvl]);

                // TODO: Need to design better algorithm for pairwise BSR -- Xiaozhe
                // TODO: Unsymmetric pairwise aggregation not finished -- Chensong
                // TODO: Check why this fails for BSR --Chensong

               break;
        }

        if ( status < 0 ) {
            // When error happens, force solver to use the current multigrid levels!
            if ( prtlvl > PRINT_MIN ) {
                printf("### WARNING: Forming aggregates on level-%d failed!\n", lvl);
            }
            status = FASP_SUCCESS; break;
        }

        /* -- Form Prolongation --*/
        if ( lvl == 0 && mgl[0].near_kernel_dim >0 ) {
            form_tentative_p_bsr1(&vertices[lvl], &mgl[lvl].P, &mgl[0],
                                  num_aggs[lvl], mgl[0].near_kernel_dim,
                                  mgl[0].near_kernel_basis);
        }
        else {
            form_boolean_p_bsr(&vertices[lvl], &mgl[lvl].P, &mgl[0], num_aggs[lvl]);
        }

        /*-- Form resitriction --*/
        fasp_dbsr_trans(&mgl[lvl].P, &mgl[lvl].R);

        /*-- Form coarse level stiffness matrix --*/
        fasp_blas_dbsr_rap(&mgl[lvl].R, &mgl[lvl].A, &mgl[lvl].P, &mgl[lvl+1].A);

        /* -- Form extra near kernal space if needed --*/
        if (mgl[lvl].A_nk != NULL){

            mgl[lvl+1].A_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));
            mgl[lvl+1].P_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));
            mgl[lvl+1].R_nk = (dCSRmat *)fasp_mem_calloc(1, sizeof(dCSRmat));

            temp1 = fasp_format_dbsr_dcsr(&mgl[lvl].R);
            fasp_blas_dcsr_mxm(&temp1, mgl[lvl].P_nk, mgl[lvl+1].P_nk);
            fasp_dcsr_trans(mgl[lvl+1].P_nk, mgl[lvl+1].R_nk);
            temp2 = fasp_format_dbsr_dcsr(&mgl[lvl+1].A);
            fasp_blas_dcsr_rap(mgl[lvl+1].R_nk, &temp2, mgl[lvl+1].P_nk, mgl[lvl+1].A_nk);
            fasp_dcsr_free(&temp1);
            fasp_dcsr_free(&temp2);

        }

        fasp_dcsr_free(&Neighbor[lvl]);
        fasp_ivec_free(&vertices[lvl]);

        ++lvl;
    }

    // Setup coarse level systems for direct solvers (BSR version)
    switch (csolver) {

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // Setup MUMPS direct solver on the coarsest level
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            mgl[lvl].mumps.job = 1;
            fasp_solver_mumps_steps(&mgl[lvl].Ac, &mgl[lvl].b, &mgl[lvl].x, &mgl[lvl].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // Need to sort the matrix A for UMFPACK to work
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            dCSRmat Ac_tran;
            fasp_dcsr_trans(&mgl[lvl].Ac, &Ac_tran);
            fasp_dcsr_sort(&Ac_tran);
            fasp_dcsr_cp(&Ac_tran, &mgl[lvl].Ac);
            fasp_dcsr_free(&Ac_tran);
            mgl[lvl].Numeric = fasp_umfpack_factorize(&mgl[lvl].Ac, 0);
            break;
        }
#endif

#if WITH_SuperLU
        case SOLVER_SUPERLU:  {
            /* Setup SuperLU direct solver on the coarsest level */
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
        }
#endif

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
            mgl[lvl].Ac = fasp_format_dbsr_dcsr(&mgl[lvl].A);
            fasp_dcsr_sort(&mgl[lvl].Ac);
            fasp_pardiso_factorize(&mgl[lvl].Ac, &mgl[lvl].pdata, prtlvl);
            break;
         }
#endif

        default:
            // Do nothing!
            break;
    }


    // setup total level number and current level
    mgl[0].num_levels = max_levels = lvl+1;
    mgl[0].w = fasp_dvec_create(3*m*nb);

    if (mgl[0].A_nk != NULL){

#if WITH_UMFPACK
        // Need to sort the matrix A_nk for UMFPACK
        fasp_dcsr_trans(mgl[0].A_nk, &temp1);
        fasp_dcsr_sort(&temp1);
        fasp_dcsr_cp(&temp1, mgl[0].A_nk);
        fasp_dcsr_free(&temp1);
#endif

    }

    for ( lvl = 1; lvl < max_levels; lvl++ ) {
        const INT mm = mgl[lvl].A.ROW*nb;
        mgl[lvl].num_levels = max_levels;
        mgl[lvl].b          = fasp_dvec_create(mm);
        mgl[lvl].x          = fasp_dvec_create(mm);
        mgl[lvl].w          = fasp_dvec_create(3*mm);
        mgl[lvl].ILU_levels = param->ILU_levels - lvl; // initialize ILU levels!

        if (mgl[lvl].A_nk != NULL){

#if WITH_UMFPACK
            // Need to sort the matrix A_nk for UMFPACK
            fasp_dcsr_trans(mgl[lvl].A_nk, &temp1);
            fasp_dcsr_sort(&temp1);
            fasp_dcsr_cp(&temp1, mgl[lvl].A_nk);
            fasp_dcsr_free(&temp1);
#endif

        }

    }

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&setup_end);
        fasp_amgcomplexity_bsr(mgl,prtlvl);
        fasp_cputime("Unsmoothed aggregation (BSR) setup", setup_end - setup_start);
    }

    fasp_mem_free(vertices); vertices = NULL;
    fasp_mem_free(num_aggs); num_aggs = NULL;
    fasp_mem_free(Neighbor); Neighbor = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.155 PreBLC.c File Reference

Preconditioners for dBLCmat matrices.

```
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"
```

Functions

- void fasp_precond_dblc_diag_3 (REAL ∗r, REAL ∗z, void ∗data)
  Block diagonal preconditioner (3x3 blocks)
- void fasp_precond_dblc_diag_3_amg (REAL ∗r, REAL ∗z, void ∗data)
  Block diagonal preconditioning (3x3 blocks)
- void fasp_precond_dblc_diag_4 (REAL ∗r, REAL ∗z, void ∗data)
  Block diagonal preconditioning (4x4 blocks)
- void fasp_precond_dblc_lower_3 (REAL ∗r, REAL ∗z, void ∗data)
  block lower triangular preconditioning (3x3 blocks)
- void fasp_precond_dblc_lower_3_amg (REAL ∗r, REAL ∗z, void ∗data)
  block lower triangular preconditioning (3x3 blocks)
- void fasp_precond_dblc_lower_4 (REAL ∗r, REAL ∗z, void ∗data)
  block lower triangular preconditioning (4x4 blocks)
- void fasp_precond_dblc_upper_3 (REAL ∗r, REAL ∗z, void ∗data)
  block upper triangular preconditioning (3x3 blocks)
- void fasp_precond_dblc_upper_3_amg (REAL ∗r, REAL ∗z, void ∗data)
  block upper triangular preconditioning (3x3 blocks)
- void fasp_precond_dblc_SGS_3 (REAL ∗r, REAL ∗z, void ∗data)
  Block symmetric GS preconditioning (3x3 blocks)
- void fasp_precond_dblc_SGS_3_amg (REAL ∗r, REAL ∗z, void ∗data)
  Block symmetric GS preconditioning (3x3 blocks)
- void fasp_precond_dblc_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

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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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"
```

```
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_precond_dblc_diag_3 (REAL *r,
                               REAL *z,
                               void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata = (precond_data_blc *)data;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr  = &(precdata->r);

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N  = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
#elif WITH_SuperLU
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);
#endif

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_diag_3_amg (REAL *r,
                                    REAL *z,
                                    void *data)
{
    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A     = precdata->Ablc;
    dvector *tempr = &(precdata->r);

    AMG_param *amgparam = precdata->amgparam;
    AMG_data **mgl      = precdata->mgl;

    const INT N0 = A->blocks[0]->row;
    const INT N1 = A->blocks[4]->row;
    const INT N2 = A->blocks[8]->row;
    const INT N  = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;
    r0.row = N0; z0.row = N0; r1.row = N1; z1.row = N1; r2.row = N2; z2.row = N2;
    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); z0.val = z;
    z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A00 block
    mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
    mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);

    fasp_solver_mgcycle(mgl[0], amgparam);
    fasp_darray_cp(N0, mgl[0]->x.val, z0.val);

    // Preconditioning A11 block
    mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
    mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);

    fasp_solver_mgcycle(mgl[1], amgparam);
    fasp_darray_cp(N1, mgl[1]->x.val, z1.val);

    // Preconditioning A22 block
    mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
    mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);

    fasp_solver_mgcycle(mgl[2], amgparam);
    fasp_darray_cp(N2, mgl[2]->x.val, z2.val);

    // restore r
    fasp_darray_cp(N, tempr->val, r);
}

void fasp_precond_dblc_diag_4 (REAL *r,
                               REAL *z,
                               void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata=(precond_data_blc *)data;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr = &(precdata->r);

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N3 = A_diag[3].row;
    const INT N = N0 + N1 + N2 + N3;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
    dvector r0, r1, r2, r3, z0, z1, z2, z3;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;
    r3.row = N3; z3.row = N3;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);
#elif WITH_SuperLU
    dvector r0, r1, r2, r3, z0, z1, z2, z3;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;
    r3.row = N3; z3.row = N3;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);
#endif

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif

    // Preconditioning A33 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[3], &r3, &z3, LU_diag[3], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[3], &r3, &z3, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_lower_3 (REAL *r,
                                REAL *z,
                                void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr = &(precdata->r);

#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
#endif

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N  = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // r1 = r1 - A3*z0
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[3], z0.val, r1.val);

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // r2 = r2 - A6*z0 - A7*z1
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[6], z0.val, r2.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[7], z1.val, r2.val);

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_lower_3_amg (REAL *r,
                                    REAL *z,
                                    void *data)
{
    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dvector *tempr = &(precdata->r);

    AMG_param *amgparam = precdata->amgparam;
    AMG_data **mgl = precdata->mgl;

    const INT N0 = A->blocks[0]->row;
    const INT N1 = A->blocks[4]->row;
    const INT N2 = A->blocks[8]->row;
    const INT N = N0 + N1 + N2;

    INT i;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;
    r0.row = N0; z0.row = N0; r1.row = N1; z1.row = N1; r2.row = N2; z2.row = N2;
    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); z0.val = z;
    z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A00 block
    mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
    mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);
    fasp_darray_cp(N0, mgl[0]->x.val, z0.val);

    // r1 = r1 - A10*z0
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[3], z0.val, r1.val);

    // Preconditioning A11 block
    mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
    mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
    fasp_darray_cp(N1, mgl[1]->x.val, z1.val);

    // r2 = r2 - A20*z0 - A21*z1
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[6], z0.val, r2.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[7], z1.val, r2.val);

    // Preconditioning A22 block
    mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
    mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[2], amgparam);
    fasp_darray_cp(N2, mgl[2]->x.val, z2.val);

    // restore r
    fasp_darray_cp(N, tempr->val, r);
}

void fasp_precond_dblc_lower_4 (REAL *r,
                                REAL *z,
                                void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr = &(precdata->r);

#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
#endif

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N3 = A_diag[3].row;
    const INT N  = N0 + N1 + N2 + N3;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, r3, z0, z1, z2, z3;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;
    r3.row = N3; z3.row = N3;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]); r3.val = &(r[N0+N1+N2]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]); z3.val = &(z[N0+N1+N2]);

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // r1 = r1 - A4*z0
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[4], z0.val, r1.val);

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // r2 = r2 - A8*z0 - A9*z1
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[8], z0.val, r2.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[9], z1.val, r2.val);

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif

    // r3 = r3 - A12*z0 - A13*z1-A14*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[12], z0.val, r3.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[13], z1.val, r3.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[14], z2.val, r3.val);

    // Preconditioning A33 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[3], &r3, &z3, LU_diag[3], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[3], &r3, &z3, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_upper_3 (REAL *r,
                                REAL *z,
                                void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr = &(precdata->r);

#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
#endif

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N  = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif

    // r1 = r1 - A5*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[5], z2.val, r1.val);

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // r0 = r0 - A1*z1 - A2*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[1], z1.val, r0.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[2], z2.val, r0.val);

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_upper_3_amg (REAL *r,
                                    REAL *z,
                                    void *data)
{
    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;

    AMG_param *amgparam = precdata->amgparam;
    AMG_data **mgl = precdata->mgl;

    dvector *tempr = &(precdata->r);

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N  = N0 + N1 + N2;

    INT i;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A22 block
    mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
    mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[2], amgparam);
    fasp_darray_cp(N2, mgl[2]->x.val, z2.val);

    // r1 = r1 - A5*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[5], z2.val, r1.val);

    // Preconditioning A11 block
    mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
    mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
    fasp_darray_cp(N1, mgl[1]->x.val, z1.val);

    // r0 = r0 - A1*z1 - A2*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[1], z1.val, r0.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[2], z2.val, r0.val);

    // Preconditioning A00 block
    mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
    mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);
    fasp_darray_cp(N0, mgl[0]->x.val, z0.val);

    // restore r
    fasp_darray_cp(N, tempr->val, r);
}

void fasp_precond_dblc_SGS_3 (REAL *r,
                              REAL *z,
                              void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;
    dvector *tempr = &(precdata->r);

#if WITH_UMFPACK
    void **LU_diag = precdata->LU_diag;
#endif

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // r1 = r1 - A3*z0
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[3], z0.val, r1.val);

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // r2 = r2 - A6*z0 - A7*z1
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[6], z0.val, r2.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[7], z1.val, r2.val);

    // Preconditioning A22 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[2], &r2, &z2, LU_diag[2], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[2], &r2, &z2, 0);
#endif
```

```

    // r1 = r1 - A5*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[5], z2.val, r1.val);

    // Preconditioning A11 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[1], &r1, &z1, LU_diag[1], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[1], &r1, &z1, 0);
#endif

    // r0 = r0 - A1*z1 - A2*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[1], z1.val, r0.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[2], z2.val, r0.val);

    // Preconditioning A00 block
#if WITH_UMFPACK
    /* use UMFPACK direct solver */
    fasp_umfpack_solve(&A_diag[0], &r0, &z0, LU_diag[0], 0);
#elif WITH_SuperLU
    /* use SuperLU direct solver */
    fasp_solver_superlu(&A_diag[0], &r0, &z0, 0);
#endif

    // restore r
    fasp_darray_cp(N, tempr->val, r);

#endif
}

void fasp_precond_dblc_SGS_3_amg (REAL *r,
                                  REAL *z,
                                  void *data)
{
    precond_data_blc *precdata = (precond_data_blc *)data;
    dBLCmat *A = precdata->Ablc;
    dCSRmat *A_diag = precdata->A_diag;

    AMG_param *amgparam = precdata->amgparam;
    AMG_data **mgl = precdata->mgl;

    INT i;

    dvector *tempr = &(precdata->r);

    const INT N0 = A_diag[0].row;
    const INT N1 = A_diag[1].row;
    const INT N2 = A_diag[2].row;
    const INT N = N0 + N1 + N2;

    // back up r, setup z;
    fasp_darray_cp(N, r, tempr->val);
    fasp_darray_set(N, z, 0.0);

    // prepare
    dvector r0, r1, r2, z0, z1, z2;

    r0.row = N0; z0.row = N0;
    r1.row = N1; z1.row = N1;
    r2.row = N2; z2.row = N2;

    r0.val = r; r1.val = &(r[N0]); r2.val = &(r[N0+N1]);
    z0.val = z; z1.val = &(z[N0]); z2.val = &(z[N0+N1]);

    // Preconditioning A00 block
    mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
    mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);
    fasp_darray_cp(N0, mgl[0]->x.val, z0.val);

    // r1 = r1 - A3*z0
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[3], z0.val, r1.val);

    // Preconditioning A11 block
    mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
    mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
    fasp_darray_cp(N1, mgl[1]->x.val, z1.val);

    // r2 = r2 - A6*z0 - A7*z1
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[6], z0.val, r2.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[7], z1.val, r2.val);

    // Preconditioning A22 block
    mgl[2]->b.row=N2; fasp_darray_cp(N2, r2.val, mgl[2]->b.val);
    mgl[2]->x.row=N2; fasp_dvec_set(N2, &mgl[2]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[2], amgparam);
    fasp_darray_cp(N2, mgl[2]->x.val, z2.val);

    // r1 = r1 - A5*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[5], z2.val, r1.val);

    // Preconditioning A11 block
    mgl[1]->b.row=N1; fasp_darray_cp(N1, r1.val, mgl[1]->b.val);
    mgl[1]->x.row=N1; fasp_dvec_set(N1, &mgl[1]->x,0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[1], amgparam);
    fasp_darray_cp(N1, mgl[1]->x.val, z1.val);

    // r0 = r0 - A1*z1 - A2*z2
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[1], z1.val, r0.val);
    fasp_blas_dcsr_aAxpy(-1.0, A->blocks[2], z2.val, r0.val);

    // Preconditioning A00 block
    mgl[0]->b.row=N0; fasp_darray_cp(N0, r0.val, mgl[0]->b.val);
    mgl[0]->x.row=N0; fasp_dvec_set(N0, &mgl[0]->x, 0.0);

    for(i=0;i<1;++i) fasp_solver_mgcycle(mgl[0], amgparam);
    fasp_darray_cp(N0, mgl[0]->x.val, z0.val);

    // restore r
    fasp_darray_cp(N, tempr->val, r);
}

void fasp_precond_dblc_sweeping (REAL *r,
                                 REAL *z,
                                 void *data)
{
#if WITH_UMFPACK || WITH_SuperLU // Must use direct solvers for this method!

    precond_data_sweeping *precdata = (precond_data_sweeping *)data;

    INT NumLayers = precdata->NumLayers;
    dBLCmat *A = precdata->A;
    dBLCmat *Ai = precdata->Ai;
    dCSRmat *local_A = precdata->local_A;
    ivector *local_index = precdata->local_index;

#if WITH_UMFPACK
    void **local_LU = precdata->local_LU;
#endif

    dvector *r_backup = &(precdata->r);
    REAL *w = precdata->w;

    // local veriables
    INT i,l;
    dvector temp_r;
    dvector temp_e;

    dvector *local_r = (dvector *)fasp_mem_calloc(NumLayers, sizeof(dvector));
    dvector *local_e = (dvector *)fasp_mem_calloc(NumLayers, sizeof(dvector));

    // calculate the size and generate block local_r and local_z
    INT N=0;

    for (l=0;l<NumLayers; l++) {

        local_r[l].row = A->blocks[l*NumLayers+l]->row;
        local_r[l].val = r+N;

        local_e[l].row = A->blocks[l*NumLayers+l]->col;
        local_e[l].val = z+N;

        N = N+A->blocks[l*NumLayers+l]->col;

    }

    temp_r.val = w;
    temp_e.val = w+N;

    // back up r, setup z;
    fasp_darray_cp(N, r, r_backup->val);
    fasp_darray_cp(N, r, z);

    // L^{-1}r
    for (l=0; l<NumLayers-1; l++){

        temp_r.row = local_A[l].row;
        temp_e.row = local_A[l].row;

        fasp_dvec_set(local_A[l].row, &temp_r, 0.0);

        for (i=0; i<local_e[l].row; i++){
            temp_r.val[local_index[l].val[i]] = local_e[l].val[i];
        }

#if WITH_UMFPACK
        /* use UMFPACK direct solver */
        fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
#elif WITH_SuperLU
        /* use SuperLU direct solver */
        fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
#endif

        for (i=0; i<local_r[l].row; i++){
            local_r[l].val[i] = temp_e.val[local_index[l].val[i]];
        }

        fasp_blas_dcsr_aAxpy(-1.0, Ai->blocks[(l+1)*NumLayers+l], local_r[l].val,
                             local_e[l+1].val);

    }

    // D^{-1}L^{-1}r
    for (l=0; l<NumLayers; l++){

        temp_r.row = local_A[l].row;
        temp_e.row = local_A[l].row;

        fasp_dvec_set(local_A[l].row, &temp_r, 0.0);

        for (i=0; i<local_e[l].row; i++){
            temp_r.val[local_index[l].val[i]] = local_e[l].val[i];
        }

#if WITH_UMFPACK
        /* use UMFPACK direct solver */
        fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
#elif WITH_SuperLU
        /* use SuperLU direct solver */
        fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
#endif

        for (i=0; i<local_e[l].row; i++) {
            local_e[l].val[i] = temp_e.val[local_index[l].val[i]];
        }

    }

    // L^{-t}D^{-1}L^{-1}u
    for (l=NumLayers-2; l>=0; l--){

        temp_r.row = local_A[l].row;
        temp_e.row = local_A[l].row;

        fasp_dvec_set(local_A[l].row, &temp_r, 0.0);

        fasp_blas_dcsr_mxv (Ai->blocks[l*NumLayers+l+1], local_e[l+1].val, local_r[l].val);

        for (i=0; i<local_r[l].row; i++){
            temp_r.val[local_index[l].val[i]] = local_r[l].val[i];
        }

#if WITH_UMFPACK
        /* use UMFPACK direct solver */
        fasp_umfpack_solve(&local_A[l], &temp_r, &temp_e, local_LU[l], 0);
#elif WITH_SuperLU
```

```
        /* use SuperLU direct solver */
        fasp_solver_superlu(&local_A[l], &temp_r, &temp_e, 0);
#endif

        for (i=0; i<local_e[l].row; i++){
            local_e[l].val[i] = local_e[l].val[i] - temp_e.val[local_index[l].val[i]];
        }

    }

    // restore r
    fasp_darray_cp(N, r_backup->val, r);

#endif
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.157 PreBSR.c File Reference

Preconditioners for dBSRmat matrices.

```
#include "fasp.h"
#include "fasp_functs.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

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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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.

```c

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreMGUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_precond_dbsr_diag (REAL *r,
                             REAL *z,
                             void *data)
{
    precond_diag_bsr * diag = (precond_diag_bsr *)data;
    const INT nb = diag->nb;

    switch (nb) {

        case 2:
            fasp_precond_dbsr_diag_nc2( r, z, diag);
            break;
        case 3:
            fasp_precond_dbsr_diag_nc3( r, z, diag);
            break;

        case 5:
            fasp_precond_dbsr_diag_nc5( r, z, diag);
            break;

        case 7:
            fasp_precond_dbsr_diag_nc7( r, z, diag);
            break;

        default:
        {
            REAL *diagptr = diag->diag.val;
            const INT nb2 = nb*nb;
            const INT m = diag->diag.row/nb2;
            INT i;

#ifdef _OPENMP
            if (m > OPENMP_HOLDS) {
                INT myid, mybegin, myend;
                INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, mybegin, myend, i)
                for (myid = 0; myid < nthreads; myid++) {
                    fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
                    for (i=mybegin; i<myend; ++i) {
                        fasp_blas_smat_mxv(&(diagptr[i*nb2]),&(r[i*nb]),&(z[i*nb]),nb);
                    }
                }
            }
            else {
#endif
                for (i = 0; i < m; ++i) {
                    fasp_blas_smat_mxv(&(diagptr[i*nb2]),&(r[i*nb]),&(z[i*nb]),nb);
                }
#ifdef _OPENMP
            }
#endif
```

```
            break;
        }
    }
}

void fasp_precond_dbsr_diag_nc2 (REAL *r,
                                 REAL *z,
                                 void *data)
{
    precond_diag_bsr  * diag    = (precond_diag_bsr *)data;
    REAL              * diagptr = diag->diag.val;

    INT i;
    const INT m = diag->diag.row/4;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        INT myid, mybegin, myend;
        INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, mybegin, myend, i)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                fasp_blas_smat_mxv_nc2(&(diagptr[i*4]),&(r[i*2]),&(z[i*2]));
            }
        }
    }
    else {
#endif
        for (i = 0; i < m; ++i) {
            fasp_blas_smat_mxv_nc2(&(diagptr[i*4]),&(r[i*2]),&(z[i*2]));
        }
#ifdef _OPENMP
    }
#endif
}

void fasp_precond_dbsr_diag_nc3 (REAL *r,
                                 REAL *z,
                                 void *data)
{
    precond_diag_bsr  * diag    = (precond_diag_bsr *)data;
    REAL              * diagptr = diag->diag.val;

    const INT m = diag->diag.row/9;
    INT i;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        INT myid, mybegin, myend;
        INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, mybegin, myend, i)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                fasp_blas_smat_mxv_nc3(&(diagptr[i*9]),&(r[i*3]),&(z[i*3]));
            }
        }
    }
    else {
#endif
        for (i = 0; i < m; ++i) {
            fasp_blas_smat_mxv_nc3(&(diagptr[i*9]),&(r[i*3]),&(z[i*3]));
        }
#ifdef _OPENMP
    }
#endif
}

void fasp_precond_dbsr_diag_nc5 (REAL *r,
                                 REAL *z,
                                 void *data)
{
    precond_diag_bsr  * diag    = (precond_diag_bsr *)data;
    REAL              * diagptr = diag->diag.val;

    const INT m = diag->diag.row/25;
    INT i;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        INT myid, mybegin, myend;
        INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, mybegin, myend, i)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                fasp_blas_smat_mxv_nc5(&(diagptr[i*25]),&(r[i*5]),&(z[i*5]));
            }
        }
    }
    else {
#endif
        for (i = 0; i < m; ++i) {
            fasp_blas_smat_mxv_nc5(&(diagptr[i*25]),&(r[i*5]),&(z[i*5]));
        }
#ifdef _OPENMP
    }
#endif
}

void fasp_precond_dbsr_diag_nc7 (REAL *r,
                                 REAL *z,
                                 void *data)
{
    precond_diag_bsr  * diag    = (precond_diag_bsr *)data;
    REAL              * diagptr = diag->diag.val;

    const INT m = diag->diag.row/49;
    INT i;

#ifdef _OPENMP
    if (m > OPENMP_HOLDS) {
        INT myid, mybegin, myend;
        INT nthreads = fasp_get_num_threads();
#pragma omp parallel for private(myid, mybegin, myend, i)
        for (myid = 0; myid < nthreads; myid++) {
            fasp_get_start_end(myid, nthreads, m, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                fasp_blas_smat_mxv_nc7(&(diagptr[i*49]),&(r[i*7]),&(z[i*7]));
            }
        }
    }
    else {
#endif
        for (i = 0; i < m; ++i) {
            fasp_blas_smat_mxv_nc7(&(diagptr[i*49]),&(r[i*7]),&(z[i*7]));
        }
#ifdef _OPENMP
    }
#endif
}

void fasp_precond_dbsr_ilu (REAL *r,
                            REAL *z,
                            void *data)
{
    const ILU_data  *iludata=(ILU_data *)data;
    const INT        m=iludata->row, mm1=m-1, mm2=m-2, memneed=2*m;
    const INT        nb=iludata->nb, nb2=nb*nb, size=m*nb;

    INT       *ijlu=iludata->ijlu;
    REAL      *lu=iludata->luval;

    INT        ib, ibstart,ibstart1;
    INT        i, j, jj, begin_row, end_row;
    REAL      *zz, *zr, *mult;

    if (iludata->nwork<memneed) {
        printf("### ERROR: Need %d memory, only %d available!\n",
               memneed, iludata->nwork);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    zz   = iludata->work;
    zr   = zz + size;
    mult = zr + size;

    memcpy(zr, r, size*sizeof(REAL));

    switch (nb) {

        case 1:

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            zz[0]=zr[0];
            for (i=1;i<=mm1;++i) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                for (j=begin_row;j<=end_row;++j) {
                    jj=ijlu[j];
                    if (jj<i) zr[i]-=lu[j]*zz[jj];
                    else break;
                }
                zz[i]=zr[i];
            }

            // backward sweep:  solve upper matrix equation U*z=zz
            z[mm1]=zz[mm1]*lu[mm1];
            for (i=mm2;i>=0;i--) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                for (j=end_row;j>=begin_row;j--) {
                    jj=ijlu[j];
                    if (jj>i) zz[i]-=lu[j]*z[jj];
                    else break;
                }
                z[i]=zz[i]*lu[i];
            }

            break; //end (if nb==1)

        case 3:

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            zz[0] = zr[0];
            zz[1] = zr[1];
            zz[2] = zr[2];

            for (i=1;i<=mm1;++i) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb;
                for (j=begin_row;j<=end_row;++j) {
                    jj=ijlu[j];
                    if (jj<i)
                    {
                        fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                    }
                    else break;
                }

                zz[ibstart]   = zr[ibstart];
                zz[ibstart+1] = zr[ibstart+1];
                zz[ibstart+2] = zr[ibstart+2];
            }

            // backward sweep:  solve upper matrix equation U*z=zz
            ibstart=mm1*nb2;
            ibstart1=mm1*nb;
            fasp_blas_smat_mxv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

            for (i=mm2;i>=0;i--) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb2;
                ibstart1=i*nb;
                for (j=end_row;j>=begin_row;j--) {
                    jj=ijlu[j];
                    if (jj>i) {
                        fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(z[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                    }

                    else break;
                }

                fasp_blas_smat_mxv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

            }

            break; // end (if nb=3)

        case 5:

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp(nb,&(zr[0]),&(zz[0]));

            for (i=1;i<=mm1;++i) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb;
                for (j=begin_row;j<=end_row;++j) {
                    jj=ijlu[j];
                    if (jj<i) {
                        fasp_blas_smat_mxv_nc5(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                    }
                    else break;
                }

                fasp_darray_cp(nb,&(zr[ibstart]),&(zz[ibstart]));
            }

            // backward sweep:  solve upper matrix equation U*z=zz
            ibstart=mm1*nb2;
            ibstart1=mm1*nb;
            fasp_blas_smat_mxv_nc5(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

            for (i=mm2;i>=0;i--) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb2;
                ibstart1=i*nb;
                for (j=end_row;j>=begin_row;j--) {
                    jj=ijlu[j];
                    if (jj>i) {
                        fasp_blas_smat_mxv_nc5(&(lu[j*nb2]),&(z[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                    }

                    else break;
                }

                fasp_blas_smat_mxv_nc5(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

            }

            break; //end (if nb==5)

        case 7:

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp(nb,&(zr[0]),&(zz[0]));

            for (i=1;i<=mm1;++i) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb;
                for (j=begin_row;j<=end_row;++j) {
                    jj=ijlu[j];
                    if (jj<i) {
                        fasp_blas_smat_mxv_nc7(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                    }
                    else break;
                }

                fasp_darray_cp(nb,&(zr[ibstart]),&(zz[ibstart]));
            }

            // backward sweep:  solve upper matrix equation U*z=zz
            ibstart=mm1*nb2;
            ibstart1=mm1*nb;
            fasp_blas_smat_mxv_nc7(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

            for (i=mm2;i>=0;i--) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb2;
                ibstart1=i*nb;
                for (j=end_row;j>=begin_row;j--) {
                    jj=ijlu[j];
                    if (jj>i) {
                        fasp_blas_smat_mxv_nc7(&(lu[j*nb2]),&(z[jj*nb]),mult);
                        for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                    }

                    else break;
                }

                fasp_blas_smat_mxv_nc7(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));
            }

            break; //end (if nb==7)

        default:

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp(nb,&(zr[0]),&(zz[0]));

            for (i=1;i<=mm1;++i) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb;
                for (j=begin_row;j<=end_row;++j) {
                    jj=ijlu[j];
                    if (jj<i) {
                        fasp_blas_smat_mxv(&(lu[j*nb2]),&(zz[jj*nb]),mult,nb);
                        for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                    }
                    else break;
                }

                fasp_darray_cp(nb,&(zr[ibstart]),&(zz[ibstart]));
            }

            // backward sweep:  solve upper matrix equation U*z=zz
            ibstart=mm1*nb2;
            ibstart1=mm1*nb;
            fasp_blas_smat_mxv(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]),nb);

            for (i=mm2;i>=0;i--) {
                begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                ibstart=i*nb2;
                ibstart1=i*nb;
                for (j=end_row;j>=begin_row;j--) {
                    jj=ijlu[j];
                    if (jj>i) {
                        fasp_blas_smat_mxv(&(lu[j*nb2]),&(z[jj*nb]),mult,nb);
                        for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                    }

                    else break;
                }

                fasp_blas_smat_mxv(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]),nb);
            }

            break; // end everything else
    }

    return;
}

void fasp_precond_dbsr_ilu_mc_omp (REAL *r,
                                   REAL *z,
                                   void *data)
{
#ifdef _OPENMP
    const ILU_data  *iludata=(ILU_data *)data;
    const INT        m=iludata->row, memneed=2*m;
    const INT        nb=iludata->nb, nb2=nb*nb, size=m*nb;

    INT        *ijlu=iludata->ijlu;
    REAL       *lu=iludata->luval;
    INT        ncolors = iludata->nlevL;
    INT        *ic = iludata->ilevL;

    INT         ib, ibstart,ibstart1;
    INT         i, j, jj, k, begin_row, end_row;
    REAL        *zz, *zr, *mult;

    if (iludata->nwork<memneed) {
        printf("### ERROR: Need %d memory, only %d available!\n",
               memneed, iludata->nwork);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    zz   = iludata->work;
    zr   = zz + size;

    memcpy(zr, r, size*sizeof(REAL));

    switch (nb) {

        case 1:
            // forward sweep:  solve unit lower matrix equation L*zz=zr
            for (k=0; k<ncolors; ++k) {
#pragma omp parallel for private(i,begin_row,end_row,j,jj)
                for (i=ic[k];i<ic[k+1];++i) {
                    begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                    for (j=begin_row;j<=end_row;++j) {
                        jj=ijlu[j];
                        if (jj<i) zr[i]-=lu[j]*zz[jj];
                        else break;
                    }
                    zz[i]=zr[i];
                }
            }
            // backward sweep:  solve upper matrix equation U*z=zz
            for (k=ncolors-1; k>=0; k--) {
#pragma omp parallel for private(i,begin_row,end_row,j,jj)
                for (i=ic[k+1]-1;i>=ic[k];i--) {
                    begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                    for (j=end_row;j>=begin_row;j--) {
                        jj=ijlu[j];
                        if (jj>i) zz[i]-=lu[j]*z[jj];
                        else break;
                    }
                    z[i]=zz[i]*lu[i];
                }
            }

            break; //end (if nb==1)

        case 2:

            for (k=0; k<ncolors; ++k) {
#pragma omp parallel private(i,begin_row,end_row,ibstart,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (i=ic[k];i<ic[k+1];++i) {
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb;
                        for (j=begin_row;j<=end_row;++j) {
                            jj=ijlu[j];
                            if (jj<i)
                            {
                                fasp_blas_smat_mxv_nc2(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                            }
                            else break;
                        }

                        zz[ibstart]   = zr[ibstart];
                        zz[ibstart+1] = zr[ibstart+1];
                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            for (k=ncolors-1; k>=0; k--) {
#pragma omp parallel private(i,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (i=ic[k+1]-1;i>=ic[k];i--) {
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb2;
                        ibstart1=i*nb;
                        for (j=end_row;j>=begin_row;j--) {
                            jj=ijlu[j];
                            if (jj>i) {
                                fasp_blas_smat_mxv_nc2(&(lu[j*nb2]),&(z[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                            }

                            else break;
                        }

                        fasp_blas_smat_mxv_nc2(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            break; // end (if nb=2)
        case 3:

            for (k=0; k<ncolors; ++k) {
#pragma omp parallel private(i,begin_row,end_row,ibstart,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (i=ic[k];i<ic[k+1];++i) {
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb;
                        for (j=begin_row;j<=end_row;++j) {
                            jj=ijlu[j];
                            if (jj<i)
                            {
                                fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                            }
                            else break;
                        }

                        zz[ibstart]   = zr[ibstart];
                        zz[ibstart+1] = zr[ibstart+1];
                        zz[ibstart+2] = zr[ibstart+2];
                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            for (k=ncolors-1; k>=0; k--) {
#pragma omp parallel private(i,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (i=ic[k+1]-1;i>=ic[k];i--) {
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb2;
                        ibstart1=i*nb;
                        for (j=end_row;j>=begin_row;j--) {
                            jj=ijlu[j];
                            if (jj>i) {
                                fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(z[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                            }

                            else break;
                        }

                        fasp_blas_smat_mxv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            break; // end (if nb=3)

        default:
        {
            if (nb > 3) {
                printf("### ERROR: Multi-thread Parallel ILU for %d components \
has not yet been implemented!!!", nb);
                fasp_chkerr(ERROR_UNKNOWN, __FUNCTION__);
            }
            break;
        }
    }

    return;
#endif
}

```

```
void fasp_precond_dbsr_ilu_ls_omp (REAL *r,
                                   REAL *z,
                                   void *data)
{
#ifdef _OPENMP
    const ILU_data  *iludata=(ILU_data *)data;
    const INT        m=iludata->row, memneed=2*m;
    const INT        nb=iludata->nb, nb2=nb*nb, size=m*nb;

    INT        *ijlu=iludata->ijlu;
    REAL       *lu=iludata->luval;
    INT        nlevL = iludata->nlevL;
    INT        *ilevL = iludata->ilevL;
    INT        *jlevL = iludata->jlevL;
    INT        nlevU = iludata->nlevU;
    INT        *ilevU = iludata->ilevU;
    INT        *jlevU = iludata->jlevU;

    INT         ib, ibstart,ibstart1;
    INT         i, ii, j, jj, k, begin_row, end_row;
    REAL       *zz, *zr, *mult;

    if (iludata->nwork<memneed) {
        printf("### ERROR: Need %d memory, only %d available!\n",
               memneed, iludata->nwork);
        fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
    }

    zz   = iludata->work;
    zr   = zz + size;
    //mult = zr + size;

    memcpy(zr, r, size*sizeof(REAL));

    switch (nb) {

        case 1:
            // forward sweep:  solve unit lower matrix equation L*zz=zr
            for (k=0; k<nlevL; ++k) {
#pragma omp parallel for private(i,ii,begin_row,end_row,j,jj)
                for (ii=ilevL[k];ii<ilevL[k+1];++ii) {
                    i = jlevL[ii];
                    begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                    for (j=begin_row;j<=end_row;++j) {
                        jj=ijlu[j];
                        if (jj<i) zr[i]-=lu[j]*zz[jj];
                        else break;
                    }
                    zz[i]=zr[i];
                }
            }
            // backward sweep:  solve upper matrix equation U*z=zz
            for (k=0; k<nlevU; k++) {
#pragma omp parallel for private(i,ii,begin_row,end_row,j,jj)
                for (ii=ilevU[k+1]-1;ii>=ilevU[k];ii--) {
                    i = jlevU[ii];
                    begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                    for (j=end_row;j>=begin_row;j--) {
                        jj=ijlu[j];
                        if (jj>i) zz[i]-=lu[j]*z[jj];
                        else break;
                    }
                    z[i]=zz[i]*lu[i];
                }
            }

            break; //end (if nb==1)

        case 2:

            for (k=0; k<nlevL; ++k) {
#pragma omp parallel private(i,ii,begin_row,end_row,ibstart,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (ii=ilevL[k];ii<ilevL[k+1];++ii) {
                        i = jlevL[ii];
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb;
                        for (j=begin_row;j<=end_row;++j) {
                            jj=ijlu[j];
                            if (jj<i)
                            {
                                fasp_blas_smat_mxv_nc2(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                            }
                            else break;
                        }

                        zz[ibstart]   = zr[ibstart];
                        zz[ibstart+1] = zr[ibstart+1];
                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            for (k=0; k<nlevU; k++) {
#pragma omp parallel private(i,ii,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (ii=ilevU[k+1]-1;ii>=ilevU[k];ii--) {
                        i = jlevU[ii];
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb2;
                        ibstart1=i*nb;
                        for (j=end_row;j>=begin_row;j--) {
                            jj=ijlu[j];
                            if (jj>i) {
                                fasp_blas_smat_mxv_nc2(&(lu[j*nb2]),&(z[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                            }

                            else break;
                        }

                        fasp_blas_smat_mxv_nc2(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            break; // end (if nb=2)
        case 3:

            for (k=0; k<nlevL; ++k) {
#pragma omp parallel private(i,ii,begin_row,end_row,ibstart,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (ii=ilevL[k];ii<ilevL[k+1];++ii) {
                        i = jlevL[ii];
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb;
                        for (j=begin_row;j<=end_row;++j) {
                            jj=ijlu[j];
                            if (jj<i)
                            {
                                fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(zz[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zr[ibstart+ib]-=mult[ib];
                            }
                            else break;
                        }

                        zz[ibstart]   = zr[ibstart];
                        zz[ibstart+1] = zr[ibstart+1];
                        zz[ibstart+2] = zr[ibstart+2];
                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            for (k=0; k<nlevU; k++) {
#pragma omp parallel private(i,ii,begin_row,end_row,ibstart,ibstart1,j,jj,ib,mult)
                {
                    mult = (REAL*)fasp_mem_calloc(nb,sizeof(REAL));
#pragma omp for
                    for (ii=ilevU[k+1]-1;ii>=ilevU[k];ii--) {
                        i = jlevU[ii];
                        begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
                        ibstart=i*nb2;
                        ibstart1=i*nb;
                        for (j=end_row;j>=begin_row;j--) {
                            jj=ijlu[j];
                            if (jj>i) {
                                fasp_blas_smat_mxv_nc3(&(lu[j*nb2]),&(z[jj*nb]),mult);
                                for (ib=0;ib<nb;++ib) zz[ibstart1+ib]-=mult[ib];
                            }

                            else break;
                        }

                        fasp_blas_smat_mxv_nc3(&(lu[ibstart]),&(zz[ibstart1]),&(z[ibstart1]));

                    }

                    fasp_mem_free(mult); mult = NULL;
                }
            }

            break; // end (if nb=3)

        default:
        {
            if (nb > 3) {
                printf("### ERROR: Multi-thread Parallel ILU for %d components \
has not yet been implemented!!!", nb);
                fasp_chkerr(ERROR_UNKNOWN, __FUNCTION__);
            }
            break;
        }
    }

    return;
#endif
}

void fasp_precond_dbsr_amg (REAL *r,
                            REAL *z,
                            void *data)
{
    precond_data_bsr *predata=(precond_data_bsr *)data;
    const INT row=predata->mgl_data[0].A.ROW;
    const INT nb = predata->mgl_data[0].A.nb;
    const INT maxit=predata->maxit;
    const INT m = row*nb;

    INT i;

    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    amgparam.cycle_type = predata->cycle_type;
    amgparam.smoother   = predata->smoother;
    amgparam.smooth_order = predata->smooth_order;
    amgparam.presmooth_iter  = predata->presmooth_iter;
    amgparam.postsmooth_iter = predata->postsmooth_iter;
    amgparam.relaxation = predata->relaxation;
    amgparam.coarse_scaling = predata->coarse_scaling;
    amgparam.tentative_smooth = predata->tentative_smooth;
    amgparam.ILU_levels = predata->mgl_data->ILU_levels;

    AMG_data_bsr *mgl = predata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_mgcycle_bsr(mgl,&amgparam);

    fasp_darray_cp(m,mgl->x.val,z);
}

void fasp_precond_dbsr_amg_nk (REAL *r,
                               REAL *z,
                               void *data)
{
    precond_data_bsr *predata=(precond_data_bsr *)data;
    const INT row=predata->mgl_data[0].A.ROW;
    const INT nb = predata->mgl_data[0].A.nb;
    const INT maxit=predata->maxit;
    const INT m = row*nb;

    INT i;

    dCSRmat *A_nk = predata->A_nk;
    dCSRmat *P_nk = predata->P_nk;
    dCSRmat *R_nk = predata->R_nk;

    fasp_darray_set(m, z, 0.0);

    // local variables
    dvector r_nk, z_nk;
    fasp_dvec_alloc(A_nk->row, &r_nk);
    fasp_dvec_alloc(A_nk->row, &z_nk);

    //----------------------
    // extra kernel solve
    //----------------------
    // r_nk = R_nk*r
    fasp_blas_dcsr_mxv(R_nk, r, r_nk.val);

    // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
    fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
#else
    fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

    // z = z + P_nk*z_nk;
    fasp_blas_dcsr_aAxpy(1.0, P_nk, z_nk.val, z);

    //----------------------
    // AMG solve
    //----------------------
    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    amgparam.cycle_type = predata->cycle_type;
    amgparam.smoother   = predata->smoother;
    amgparam.smooth_order = predata->smooth_order;
    amgparam.presmooth_iter  = predata->presmooth_iter;
    amgparam.postsmooth_iter = predata->postsmooth_iter;
    amgparam.relaxation = predata->relaxation;
    amgparam.coarse_scaling = predata->coarse_scaling;
    amgparam.tentative_smooth = predata->tentative_smooth;
    amgparam.ILU_levels = predata->mgl_data->ILU_levels;

    AMG_data_bsr *mgl = predata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; //fasp_dvec_set(m,&mgl->x,0.0);
    fasp_darray_cp(m, z, mgl->x.val);

    for ( i=maxit; i--; ) fasp_solver_mgcycle_bsr(mgl,&amgparam);

    fasp_darray_cp(m,mgl->x.val,z);

    //----------------------
    // extra kernel solve
    //----------------------
    // r = r - A*z
    fasp_blas_dbsr_aAxpy(-1.0, &(predata->mgl_data[0].A), z, mgl->b.val);

    // r_nk = R_nk*r
    fasp_blas_dcsr_mxv(R_nk, mgl->b.val, r_nk.val);

    // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
    fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
#else
    fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

    // z = z + P_nk*z_nk;
    fasp_blas_dcsr_aAxpy(1.0, P_nk, z_nk.val, z);
}

void fasp_precond_dbsr_namli (REAL *r,
                              REAL *z,
                              void *data)
{
    precond_data_bsr *pcdata=(precond_data_bsr *)data;
    const INT row=pcdata->mgl_data[0].A.ROW;
    const INT nb=pcdata->mgl_data[0].A.nb;
    const INT maxit=pcdata->maxit;
    const SHORT num_levels=pcdata->max_levels;
    const INT m=row*nb;

    INT i;

    AMG_param amgparam;
    fasp_param_amg_init(&amgparam);
    fasp_param_precbsr_to_amg(&amgparam,pcdata);

    AMG_data_bsr *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_namli_bsr(mgl,&amgparam,0, num_levels);

    fasp_darray_cp(m,mgl->x.val,z);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.159 PreCSR.c File Reference

Preconditioners for dCSRmat matrices.

```
#include "fasp.h"
#include "fasp_functs.h"
#include "PreMGUtil.inl"
```

Functions

- precond * 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=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, BlaILUSetupCSR.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

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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning

Parameters

z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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 precond_type,
            AMG_param * amgparam,
            ILU_param * iluparam,
            dCSRmat * A )
```

Setup preconditioner interface for iterative methods.

Parameters

precond_type	Preconditioner type
amgparam	Pointer to AMG parameters
iluparam	Pointer to ILU parameters
A	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

r	Pointer to residual
z	Pointer to preconditioned residual
data	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.

```

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreMGUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

precond *fasp_precond_setup (const SHORT   precond_type,
                             AMG_param    *amgparam,
                             ILU_param    *iluparam,
                             dCSRmat      *A)
{
    precond           *pc = NULL;
    AMG_data         *mgl = NULL;
    precond_data  *pcdata = NULL;
    ILU_data         *ILU = NULL;
    dvector         *diag = NULL;

    INT           max_levels, nnz, m, n;

    switch (precond_type) {

    case PREC_AMG:  // AMG preconditioner

        pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
        max_levels = amgparam->max_levels;
        nnz = A->nnz; m = A->row; n = A->col;

        // initialize A, b, x for mgl[0]
        mgl=fasp_amg_data_create(max_levels);
        mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
        mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);

        // setup preconditioner
        switch (amgparam->AMG_type) {
        case SA_AMG:  // Smoothed Aggregation AMG
            fasp_amg_setup_sa(mgl, amgparam); break;
        case UA_AMG:  // Unsmoothed Aggregation AMG
            fasp_amg_setup_ua(mgl, amgparam); break;
        default:  // Classical AMG
            fasp_amg_setup_rs(mgl, amgparam); break;
        }

        pcdata = (precond_data *)fasp_mem_calloc(1, sizeof(precond_data));
        fasp_param_amg_to_prec(pcdata, amgparam);
        pcdata->max_levels = mgl[0].num_levels;
        pcdata->mgl_data = mgl;

        pc->data = pcdata;

        switch (amgparam->cycle_type) {
        case AMLI_CYCLE:  // AMLI cycle
            pc->fct = fasp_precond_amli; break;
        case NL_AMLI_CYCLE:  // Nonlinear AMLI
            pc->fct = fasp_precond_namli; break;
        default:  // V,W-cycles or hybrid cycles
            pc->fct = fasp_precond_amg; break;
        }

        break;

    case PREC_FMG:  // FMG preconditioner

        pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
        max_levels = amgparam->max_levels;
        nnz = A->nnz; m = A->row; n = A->col;

        // initialize A, b, x for mgl[0]
        mgl=fasp_amg_data_create(max_levels);
        mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
        mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);

        // setup preconditioner
        switch (amgparam->AMG_type) {
        case SA_AMG:  // Smoothed Aggregation AMG
            fasp_amg_setup_sa(mgl, amgparam); break;
        case UA_AMG:  // Unsmoothed Aggregation AMG
            fasp_amg_setup_ua(mgl, amgparam); break;
        default:  // Classical AMG
            fasp_amg_setup_rs(mgl, amgparam); break;
        }

        pcdata = (precond_data *)fasp_mem_calloc(1, sizeof(precond_data));
        fasp_param_amg_to_prec(pcdata, amgparam);
        pcdata->max_levels = mgl[0].num_levels;
        pcdata->mgl_data = mgl;

        pc->data = pcdata; pc->fct = fasp_precond_famg;

        break;

    case PREC_ILU:  // ILU preconditioner

        pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
        ILU = (ILU_data *)fasp_mem_calloc(1, sizeof(ILU_data));
        fasp_ilu_dcsr_setup(A, ILU, iluparam);
        pc->data = ILU;
        pc->fct = fasp_precond_ilu;

        break;

    case PREC_DIAG:  // Diagonal preconditioner

        pc = (precond *)fasp_mem_calloc(1, sizeof(precond));
        diag = (dvector *)fasp_mem_calloc(1, sizeof(dvector));
        fasp_dcsr_getdiag(0, A, diag);

        pc->data = diag;
        pc->fct  = fasp_precond_diag;

        break;

    default:  // No preconditioner

        break;

    }

    return pc;
}

void fasp_precond_diag (REAL *r,
                        REAL *z,
                        void *data)
{
    dvector *diag=(dvector *)data;
    REAL *diagptr=diag->val;
    INT i, m=diag->row;

    memcpy(z,r,m*sizeof(REAL));
    for (i=0;i<m;++i) {
        if (ABS(diag->val[i])>SMALLREAL) z[i]/=diagptr[i];
    }
}

void fasp_precond_ilu (REAL *r,
                       REAL *z,
                       void *data)
{
    ILU_data *iludata=(ILU_data *)data;
    const INT m=iludata->row, mm1=m-1, memneed=2*m;
    REAL *zz, *zr;

    if (iludata->nwork<memneed) goto MEMERR; // check this outside this subroutine!!

    zz = iludata->work;
    zr = iludata->work+m;
    fasp_darray_cp(m, r, zr);

    {
        INT i, j, jj, begin_row, end_row, mm2=m-2;
        INT *ijlu=iludata->ijlu;
        REAL *lu=iludata->luval;

        // forward sweep:  solve unit lower matrix equation L*zz=zr
        zz[0]=zr[0];

        for (i=1;i<=mm1;++i) {
            begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
            for (j=begin_row;j<=end_row;++j) {
                jj=ijlu[j];
                if (jj<i) zr[i]-=lu[j]*zz[jj];
                else break;
            }
            zz[i]=zr[i];
        }

        // backward sweep:  solve upper matrix equation U*z=zz
        z[mm1]=zz[mm1]*lu[mm1];
        for (i=mm2;i>=0;i--) {
            begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
            for (j=end_row;j>=begin_row;j--) {
                jj=ijlu[j];
                if (jj>i) zz[i]-=lu[j]*z[jj];
                else break;
            }
            z[i]=zz[i]*lu[i];
        }
    }

    return;

MEMERR:
    printf("### ERROR: Need %d memory, only %d available!\n",
           memneed, iludata->nwork);
    fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
}

void fasp_precond_ilu_forward (REAL *r,
                               REAL *z,
                                void *data)
{
    ILU_data *iludata=(ILU_data *)data;
    const INT m=iludata->row, mm1=m-1, memneed=2*m;
    REAL *zz, *zr;

    if (iludata->nwork<memneed) goto MEMERR;

    zz = iludata->work;
    zr = iludata->work+m;
    fasp_darray_cp(m, r, zr);

    {
        INT i, j, jj, begin_row, end_row;
        INT *ijlu=iludata->ijlu;
        REAL *lu=iludata->luval;

        // forward sweep:  solve unit lower matrix equation L*z=r
        zz[0]=zr[0];
        for (i=1;i<=mm1;++i) {
            begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
            for (j=begin_row;j<=end_row;++j) {
                jj=ijlu[j];
                if (jj<i) zr[i]-=lu[j]*zz[jj];
                else break;
            }
            zz[i]=zr[i];
        }
    }

    fasp_darray_cp(m, zz, z);

    return;

MEMERR:
    printf("### ERROR: Need %d memory, only %d available!",
           memneed, iludata->nwork);
    fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
}

void fasp_precond_ilu_backward (REAL *r,
                                REAL *z,
                                void *data)
{
    ILU_data *iludata=(ILU_data *)data;
    const INT m=iludata->row, mm1=m-1, memneed=2*m;
    REAL *zz;

    if (iludata->nwork<memneed) goto MEMERR;

    zz = iludata->work;
    fasp_darray_cp(m, r, zz);

    {
        INT i, j, jj, begin_row, end_row, mm2=m-2;
        INT *ijlu=iludata->ijlu;
        REAL *lu=iludata->luval;

        // backward sweep:  solve upper matrix equation U*z=zz
        z[mm1]=zz[mm1]*lu[mm1];
        for (i=mm2;i>=0;i--) {
            begin_row=ijlu[i]; end_row=ijlu[i+1]-1;
            for (j=end_row;j>=begin_row;j--) {
                jj=ijlu[j];
                if (jj>i) zz[i]-=lu[j]*z[jj];
                else break;
            }
            z[i]=zz[i]*lu[i];
        }

    }

    return;

MEMERR:
    printf("### ERROR: Need %d memory, only %d available!",
           memneed, iludata->nwork);
    fasp_chkerr(ERROR_ALLOC_MEM, __FUNCTION__);
}

void fasp_precond_swz (REAL *r,
                       REAL *z,
                       void *data)
{
    SWZ_data  * swzdata  = (SWZ_data *)data;
    SWZ_param * swzparam = swzdata->swzparam;
    const INT   swztype  = swzdata->SWZ_type;
    const INT   n        = swzdata->A.row;

    dvector x, b;

    fasp_dvec_alloc(n, &x);
    fasp_dvec_alloc(n, &b);
    fasp_darray_cp(n, r, b.val);

    fasp_dvec_set(n, &x, 0);

    switch (swztype) {
        case SCHWARZ_BACKWARD:
            fasp_dcsr_swz_backward(swzdata, swzparam, &x, &b);
            break;
        case SCHWARZ_SYMMETRIC:
            fasp_dcsr_swz_forward(swzdata, swzparam, &x, &b);
            fasp_dcsr_swz_backward(swzdata, swzparam, &x, &b);
            break;
        default:
            fasp_dcsr_swz_forward(swzdata, swzparam, &x, &b);
            break;
    }

    fasp_darray_cp(n, x.val, z);
}

void fasp_precond_amg (REAL *r,
                       REAL *z,
                       void *data)
{
    precond_data *pcdata=(precond_data *)data;
    const INT m=pcdata->mgl_data[0].A.row;
    const INT maxit=pcdata->maxit;
    INT i;

    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    fasp_param_prec_to_amg(&amgparam,pcdata);

    AMG_data *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_mgcycle(mgl,&amgparam);

    fasp_darray_cp(m,mgl->x.val,z);
}

void fasp_precond_famg (REAL *r,
                        REAL *z,
                        void *data)
{
    precond_data *pcdata=(precond_data *)data;
    const INT m=pcdata->mgl_data[0].A.row;
    const INT maxit=pcdata->maxit;
    INT i;

    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    fasp_param_prec_to_amg(&amgparam,pcdata);

    AMG_data *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_fmgcycle(mgl,&amgparam);

    fasp_darray_cp(m,mgl->x.val,z);
}

void fasp_precond_amli (REAL *r,
                        REAL *z,
                        void *data)
{
    precond_data *pcdata=(precond_data *)data;
    const INT m=pcdata->mgl_data[0].A.row;
    const INT maxit=pcdata->maxit;
    INT i;

    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    fasp_param_prec_to_amg(&amgparam,pcdata);

    AMG_data *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_amli(mgl,&amgparam,0);

    fasp_darray_cp(m,mgl->x.val,z);
}

void fasp_precond_namli (REAL *r,
                         REAL *z,
                         void *data)
{
    precond_data *pcdata=(precond_data *)data;
    const INT m=pcdata->mgl_data[0].A.row;
    const INT maxit=pcdata->maxit;
    const SHORT num_levels = pcdata->max_levels;
    INT i;

    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    fasp_param_prec_to_amg(&amgparam,pcdata);

    AMG_data *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; fasp_dvec_set(m,&mgl->x,0.0);

    for ( i=maxit; i--; ) fasp_solver_namli(mgl, &amgparam, 0, num_levels);
    fasp_darray_cp(m,mgl->x.val,z);
}

void fasp_precond_amg_nk (REAL *r,
                          REAL *z,
                          void *data)
{
    precond_data *pcdata=(precond_data *)data;
    const INT m=pcdata->mgl_data[0].A.row;
    const INT maxit=pcdata->maxit;
    INT i;

    dCSRmat *A_nk = pcdata->A_nk;
    dCSRmat *P_nk = pcdata->P_nk;
    dCSRmat *R_nk = pcdata->R_nk;

    fasp_darray_set(m, z, 0.0);

    // local variables
    dvector r_nk, z_nk;
    fasp_dvec_alloc(A_nk->row, &r_nk);
    fasp_dvec_alloc(A_nk->row, &z_nk);

    //----------------------
    // extra kernel solve
    //----------------------
    // r_nk = R_nk*r
    fasp_blas_dcsr_mxv(R_nk, r, r_nk.val);

    // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
    fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
#else
    fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

    // z = z + P_nk*z_nk;
    fasp_blas_dcsr_aAxpy(1.0, P_nk, z_nk.val, z);

    //----------------------
    // AMG solve
    //----------------------
    AMG_param amgparam; fasp_param_amg_init(&amgparam);
    fasp_param_prec_to_amg(&amgparam,pcdata);

    AMG_data *mgl = pcdata->mgl_data;
    mgl->b.row=m; fasp_darray_cp(m,r,mgl->b.val); // residual is an input
    mgl->x.row=m; //fasp_dvec_set(m,&mgl->x,0.0);
    fasp_darray_cp(m, z, mgl->x.val);

    for ( i=maxit; i--; ) fasp_solver_mgcycle(mgl,&amgparam);

    fasp_darray_cp(m,mgl->x.val,z);

    //----------------------
    // extra kernel solve
    //----------------------
    // r = r - A*z
    fasp_blas_dcsr_aAxpy(-1.0, &(pcdata->mgl_data[0].A), z, mgl->b.val);

    // r_nk = R_nk*r
    fasp_blas_dcsr_mxv(R_nk, mgl->b.val, r_nk.val);

    // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
    fasp_solver_umfpack(A_nk, &r_nk, &z_nk, 0);
#else
    fasp_coarse_itsolver(A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

    // z = z + P_nk*z_nk;
    fasp_blas_dcsr_aAxpy(1.0, P_nk, z_nk.val, z);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.161 PreDataInit.c File Reference

Initialize important data structures.

```
#include "fasp.h"
#include "fasp_functs.h"
```

Functions

- void fasp_precond_data_init (precond_data *pcdata)

  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

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

max_levels	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

mgl	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

max_levels	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

mgl	Pointer to the AMG_data
param	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

iwk	Size of the index array
nwork	Size of the work array
iludata	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

iludata	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 (
             precond_data * pcdata )
```

Initialize precond_data.

Parameters

pcdata	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

swzdata	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 *pcdata)
00034 {
00035     pcdata->AMG_type             = CLASSIC_AMG;
00036     pcdata->print_level          = PRINT_NONE;
00037     pcdata->maxit                = 500;
00038     pcdata->max_levels           = 20;
00039     pcdata->tol                  = 1e-8;
00040     pcdata->cycle_type           = V_CYCLE;
```

```
    pcdata->smoother            = SMOOTHER_GS;
    pcdata->smooth_order        = CF_ORDER;
    pcdata->presmooth_iter      = 1;
    pcdata->postsmooth_iter     = 1;
    pcdata->relaxation          = 1.1;
    pcdata->coarsening_type     = 1;
    pcdata->coarse_scaling      = ON;
    pcdata->amli_degree         = 1;
    pcdata->nl_amli_krylov_type = SOLVER_GCG;
}

AMG_data * fasp_amg_data_create (SHORT max_levels)
{
    max_levels = MAX(1, max_levels); // at least allocate one level

    AMG_data *mgl = (AMG_data *)fasp_mem_calloc(max_levels,sizeof(AMG_data));

    INT i;
    for ( i=0; i<max_levels; ++i ) {
        mgl[i].max_levels = max_levels;
        mgl[i].num_levels = 0;
        mgl[i].near_kernel_dim = 0;
        mgl[i].near_kernel_basis = NULL;
        mgl[i].cycle_type = 0;
#if MULTI_COLOR_ORDER
        mgl[i].GS_Theta = 0.0E-2; //0.0; //1.0E-2;
#endif
    }

    return(mgl);
}

void fasp_amg_data_free (AMG_data   *mgl,
                         AMG_param  *param)
{
    const INT max_levels = MAX(1,mgl[0].num_levels);

    INT i;

    switch (param->coarse_solver) {

#if WITH_MUMPS
        /* Destroy MUMPS direct solver on the coarsest level */
        case SOLVER_MUMPS:  {
            mgl[max_levels-1].mumps.job = 3;
            fasp_solver_mumps_steps(&mgl[max_levels-1].A, &mgl[max_levels-1].b,
                                    &mgl[max_levels-1].x, &mgl[max_levels-1].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        /* Destroy UMFPACK direct solver on the coarsest level */
        case SOLVER_UMFPACK:  {
            fasp_mem_free(mgl[max_levels-1].Numeric); mgl[max_levels-1].Numeric = NULL;
            break;
        }
#endif

#if WITH_PARDISO
        /* Destroy PARDISO direct solver on the coarsest level */
        case SOLVER_PARDISO:  {
            fasp_pardiso_free_internal_mem(&mgl[max_levels-1].pdata);
            break;
        }

#endif
        default:  // Do nothing!
            break;
    }

    for ( i=0; i<max_levels; ++i ) {
        fasp_ilu_data_free(&mgl[i].LU);
        fasp_dcsr_free(&mgl[i].A);
        if ( max_levels > 1 ) {
            fasp_dcsr_free(&mgl[i].P);
            fasp_dcsr_free(&mgl[i].R);
        }
        fasp_dvec_free(&mgl[i].b);
        fasp_dvec_free(&mgl[i].x);
        fasp_dvec_free(&mgl[i].w);
        fasp_ivec_free(&mgl[i].cfmark);
        fasp_swz_data_free(&mgl[i].Schwarz);
    }

    for ( i=0; i<mgl->near_kernel_dim; ++i ) {
        fasp_mem_free(mgl->near_kernel_basis[i]); mgl->near_kernel_basis[i] = NULL;
    }

    fasp_mem_free(mgl->near_kernel_basis); mgl->near_kernel_basis = NULL;
    fasp_mem_free(mgl); mgl = NULL;

    if ( param == NULL ) return; // exit if no param given

    if ( param->cycle_type == AMLI_CYCLE ) {
        fasp_mem_free(param->amli_coef); param->amli_coef = NULL;
    }

}

AMG_data_bsr * fasp_amg_data_bsr_create (SHORT max_levels)
{
    max_levels = MAX(1, max_levels); // at least allocate one level

    AMG_data_bsr *mgl = (AMG_data_bsr *)fasp_mem_calloc(max_levels,sizeof(AMG_data_bsr));

    INT i;
    for (i=0; i<max_levels; ++i) {
        mgl[i].max_levels = max_levels;
        mgl[i].num_levels = 0;
        mgl[i].near_kernel_dim = 0;
        mgl[i].near_kernel_basis = NULL;
        mgl[i].A_nk = NULL;
        mgl[i].P_nk = NULL;
        mgl[i].R_nk = NULL;
    }

    return(mgl);
}

void fasp_amg_data_bsr_free (AMG_data_bsr *mgl)
{
    const INT max_levels = MAX(1,mgl[0].num_levels);

    INT i;

    for ( i = 0; i < max_levels; ++i ) {

        fasp_ilu_data_free(&mgl[i].LU);
        fasp_dbsr_free(&mgl[i].A);
        if ( max_levels > 1 ) {
            fasp_dbsr_free(&mgl[i].P);
            fasp_dbsr_free(&mgl[i].R);
        }
        fasp_dvec_free(&mgl[i].b);
        fasp_dvec_free(&mgl[i].x);
        fasp_dvec_free(&mgl[i].diaginv);
        fasp_dvec_free(&mgl[i].diaginv_SS);
        fasp_dcsr_free(&mgl[i].Ac);

        fasp_ilu_data_free(&mgl[i].PP_LU);
        fasp_dcsr_free(&mgl[i].PP);
        fasp_dbsr_free(&mgl[i].SS);
        fasp_dvec_free(&mgl[i].diaginv_SS);
        fasp_dvec_free(&mgl[i].w);
        fasp_ivec_free(&mgl[i].cfmark);

        fasp_mem_free(mgl[i].pw); mgl[i].pw = NULL;
        fasp_mem_free(mgl[i].sw); mgl[i].sw = NULL;
    }

    for ( i = 0; i < mgl->near_kernel_dim; ++i ) {
        fasp_mem_free(mgl->near_kernel_basis[i]); mgl->near_kernel_basis[i] = NULL;
    }
    fasp_mem_free(mgl->near_kernel_basis); mgl->near_kernel_basis = NULL;
    fasp_mem_free(mgl); mgl = NULL;
}

void fasp_ilu_data_create (const INT   iwk,
                           const INT   nwork,
                           ILU_data   *iludata)
{
```

```
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: iwk=%d, nwork=%d \n", iwk, nwork);
#endif

    iludata->ijlu=(INT*)fasp_mem_calloc(iwk, sizeof(INT));

    if (iludata->type == ILUtp) iludata->iperm=(INT*)fasp_mem_calloc(iludata->row*2, sizeof(INT));

    iludata->luval=(REAL*)fasp_mem_calloc(iwk, sizeof(REAL));

    iludata->work=(REAL*)fasp_mem_calloc(nwork, sizeof(REAL));
#if DEBUG_MODE > 0
    printf("### DEBUG: %s ......  %d [End]\n", __FUNCTION__,__LINE__);
#endif

    return;
}

void fasp_ilu_data_free (ILU_data *iludata)
{
    if ( iludata == NULL ) return; // There is nothing to do!

    fasp_mem_free(iludata->ijlu);  iludata->ijlu  = NULL;
    fasp_mem_free(iludata->luval); iludata->luval = NULL;
    fasp_mem_free(iludata->work);  iludata->work  = NULL;
    fasp_mem_free(iludata->ilevL); iludata->ilevL = NULL;
    fasp_mem_free(iludata->jlevL); iludata->jlevL = NULL;
    fasp_mem_free(iludata->ilevU); iludata->ilevU = NULL;
    fasp_mem_free(iludata->jlevU); iludata->jlevU = NULL;

    if ( iludata->type == ILUtp ) {

        if ( iludata->A != NULL ) {
            // To permute the matrix back to its original state use the loop:
            INT k;
            const INT  nnz   = iludata->A->nnz;
            const INT *iperm = iludata->iperm;
            for ( k = 0; k < nnz; k++ ) {
                // iperm is in Fortran array format
                iludata->A->JA[k] = iperm[ iludata->A->JA[k] ] -1;
            }
        }

        fasp_mem_free(iludata->iperm); iludata->iperm = NULL;
    }

    iludata->row = iludata->col   = iludata->nzlu  = iludata->nwork = \
    iludata->nb  = iludata->nlevL = iludata->nlevU = 0;
}

void fasp_swz_data_free (SWZ_data *swzdata)
{
    INT i;

    if ( swzdata == NULL ) return; // There is nothing to do!

    fasp_dcsr_free(&swzdata->A);

    for ( i=0; i<swzdata->nblk; ++i ) fasp_dcsr_free (&((swzdata->blk_data)[i]));

    swzdata->nblk = 0;

    fasp_mem_free  (swzdata->iblock);  swzdata->iblock = NULL;
    fasp_mem_free  (swzdata->jblock);  swzdata->jblock = NULL;

    fasp_dvec_free (&swzdata->rhsloc1);
    fasp_dvec_free (&swzdata->xloc1);

    swzdata->memt = 0;
    fasp_mem_free (swzdata->mask);  swzdata->mask = NULL;
    fasp_mem_free (swzdata->maxa);  swzdata->maxa = NULL;

#if WITH_MUMPS
    if ( swzdata->mumps == NULL ) return;

    for ( i=0; i<swzdata->nblk; ++i ) fasp_mumps_free (&((swzdata->mumps)[i]));
#endif
}

/*---------------------------------*/
```

```
/*--        End of File          --*/
/*---------------------------------*/
```

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 "PreMGSmoother.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, ItrSmootherBSR.c, ItrSmootherCSR.c, ItrSmootherCSRpoly.c, KryPcg.c, KryPvgmres.c, KrySPcg.c, and KrySPvgmres.c

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

mgl	Pointer to AMG data: AMG_data
param	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

mgl	Pointer to AMG data: AMG_data_bsr
param	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.

```
#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreMGUtil.inl"
#include "PreMGSmoother.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_solver_mgcycle (AMG_data   *mgl,
                          AMG_param  *param)
{
    const SHORT  prtlvl = param->print_level;
    const SHORT  amg_type = param->AMG_type;
    const SHORT  smoother = param->smoother;
    const SHORT  smooth_order = param->smooth_order;
    const SHORT  cycle_type = param->cycle_type;
    const SHORT  coarse_solver = param->coarse_solver;
    const SHORT  nl = mgl[0].num_levels;
    const REAL   relax = param->relaxation;
    const REAL   tol = param->tol * 1e-4;
    const SHORT  ndeg = param->polynomial_degree;

    // Schwarz parameters
    SWZ_param swzparam;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

    // local variables
    REAL alpha = 1.0;
    INT  num_lvl[MAX_AMG_LVL] = {0}, l = 0;

    // more general cycling types on each level --zcs 05/07/2020
    INT  ncycles[MAX_AMG_LVL] = {1};
    SHORT i;
    for ( i = 0; i < MAX_AMG_LVL; ++i ) ncycles[i] = 1; // initially V-cycle
    switch(cycle_type) {
        case 12:
            for ( i = MAX_AMG_LVL-2; i > 0; i -= 2 ) ncycles[i] = 2;
            break;
        case 21:
            for ( i = MAX_AMG_LVL-1; i > 0; i -= 2 ) ncycles[i] = 2;
            break;
        default:
            for ( i = 0; i < MAX_AMG_LVL; i += 1 ) ncycles[i] = cycle_type;
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
#endif

#if DEBUG_MODE > 1
    printf("### DEBUG: AMG_level = %d, ILU_level = %d\n", nl, mgl->ILU_levels);
#endif

ForwardSweep:
    while ( l < nl-1 ) {

        num_lvl[l]++;

        // pre-smoothing with ILU method
        if ( l < mgl->ILU_levels ) {
            fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
        }

        // or pre-smoothing with Schwarz method
        else if ( l < mgl->SWZ_levels ) {
            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz,&swzparam, &mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
            }
        }

        // or pre-smoothing with standard smoother
        else {
#if MULTI_COLOR_ORDER
            // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n",  __FUNCTION__, __LINE__);
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter,1);
#else
            fasp_dcsr_presmoothing(smoother, &mgl[l].A, &mgl[l].b, &mgl[l].x,
                                   param->presmooth_iter, 0, mgl[l].A.row-1, 1,
                                   relax, ndeg, smooth_order, mgl[l].cfmark.val);
#endif
        }

        // form residual r = b - A x
        fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
        fasp_blas_dcsr_aAxpy(-1.0, &mgl[l].A, mgl[l].x.val, mgl[l].w.val);

        // restriction r1 = R*r0
        switch ( amg_type ) {
            case UA_AMG:
                fasp_blas_dcsr_mxv_agg(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
                break;
            default:
                fasp_blas_dcsr_mxv(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
                break;
        }

        // prepare for the next level
        ++l; fasp_dvec_set(mgl[l].A.row, &mgl[l].x, 0.0);

    }

    // If AMG only has one level or we have arrived at the coarsest level,
    // call the coarse space solver:
    switch ( coarse_solver ) {

#if WITH_PARDISO
        case SOLVER_PARDISO:  {
            /* use Intel MKL PARDISO direct solver on the coarsest level */
            fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
            break;
        }
#endif

#if WITH_MUMPS
        case SOLVER_MUMPS:  {
            // use MUMPS direct solver on the coarsest level
            mgl[nl-1].mumps.job = 2;
            fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
            break;
        }
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:  {
            // use UMFPACK direct solver on the coarsest level
            fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
            break;
        }
#endif

#if WITH_SuperLU
        case SOLVER_SUPERLU:  {
            // use SuperLU direct solver on the coarsest level
            fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
            break;
        }
#endif

        default:
            // use iterative solver on the coarsest level
            fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);

    }

    // BackwardSweep:
    while ( l > 0 ) {

        --l;

        // find the optimal scaling factor alpha
        if ( param->coarse_scaling == ON ) {
            alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
                  / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
            alpha = MIN(alpha, 1.0); // Add this for safety!  --Chensong on 10/04/2014
        }

        // prolongation u = u + alpha*P*e1
        switch ( amg_type ) {
            case UA_AMG:
                fasp_blas_dcsr_aAxpy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
                break;
            default:
                fasp_blas_dcsr_aAxpy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
                break;
        }

        // post-smoothing with ILU method
        if ( l < mgl->ILU_levels ) {
            fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
        }

        // post-smoothing with Schwarz method
        else if ( l < mgl->SWZ_levels ) {
            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz,&swzparam, &mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz,&swzparam, &mgl[l].x, &mgl[l].b);
                    break;
            }
        }

        // post-smoothing with standard methods
        else {
#if MULTI_COLOR_ORDER
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->postsmooth_iter,-1);
#else
            fasp_dcsr_postsmoothing(smoother, &mgl[l].A, &mgl[l].b, &mgl[l].x,
                                    param->postsmooth_iter, 0, mgl[l].A.row-1, -1,
                                    relax, ndeg, smooth_order, mgl[l].cfmark.val);
#endif
        }

        // General cycling on each level --zcs
        if ( num_lvl[l] < ncycles[l] ) break;
        else num_lvl[l] = 0;
    }

    if ( l > 0 ) goto ForwardSweep;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

}

void fasp_solver_mgcycle_bsr (AMG_data_bsr  *mgl,
                              AMG_param     *param)
{
    const SHORT prtlvl        = param->print_level;
    const SHORT nl            = mgl[0].num_levels;
    const SHORT smoother      = param->smoother;
    const SHORT cycle_type    = param->cycle_type;
    const SHORT coarse_solver = param->coarse_solver;
    const REAL  relax         = param->relaxation;
    INT   steps               = param->presmooth_iter;

    // local variables
    INT nu_l[MAX_AMG_LVL] = {0}, l = 0;
    REAL alpha = 1.0;
    INT i;

    dvector r_nk, z_nk;

    if ( mgl[0].A_nk != NULL ) {
        fasp_dvec_alloc(mgl[0].A_nk->row, &r_nk);
        fasp_dvec_alloc(mgl[0].A_nk->row, &z_nk);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

#if DEBUG_MODE > 1
    printf("### DEBUG: AMG_level = %d, ILU_level = %d\n", nl, mgl->ILU_levels);
#endif

ForwardSweep:
    while ( l < nl-1 ) {
        nu_l[l]++;
        // pre smoothing
        if ( l < mgl->ILU_levels ) {
            fasp_smoother_dbsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
            for ( i=0; i<steps; i++ )
                fasp_smoother_dbsr_gs_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x, mgl[l].diaginv.val);
        }
        else {
            if ( steps > 0 ) {
                switch ( smoother ) {
                    case SMOOTHER_JACOBI:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_jacobi1(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                        mgl[l].diaginv.val);
                        break;
                    case SMOOTHER_GS:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_gs_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                          mgl[l].diaginv.val);
                        break;
                    case SMOOTHER_SGS:
                        for (i=0; i<steps; i++){
                            fasp_smoother_dbsr_gs_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                          mgl[l].diaginv.val);
                            fasp_smoother_dbsr_gs_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val);
                        }
                        break;
                    case SMOOTHER_SOR:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_sor_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val, relax);
                        break;
                    case SMOOTHER_SSOR:
                        for (i=0; i<steps; i++) {
                            fasp_smoother_dbsr_sor_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val, relax);
                        }
                        fasp_smoother_dbsr_sor_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                        mgl[l].diaginv.val, relax);
                        break;
                    default:
                        printf("### ERROR: Unknown smoother type %d!\n", smoother);
                        fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
                }
            }
        }

        // extra kernel solve
        if (mgl[l].A_nk != NULL) {

            //--------------------------------------------
            // extra kernel solve
            //--------------------------------------------
            // form residual r = b - A x
            fasp_darray_cp(mgl[l].A.ROW*mgl[l].A.nb, mgl[l].b.val, mgl[l].w.val);
            fasp_blas_dbsr_aAxpy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);

            // r_nk = R_nk*r
            fasp_blas_dcsr_mxv(mgl[l].R_nk, mgl[l].w.val, r_nk.val);

            // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
            fasp_solver_umfpack(mgl[l].A_nk, &r_nk, &z_nk, 0);
#else
            fasp_coarse_itsolver(mgl[l].A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

            // z = z + P_nk*z_nk;
            fasp_blas_dcsr_aAxpy(1.0, mgl[l].P_nk, z_nk.val, mgl[l].x.val);
        }

        // form residual r = b - A x
        fasp_darray_cp(mgl[l].A.ROW*mgl[l].A.nb, mgl[l].b.val, mgl[l].w.val);
        fasp_blas_dbsr_aAxpy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);

        // restriction r1 = R*r0
        fasp_blas_dbsr_mxv(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);

        // prepare for the next level
        ++l; fasp_dvec_set(mgl[l].A.ROW*mgl[l].A.nb, &mgl[l].x, 0.0);

    }

    // If AMG only has one level or we have arrived at the coarsest level,
    // call the coarse space solver:
    switch ( coarse_solver ) {

#if WITH_PARDISO
```

```
        case SOLVER_PARDISO:  {
            /* use Intel MKL PARDISO direct solver on the coarsest level */
            fasp_pardiso_solve(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
            break;
        }
#endif

#if WITH_MUMPS
        case SOLVER_MUMPS:
            /* use MUMPS direct solver on the coarsest level */
            mgl[nl-1].mumps.job = 2;
            fasp_solver_mumps_steps(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
            break;
#endif

#if WITH_UMFPACK
        case SOLVER_UMFPACK:
            /* use UMFPACK direct solver on the coarsest level */
            fasp_umfpack_solve(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
            break;
#endif

#if WITH_SuperLU
        case SOLVER_SUPERLU:
            /* use SuperLU direct solver on the coarsest level */
            fasp_solver_superlu(&mgl[nl-1].Ac, &mgl[nl-1].b, &mgl[nl-1].x, 0);
            break;
#endif

        default:  {
            /* use iterative solver on the coarsest level */
            const INT  csize = mgl[nl-1].A.ROW*mgl[nl-1].A.nb;
            const INT  cmaxit = MIN(csize*csize, 200); // coarse level iteration number
            const REAL ctol = param->tol; // coarse level tolerance
            if ( fasp_solver_dbsr_pvgmres(&mgl[nl-1].A,&mgl[nl-1].b,&mgl[nl-1].x,
                                          NULL,ctol,cmaxit,25,1,0) < 0 ) {
                if ( prtlvl > PRINT_MIN ) {
                    printf("### WARNING: Coarse level solver did not converge!\n");
                    printf("### WARNING: Consider to increase maxit to %d!\n", 2*cmaxit);
                }
            }
        }
    }

    // BackwardSweep:
    while ( l > 0 ) {
        --l;

        // prolongation u = u + alpha*P*e1
        if ( param->coarse_scaling == ON ) {
            dvector PeH, Aeh;
            PeH.row = Aeh.row = mgl[l].b.row;
            PeH.val = mgl[l].w.val + mgl[l].b.row;
            Aeh.val = PeH.val + mgl[l].b.row;

            fasp_blas_dbsr_mxv (&mgl[l].P, mgl[l+1].x.val,  PeH.val);
            fasp_blas_dbsr_mxv (&mgl[l].A, PeH.val, Aeh.val);

            alpha = (fasp_blas_darray_dotprod (mgl[l].b.row, Aeh.val, mgl[l].w.val))
                  / (fasp_blas_darray_dotprod (mgl[l].b.row, Aeh.val, Aeh.val));
            alpha = MIN(alpha, 1.0); // Add this for safety!  --Chensong on 10/04/2014
            fasp_blas_darray_axpy (mgl[l].b.row, alpha, PeH.val, mgl[l].x.val);
        }
        else {
            fasp_blas_dbsr_aAxpy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
        }

        // extra kernel solve
        if ( mgl[l].A_nk != NULL ) {
            //--------------------------------------------
            // extra kernel solve
            //--------------------------------------------
            // form residual r = b - A x
            fasp_darray_cp(mgl[l].A.ROW*mgl[l].A.nb, mgl[l].b.val, mgl[l].w.val);
            fasp_blas_dbsr_aAxpy(-1.0, &mgl[l].A, mgl[l].x.val, mgl[l].w.val);

            // r_nk = R_nk*r
            fasp_blas_dcsr_mxv(mgl[l].R_nk, mgl[l].w.val, r_nk.val);

            // z_nk = A_nk^{-1}*r_nk
#if WITH_UMFPACK // use UMFPACK directly
```

```
            fasp_solver_umfpack(mgl[l].A_nk, &r_nk, &z_nk, 0);
#else
            fasp_coarse_itsolver(mgl[l].A_nk, &r_nk, &z_nk, 1e-12, 0);
#endif

            // z = z + P_nk*z_nk;
            fasp_blas_dcsr_aAxpy(1.0, mgl[l].P_nk, z_nk.val, mgl[l].x.val);
        }

        // post-smoothing
        if ( l < mgl->ILU_levels ) {
            fasp_smoother_dbsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
            for ( i=0; i<steps; i++ )
                fasp_smoother_dbsr_gs_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                               mgl[l].diaginv.val);
        }
        else {
            if ( steps > 0 ) {
                switch ( smoother ) {
                    case SMOOTHER_JACOBI:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_jacobi1(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                        mgl[l].diaginv.val);
                        break;
                    case SMOOTHER_GS:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_gs_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val);
                        break;
                    case SMOOTHER_SGS:
                        for (i=0; i<steps; i++){
                            fasp_smoother_dbsr_gs_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                          mgl[l].diaginv.val);
                            fasp_smoother_dbsr_gs_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val);
                        }
                        break;
                    case SMOOTHER_SOR:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_sor_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                            mgl[l].diaginv.val, relax);
                        break;
                    case SMOOTHER_SSOR:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_sor_ascend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                           mgl[l].diaginv.val, relax);
                        fasp_smoother_dbsr_sor_descend(&mgl[l].A, &mgl[l].b, &mgl[l].x,
                                                        mgl[l].diaginv.val, relax);
                        break;
                    default:
                        printf("### ERROR: Unknown smoother type %d!\n", smoother);
                        fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
                }
            }
        }

        if ( nu_l[l] < cycle_type ) break;
        else nu_l[l] = 0;
    }

    if ( l > 0 ) goto ForwardSweep;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 "PreMGSmoother.inl"
```

Functions

- void fasp_solver_fmgcycle (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

Definition in file PreMGCycleFull.c.

9.165.2 Function Documentation

9.165.2.1 fasp_solver_fmgcycle()

```
void fasp_solver_fmgcycle (
            AMG_data * mgl,
            AMG_param * param )
```

Solve Ax=b with non-recursive full multigrid K-cycle.

Parameters

mgl	Pointer to AMG data: AMG_data
param	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.

```
#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreMGUtil.inl"
#include "PreMGSmoother.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_solver_fmgcycle (AMG_data   *mgl,
                           AMG_param  *param)
{
    const SHORT  maxit = 3; // Max allowed V-cycles in each level
    const SHORT  amg_type = param->AMG_type;
    const SHORT  prtlvl = param->print_level;
    const SHORT  nl = mgl[0].num_levels;
    const SHORT  smoother = param->smoother;
    const SHORT  smooth_order = param->smooth_order;
    const SHORT  coarse_solver = param->coarse_solver;

    const REAL   relax = param->relaxation;
    const SHORT  ndeg = param->polynomial_degree;
    const REAL   tol = param->tol*1e-4;

    // local variables
    INT l, i, lvl, num_cycle;
    REAL alpha = 1.0, relerr;

    // Schwarz parameters
    SWZ_param swzparam;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
#endif

    if ( prtlvl >= PRINT_MOST )
        printf("FMG_level = %d, ILU_level = %d\n", nl, param->ILU_levels);

    // restriction r1 = R*r0
    switch (amg_type) {

        case UA_AMG:
            for (l=0;l<nl-1;l++)
                fasp_blas_dcsr_mxv_agg(&mgl[l].R, mgl[l].b.val, mgl[l+1].b.val);
            break;

        default:
            for (l=0;l<nl-1;l++)
                fasp_blas_dcsr_mxv(&mgl[l].R, mgl[l].b.val, mgl[l+1].b.val);
            break;

    }

    fasp_dvec_set(mgl[l].A.row, &mgl[l].x, 0.0); // initial guess

    // If only one level, just direct solver
    if ( nl==1 ) {

        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[nl-1].mumps.job = 2;
                fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
                break;
#endif

            default:
                /* use iterative solver on the coarest level */
                fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);

        }

        return;

    }

    for ( i=1; i<nl; i++ ) {

        // Coarse Space Solver:
        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[nl-1].mumps.job = 2;
                fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
                break;
#endif

            default:
                /* use iterative solver on the coarest level */
                fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);

        }

        // Slash part:  /-cycle
        {
            --l; // go back to finer level

            // find the optimal scaling factor alpha
            if ( param->coarse_scaling == ON ) {
                alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
                      / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
                alpha = MIN(alpha, 1.0); // Add this for safty!  --Chensong on 10/04/2014
            }

            // prolongation u = u + alpha*P*e1
            switch (amg_type) {
                case UA_AMG:
                    fasp_blas_dcsr_aAxpy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val); break;
                default:
                    fasp_blas_dcsr_aAxpy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val); break;
            }
        }

        // initialzie rel error
        num_cycle = 0; relerr = BIGREAL;

        while ( relerr > param->tol && num_cycle < maxit) {

            ++num_cycle;

            // form residual r = b - A x
            fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
            fasp_blas_dcsr_aAxpy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);
            relerr = fasp_blas_dvec_norm2(&mgl[l].w) / fasp_blas_dvec_norm2(&mgl[l].b);

            // Forward Sweep
            for ( lvl=0; lvl<i; lvl++ ) {

                // pre smoothing
                if (l<param->ILU_levels) {
                    fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
                }
                else if (l<mgl->SWZ_levels) {
                    switch (mgl[l].Schwarz.SWZ_type) {
                        case SCHWARZ_SYMMETRIC:
                            fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                            fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                            break;
                        default:
                            fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                            break;
                    }
                }

                else {
                    fasp_dcsr_presmoothing(smoother,&mgl[l].A,&mgl[l].b,&mgl[l].x,param->presmooth_iter,
                                           0,mgl[l].A.row-1,1,relax,ndeg,smooth_order,mgl[l].cfmark.val);
                }

                // form residual r = b - A x
                fasp_darray_cp(mgl[l].A.row, mgl[l].b.val, mgl[l].w.val);
                fasp_blas_dcsr_aAxpy(-1.0,&mgl[l].A, mgl[l].x.val, mgl[l].w.val);

                // restriction r1 = R*r0
                switch (amg_type) {
                    case UA_AMG:
                        fasp_blas_dcsr_mxv_agg(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
                        break;
                    default:
                        fasp_blas_dcsr_mxv(&mgl[l].R, mgl[l].w.val, mgl[l+1].b.val);
                        break;
                }

                ++l;

                // prepare for the next level
                fasp_dvec_set(mgl[l].A.row, &mgl[l].x, 0.0);

            }    // end for lvl

            // CoarseSpaceSolver:
            switch (coarse_solver) {

#if WITH_PARDISO
                case SOLVER_PARDISO:  {
                    /* use Intel MKL PARDISO direct solver on the coarsest level */
                    fasp_pardiso_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].pdata, 0);
                    break;
                }
#endif

#if WITH_SuperLU
                case SOLVER_SUPERLU:
                    /* use SuperLU direct solver on the coarsest level */
                    fasp_solver_superlu(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, 0);
                    break;
#endif

#if WITH_UMFPACK
                case SOLVER_UMFPACK:
                    /* use UMFPACK direct solver on the coarsest level */
                    fasp_umfpack_solve(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, mgl[nl-1].Numeric, 0);
                    break;
#endif

#if WITH_MUMPS
                case SOLVER_MUMPS:
                    /* use MUMPS direct solver on the coarsest level */
                    mgl[nl-1].mumps.job = 2;
                    fasp_solver_mumps_steps(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, &mgl[nl-1].mumps);
                    break;
#endif

                default:
                    /* use iterative solver on the coarest level */
                    fasp_coarse_itsolver(&mgl[nl-1].A, &mgl[nl-1].b, &mgl[nl-1].x, tol, prtlvl);

            }

            // Backward Sweep
            for ( lvl=0; lvl<i; lvl++ ) {

                --l;

                // find the optimal scaling factor alpha
                if ( param->coarse_scaling == ON ) {
                    alpha = fasp_blas_darray_dotprod(mgl[l+1].A.row, mgl[l+1].x.val, mgl[l+1].b.val)
                          / fasp_blas_dcsr_vmv(&mgl[l+1].A, mgl[l+1].x.val, mgl[l+1].x.val);
                    alpha = MIN(alpha, 1.0); // Add this for safty!  --Chensong on 10/04/2014
                }

                // prolongation u = u + alpha*P*e1
                switch (amg_type)
                {
                    case UA_AMG:
                        fasp_blas_dcsr_aAxpy_agg(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
                        break;
                    default:
                        fasp_blas_dcsr_aAxpy(alpha, &mgl[l].P, mgl[l+1].x.val, mgl[l].x.val);
                        break;
                }

                // post-smoothing
                if (l<param->ILU_levels) {
                    fasp_smoother_dcsr_ilu(&mgl[l].A, &mgl[l].b, &mgl[l].x, &mgl[l].LU);
                }
                else if (l<mgl->SWZ_levels) {
                    switch (mgl[l].Schwarz.SWZ_type) {
                        case SCHWARZ_SYMMETRIC:
                            fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                            fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                            break;
                        default:
                            fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                            break;
                    }
                }

                else {
                    fasp_dcsr_postsmoothing(smoother,&mgl[l].A,&mgl[l].b,&mgl[l].x,param->postsmooth_iter,

0,mgl[l].A.row-1,-1,relax,ndeg,smooth_order,mgl[l].cfmark.val);
                }

            } // end while

        } //end while

    } // end for
```

```

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 "PreMGSmoother.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.

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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param
level	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 ) {

        // pre smoothing
        if ( level < mgl[level].ILU_levels ) {
            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
        }
        else {
            fasp_dcsr_presmoothing(smoother,A0,b0,e0,param->presmooth_iter,
                                   0,m0-1,1,relax,ndeg,smooth_order,ordering);
        }

        // form residual r = b - A x
        fasp_darray_cp(m0,b0->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A0,e0->val,r);

        // restriction r1 = R*r0
        fasp_blas_dcsr_mxv(&mgl[level].R, r, b1->val);

        { // call MG recursively:  type = 1 for V cycle, type = 2 for W cycle
            SHORT i;
            fasp_dvec_set(m1,e1,0.0);
            for (i=0; i<cycle_type; ++i) fasp_solver_mgrecur (mgl, param, level+1);
        }

        // prolongation e0 = e0 + P*e1
        fasp_blas_dcsr_aAxpy(1.0, &mgl[level].P, e1->val, e0->val);

        // post smoothing
        if ( level < mgl[level].ILU_levels ) {
            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);
        }
        else {
            fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
                                    0,m0-1,-1,relax,ndeg,smooth_order,ordering);
        }

    }

    else { // coarsest level solver

        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(A0, b0, e0, &mgl[level].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(A0, b0, e0, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(A0, b0, e0, mgl[level].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[level].mumps.job = 2;
                fasp_solver_mumps_steps(A0, b0, e0, &mgl[level].mumps);
                break;
#endif

            /* use iterative solver on the coarsest level */
            default:
                fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);

        }

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
```

```
#endif
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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 "PreMGSmoother.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, ItrSmootherBSR.c, ItrSmootherCSR.c, ItrSmootherCSRpoly.c, KryPcg.c, KryPvfgmres.c, KrySPcg.c, KrySPvgmres.c, PreBSR.c, and PreCSR.c

This file includes both AMLI and non-linear AMLI cycles

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

lambda_max	Maximal lambda
lambda_min	Minimal lambda
degree	Degree of polynomial approximation
coef	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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param
l	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

mgl	Pointer to AMG_data data
param	Pointer to AMG parameters
l	Current level
num_levels	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

mgl	Pointer to AMG data: AMG_data
param	Pointer to AMG parameters: AMG_param
l	Current level
num_levels	Total number of levels

Author

Xiaozhe Hu

Date

04/06/2010

Note

Nonlinear AMLI-cycle. 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 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.

```c

#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreMGUtil.inl"
#include "PreMGSmoother.inl"
#include "PreMGRecurAMLI.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_solver_amli (AMG_data   *mgl,
                       AMG_param  *param,
                       INT         l)
{
    const SHORT  amg_type=param->AMG_type;
    const SHORT  prtlvl = param->print_level;
    const SHORT  smoother = param->smoother;
    const SHORT  smooth_order = param->smooth_order;
    const SHORT  coarse_solver = param->coarse_solver;
    const SHORT  degree= param->amli_degree;
    const REAL   relax = param->relaxation;
    const REAL   tol = param->tol*1e-4;
    const SHORT  ndeg = param->polynomial_degree;

    // local variables
    REAL   alpha  = 1.0;
    REAL * coef   = param->amli_coef;

    dvector *b0 = &mgl[l].b,   *e0 = &mgl[l].x;   // fine level b and x
    dvector *b1 = &mgl[l+1].b, *e1 = &mgl[l+1].x; // coarse level b and x

    dCSRmat *A0 = &mgl[l].A;   // fine level matrix
    dCSRmat *A1 = &mgl[l+1].A; // coarse level matrix

    const INT m0 = A0->row, m1 = A1->row;

    INT      *ordering = mgl[l].cfmark.val; // smoother ordering
    ILU_data *LU_level = &mgl[l].LU;        // fine level ILU decomposition
    REAL     *r        = mgl[l].w.val;      // work array for residual
    REAL     *r1       = mgl[l+1].w.val+m1; // work array for residual

    // Schwarz parameters
    SWZ_param swzparam;
    if ( param->SWZ_levels > 0 ) {
        swzparam.SWZ_blksolver = param->SWZ_blksolver;
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
#endif

    if ( prtlvl >= PRINT_MOST )
        printf("AMLI level %d, smoother %d.\n", l, smoother);

    if ( l < mgl[l].num_levels-1 ) {

        // pre smoothing
        if ( l < mgl[l].ILU_levels ) {

            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);

        }

        else if ( l < mgl->SWZ_levels ) {

            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
            }
        }

        else {
#if MULTI_COLOR_ORDER
            // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n",  __FUNCTION__, __LINE__);
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter,1);
#else
            fasp_dcsr_presmoothing(smoother,A0,b0,e0,param->presmooth_iter,
                                   0,m0-1,1,relax,ndeg,smooth_order,ordering);
#endif
        }

        // form residual r = b - A x
        fasp_darray_cp(m0,b0->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A0,e0->val,r);

        // restriction r1 = R*r0
        switch (amg_type) {
            case UA_AMG:
                fasp_blas_dcsr_mxv_agg(&mgl[l].R, r, b1->val); break;
            default:
                fasp_blas_dcsr_mxv(&mgl[l].R, r, b1->val); break;
        }

        // coarse grid correction
        {
            INT i;

            fasp_darray_cp(m1,b1->val,r1);

            for ( i=1; i<=degree; i++ ) {
                fasp_dvec_set(m1,e1,0.0);
                fasp_solver_amli(mgl, param, l+1);

                // b1 = (coef[degree-i]/coef[degree])*r1 + A1*e1;
                // First, compute b1 = A1*e1
                fasp_blas_dcsr_mxv(A1, e1->val, b1->val);
                // Then, compute b1 = b1 + (coef[degree-i]/coef[degree])*r1
                fasp_blas_darray_axpy(m1, coef[degree-i]/coef[degree], r1, b1->val);
            }

            fasp_dvec_set(m1,e1,0.0);
            fasp_solver_amli(mgl, param, l+1);
        }

        // find the optimal scaling factor alpha
        fasp_blas_darray_ax(m1, coef[degree], e1->val);
        if ( param->coarse_scaling == ON ) {
            alpha = fasp_blas_darray_dotprod(m1, e1->val, r1)
            / fasp_blas_dcsr_vmv(A1, e1->val, e1->val);
            alpha = MIN(alpha, 1.0);
        }

        // prolongation e0 = e0 + alpha * P * e1
        switch (amg_type) {
            case UA_AMG:
                fasp_blas_dcsr_aAxpy_agg(alpha, &mgl[l].P, e1->val, e0->val);
                break;
            default:
                fasp_blas_dcsr_aAxpy(alpha, &mgl[l].P, e1->val, e0->val);
                break;
        }

        // post smoothing
        if ( l < mgl[l].ILU_levels ) {

            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);

        }

        else if (l<mgl->SWZ_levels) {

            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                    break;
            }
        }

        else {
#if MULTI_COLOR_ORDER
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->postsmooth_iter,-1);
#else
            fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
                                    0,m0-1,-1,relax,ndeg,smooth_order,ordering);
#endif
        }

    }

    else { // coarsest level solver

        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(A0, b0, e0, &mgl[l].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(A0, b0, e0, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(A0, b0, e0, mgl[l].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[l].mumps.job = 2;
                fasp_solver_mumps_steps(A0, b0, e0, &mgl[l].mumps);
                break;
#endif

            default:
                /* use iterative solver on the coarsest level */
                fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);

        }

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
}

void fasp_solver_namli (AMG_data   *mgl,
                        AMG_param  *param,
                        INT         l,
                        INT         num_levels)
{
    const SHORT  amg_type=param->AMG_type;
    const SHORT  prtlvl = param->print_level;
    const SHORT  smoother = param->smoother;
    const SHORT  smooth_order = param->smooth_order;
    const SHORT  coarse_solver = param->coarse_solver;
    const REAL   relax = param->relaxation;
    const REAL   tol = param->tol*1e-4;
    const SHORT  ndeg = param->polynomial_degree;

    dvector *b0 = &mgl[l].b,   *e0 = &mgl[l].x;   // fine level b and x
    dvector *b1 = &mgl[l+1].b, *e1 = &mgl[l+1].x; // coarse level b and x

    dCSRmat *A0 = &mgl[l].A;   // fine level matrix
    dCSRmat *A1 = &mgl[l+1].A; // coarse level matrix

    const INT m0 = A0->row, m1 = A1->row;

    INT      *ordering = mgl[l].cfmark.val; // smoother ordering
    ILU_data *LU_level = &mgl[l].LU;        // fine level ILU decomposition
    REAL     *r        = mgl[l].w.val;      // work array for residual

    dvector uH;  // for coarse level correction
    uH.row = m1; uH.val = mgl[l+1].w.val + m1;

    // Schwarz parameters
    SWZ_param swzparam;
    if ( param->SWZ_levels > 0 )
        swzparam.SWZ_blksolver = param->SWZ_blksolver;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.row, mgl[0].A.nnz);
#endif

    if ( prtlvl >= PRINT_MOST )
        printf("Nonlinear AMLI level %d, smoother %d.\n", num_levels, smoother);

    if ( l < num_levels-1 ) {

        // pre smoothing
        if ( l < mgl[l].ILU_levels ) {

            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);

        }

        else if ( l < mgl->SWZ_levels ) {

            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_forward (&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_forward (&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
            }
        }

        else {
#if MULTI_COLOR_ORDER
            // printf("fasp_smoother_dcsr_gs_multicolor, %s, %d\n",  __FUNCTION__, __LINE__);
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->presmooth_iter,1);
#else
            fasp_dcsr_presmoothing(smoother,A0,b0,e0,param->presmooth_iter,
                                   0,m0-1,1,relax,ndeg,smooth_order,ordering);
```

```
#endif
        }

        // form residual r = b - A x
        fasp_darray_cp(m0,b0->val,r);
        fasp_blas_dcsr_aAxpy(-1.0,A0,e0->val,r);

        // restriction r1 = R*r0
        switch (amg_type) {
            case UA_AMG:
                fasp_blas_dcsr_mxv_agg(&mgl[l].R, r, b1->val);
                break;
            default:
                fasp_blas_dcsr_mxv(&mgl[l].R, r, b1->val);
        }

        // call nonlinear AMLI-cycle recursively
        {
            fasp_dvec_set(m1,e1,0.0);

            // V-cycle will be enforced when needed !!!
            if ( mgl[l+1].cycle_type <= 1 ) {

                fasp_solver_namli(&mgl[l+1], param, 0, num_levels-1);

            }

            else { // recursively call preconditioned Krylov method on coarse grid

                precond_data pcdata;

                fasp_param_amg_to_prec(&pcdata, param);
                pcdata.maxit = 1;
                pcdata.max_levels = num_levels-1;
                pcdata.mgl_data = &mgl[l+1];

                precond pc;
                pc.data = &pcdata;
                pc.fct = fasp_precond_namli;

                fasp_darray_cp (m1, e1->val, uH.val);

                switch (param->nl_amli_krylov_type) {
                    case SOLVER_GCG:  // Use GCG
                        Kcycle_dcsr_pgcg(A1, b1, &uH, &pc);
                        break;
                    default:  // Use GCR
                        Kcycle_dcsr_pgcr(A1, b1, &uH, &pc);
                }

                fasp_darray_cp (m1, uH.val, e1->val);
            }

        }

        // prolongation e0 = e0 + P*e1
        switch (amg_type) {
            case UA_AMG:
                fasp_blas_dcsr_aAxpy_agg(1.0, &mgl[l].P, e1->val, e0->val);
                break;
            default:
                fasp_blas_dcsr_aAxpy(1.0, &mgl[l].P, e1->val, e0->val);
        }

        // post smoothing
        if ( l < mgl[l].ILU_levels ) {

            fasp_smoother_dcsr_ilu(A0, b0, e0, LU_level);

        }
        else if ( l < mgl->SWZ_levels ) {

            switch (mgl[l].Schwarz.SWZ_type) {
                case SCHWARZ_SYMMETRIC:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
                    fasp_dcsr_swz_forward(&mgl[l].Schwarz, &swzparam, &mgl[l].x, &mgl[l].b);
                    break;
                default:
                    fasp_dcsr_swz_backward(&mgl[l].Schwarz, &swzparam,&mgl[l].x, &mgl[l].b);
            }

        }

        else {
#if MULTI_COLOR_ORDER
            fasp_smoother_dcsr_gs_multicolor (&mgl[l].x, &mgl[l].A, &mgl[l].b, param->postsmooth_iter,-1);
#else
            fasp_dcsr_postsmoothing(smoother,A0,b0,e0,param->postsmooth_iter,
                                    0,m0-1,-1,relax,ndeg,smooth_order,ordering);
#endif
        }

    }

    else { // coarsest level solver

        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(A0, b0, e0, &mgl[l].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(A0, b0, e0, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(A0, b0, e0, mgl[l].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[l].mumps.job = 2;
                fasp_solver_mumps_steps(A0, b0, e0, &mgl[l].mumps);
                break;
#endif

            default:
                /* use iterative solver on the coarsest level */
                fasp_coarse_itsolver(A0, b0, e0, tol, prtlvl);

        }

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
}

void fasp_solver_namli_bsr (AMG_data_bsr  *mgl,
                            AMG_param     *param,
                            INT            l,
                            INT            num_levels)
{
    const SHORT  prtlvl = param->print_level;
    const SHORT  smoother = param->smoother;
    const SHORT  coarse_solver = param->coarse_solver;
    const REAL   relax = param->relaxation;
    const REAL   tol = param->tol;
    INT i;

    dvector *b0 = &mgl[l].b,   *e0 = &mgl[l].x; // fine level b and x
    dvector *b1 = &mgl[l+1].b, *e1 = &mgl[l+1].x; // coarse level b and x

    dBSRmat *A0 = &mgl[l].A; // fine level matrix
    dBSRmat *A1 = &mgl[l+1].A; // coarse level matrix
    const INT m0 = A0->ROW*A0->nb, m1 = A1->ROW*A1->nb;

    ILU_data *LU_level = &mgl[l].LU; // fine level ILU decomposition
    REAL *r = mgl[l].w.val; // for residual

    dvector uH, bH;  // for coarse level correction
    uH.row = m1; uH.val = mgl[l+1].w.val + m1;
    bH.row = m1; bH.val = mgl[l+1].w.val + 2*m1;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: n=%d, nnz=%d\n", mgl[0].A.ROW, mgl[0].A.NNZ);
#endif

    if (prtlvl>=PRINT_MOST)
        printf("Nonlinear AMLI: level %d, smoother %d.\n", l, smoother);

    if (l < num_levels-1) {

        // pre smoothing
        if (l<param->ILU_levels) {
            fasp_smoother_dbsr_ilu(A0, b0, e0, LU_level);
        }
        else {
            SHORT steps = param->presmooth_iter;

            if (steps > 0) {
                switch (smoother) {
                    case SMOOTHER_JACOBI:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_jacobi (A0, b0, e0);
                        break;
                    case SMOOTHER_GS:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_gs (A0, b0, e0, ASCEND, NULL);
                        break;
                    case SMOOTHER_SOR:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_sor (A0, b0, e0, ASCEND, NULL,relax);
                        break;
                    default:
                        printf("### ERROR: Unknown smoother type %d!\n", smoother);
                        fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
                }
            }
        }

        // form residual r = b - A x
        fasp_darray_cp(m0,b0->val,r);
        fasp_blas_dbsr_aAxpy(-1.0,A0,e0->val,r);

        fasp_blas_dbsr_mxv(&mgl[l].R, r, b1->val);

        // call nonlinear AMLI-cycle recursively
        {
            fasp_dvec_set(m1,e1,0.0);

            // The coarsest problem is solved exactly.
            // No need to call Krylov method on second coarsest level
            if (l == num_levels-2) {
                fasp_solver_namli_bsr(&mgl[l+1], param, 0, num_levels-1);
            }
            else { // recursively call preconditioned Krylov method on coarse grid
                precond_data_bsr pcdata;

                fasp_param_amg_to_precbsr (&pcdata, param);
                pcdata.maxit = 1;
                pcdata.max_levels = num_levels-1;
                pcdata.mgl_data = &mgl[l+1];

                precond pc;
                pc.data = &pcdata;
                pc.fct = fasp_precond_dbsr_namli;

                fasp_darray_cp (m1, b1->val, bH.val);
                fasp_darray_cp (m1, e1->val, uH.val);

                const INT maxit = param->amli_degree+1;

                fasp_solver_dbsr_pvfgmres(A1,&bH,&uH,&pc,param->tol,
                                          maxit,MIN(maxit,30),1,PRINT_NONE);

                fasp_darray_cp (m1, bH.val, b1->val);
                fasp_darray_cp (m1, uH.val, e1->val);
            }

        }

        fasp_blas_dbsr_aAxpy(1.0, &mgl[l].P, e1->val, e0->val);

        // post smoothing
        if (l < param->ILU_levels) {
            fasp_smoother_dbsr_ilu(A0, b0, e0, LU_level);
        }
        else {
            SHORT steps = param->postsmooth_iter;

            if (steps > 0) {
                switch (smoother) {
                    case SMOOTHER_JACOBI:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_jacobi (A0, b0, e0);
                        break;
                    case SMOOTHER_GS:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_gs(A0, b0, e0, ASCEND, NULL);
                        break;
                    case SMOOTHER_SOR:
                        for (i=0; i<steps; i++)
                            fasp_smoother_dbsr_sor(A0, b0, e0, ASCEND, NULL,relax);
                        break;
                    default:
                        printf("### ERROR: Unknown smoother type %d!\n", smoother);
                        fasp_chkerr(ERROR_SOLVER_TYPE, __FUNCTION__);
                }
            }
        }

    }

    else { // coarsest level solver

        switch (coarse_solver) {

#if WITH_PARDISO
            case SOLVER_PARDISO:  {
                /* use Intel MKL PARDISO direct solver on the coarsest level */
                fasp_pardiso_solve(&mgl[l].Ac, b0, e0, &mgl[l].pdata, 0);
                break;
            }
#endif

#if WITH_SuperLU
            case SOLVER_SUPERLU:
                /* use SuperLU direct solver on the coarsest level */
                fasp_solver_superlu(&mgl[l].Ac, b0, e0, 0);
                break;
#endif

#if WITH_UMFPACK
            case SOLVER_UMFPACK:
                /* use UMFPACK direct solver on the coarsest level */
                fasp_umfpack_solve(&mgl[l].Ac, b0, e0, mgl[l].Numeric, 0);
                break;
#endif

#if WITH_MUMPS
            case SOLVER_MUMPS:
                /* use MUMPS direct solver on the coarsest level */
                mgl[l].mumps.job = 2;
                fasp_solver_mumps_steps(&mgl[l].Ac, b0, e0, &mgl[l].mumps);
                break;
#endif

            default:
                /* use iterative solver on the coarsest level */
                fasp_coarse_itsolver(&mgl[l].Ac, b0, e0, tol, prtlvl);

        }

    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif
}
```

```

void fasp_amg_amli_coef (const REAL  lambda_max,
                         const REAL  lambda_min,
                         const INT   degree,
                         REAL       *coef)
{
    const REAL mu0 = 1.0/lambda_max, mu1 = 1.0/lambda_min;
    const REAL c = (sqrt(mu0)+sqrt(mu1))*(sqrt(mu0)+sqrt(mu1));
    const REAL a = (4*mu0*mu1)/(c);

    const REAL kappa = lambda_max/lambda_min; // condition number
    const REAL delta = (sqrt(kappa) - 1.0)/(sqrt(kappa)+1.0);
    const REAL b = delta*delta;

    if (degree == 0) {
        coef[0] = 0.5*(mu0+mu1);
    }

    else if (degree == 1) {
        coef[0] = 0.5*c;
        coef[1] = -1.0*mu0*mu1;
    }

    else if (degree > 1) {
        INT i;

        // allocate memory
        REAL *work = (REAL *)fasp_mem_calloc(2*degree-1, sizeof(REAL));
        REAL *coef_k, *coef_km1;
        coef_k = work; coef_km1 = work+degree;

        // get q_k
        fasp_amg_amli_coef(lambda_max, lambda_min, degree-1, coef_k);
        // get q_km1
        fasp_amg_amli_coef(lambda_max, lambda_min, degree-2, coef_km1);

        // get coef
        coef[0] = a - b*coef_km1[0] + (1+b)*coef_k[0];

        for (i=1; i<degree-1; i++) {
            coef[i] = -b*coef_km1[i] + (1+b)*coef_k[i] - a*coef_k[i-1];
        }

        coef[degree-1] = (1+b)*coef_k[degree-1] - a*coef_k[degree-2];

        coef[degree] = -a*coef_k[degree-1];

        // clean memory
        fasp_mem_free(work); work = NULL;
    }

    else {
        printf("### ERROR: Wrong AMLI degree %d!\n", degree);
        fasp_chkerr(ERROR_INPUT_PAR, __FUNCTION__);
    }

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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

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

mgl	Pointer to AMG data: AMG_data
param	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

mgl	Pointer to AMG data: AMG_data
param	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

mgl	Pointer to AMG data: AMG_data
param	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 Xiazhe 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

mgl	Pointer to AMG data: AMG_data
param	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.

```

#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

```

```
/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_amg_solve (AMG_data   *mgl,
                    AMG_param  *param)
{
    dCSRmat      *ptrA = &mgl[0].A;
    dvector      *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;

    const SHORT   prtlvl = param->print_level;
    const INT     MaxIt  = param->maxit;
    const REAL    tol    = param->tol;
    const REAL    sumb   = fasp_blas_dvec_norm2(b); // L2norm(b)

    // local variables
    REAL  solve_start, solve_end;
    REAL  relres1 = 1.0, absres0 = sumb, absres, factor;
    INT   iter = 0;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    fasp_gettime(&solve_start);

    // Print iteration information if needed
    fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, sumb, 0.0);

    // If b = 0, set x = 0 to be a trivial solution
    if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);

    // MG solver here
    while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) {

#if TRUE
        // Call one multigrid cycle -- non recursive version
        fasp_solver_mgcycle(mgl, param);
#else
        // Call one multigrid cycle -- recursive version
        fasp_solver_mgrecur(mgl, param, 0);
#endif

        // Form residual r = b - A*x
        fasp_dvec_cp(b, r);
        fasp_blas_dcsr_aAxpy(-1.0, ptrA, x->val, r->val);

        // Compute norms of r and convergence factor
        absres  = fasp_blas_dvec_norm2(r);       // residual ||r||
        relres1 = absres/MAX(SMALLREAL,sumb);   // relative residual ||r||/||b||
        factor  = absres/absres0;                // contraction factor
        absres0 = absres;                        // prepare for next iteration

        // Print iteration information if needed
        fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, absres, factor);

        // Check convergence
        if ( relres1 < tol ) break;
    }

    if ( prtlvl > PRINT_NONE ) {
        ITS_FINAL(iter, MaxIt, relres1);
        fasp_gettime(&solve_end);
        fasp_cputime("AMG solve", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;}

INT fasp_amg_solve_amli (AMG_data   *mgl,
                         AMG_param  *param)
{
    dCSRmat     *ptrA = &mgl[0].A;
    dvector     *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;

    const INT    MaxIt  = param->maxit;
    const SHORT  prtlvl = param->print_level;
    const REAL   tol    = param->tol;
    const REAL   sumb   = fasp_blas_dvec_norm2(b); // L2norm(b)

    // local variables
    REAL         solve_start, solve_end;
    REAL         relres1 = 1.0, absres0 = sumb, absres, factor;
    INT          iter = 0;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    fasp_gettime(&solve_start);

    // Print iteration information if needed
    fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, sumb, 0.0);

    // If b = 0, set x = 0 to be a trivial solution
    if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);

    // MG solver here
    while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) {

        // Call one AMLI cycle
        fasp_solver_amli(mgl, param, 0);

        // Form residual r = b-A*x
        fasp_dvec_cp(b, r);
        fasp_blas_dcsr_aAxpy(-1.0, ptrA, x->val, r->val);

        // Compute norms of r and convergence factor
        absres  = fasp_blas_dvec_norm2(r);      // residual ||r||
        relres1 = absres/MAX(SMALLREAL,sumb);   // relative residual ||r||/||b||
        factor  = absres/absres0;               // contraction factor
        absres0 = absres;                       // prepare for next iteration

        // Print iteration information if needed
        fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, absres, factor);

        // Check convergence
        if ( relres1 < tol ) break;
    }

    if ( prtlvl > PRINT_NONE ) {
        ITS_FINAL(iter, MaxIt, relres1);
        fasp_gettime(&solve_end);
        fasp_cputime("AMLI solve", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

INT fasp_amg_solve_namli (AMG_data   *mgl,
                          AMG_param  *param)
{
    dCSRmat      *ptrA = &mgl[0].A;
    dvector      *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;

    const INT     MaxIt  = param->maxit;
    const SHORT   prtlvl = param->print_level;
    const REAL    tol    = param->tol;
    const REAL    sumb   = fasp_blas_dvec_norm2(b); // L2norm(b)

    // local variables
    REAL          solve_start, solve_end;
    REAL          relres1 = 1.0, absres0 = sumb, absres, factor;
    INT           iter = 0;

```

```
#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    fasp_gettime(&solve_start);

    // Print iteration information if needed
    fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, sumb, 0.0);

    // If b = 0, set x = 0 to be a trivial solution
    if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);

    while ( (iter++ < MaxIt) & (sumb > SMALLREAL) ) // MG solver here
    {
        // one multigrid cycle
        fasp_solver_namli(mgl, param, 0, mgl[0].num_levels);

        // r = b-A*x
        fasp_dvec_cp(b, r);
        fasp_blas_dcsr_aAxpy(-1.0, ptrA, x->val, r->val);

        absres  = fasp_blas_dvec_norm2(r);      // residual ||r||
        relres1 = absres/MAX(SMALLREAL,sumb);   // relative residual ||r||/||b||
        factor  = absres/absres0;               // contraction factor

        // output iteration information if needed
        fasp_itinfo(prtlvl, STOP_REL_RES, iter, relres1, absres, factor);

        if ( relres1 < tol ) break; // early exit condition

        absres0 = absres;
    }

    if ( prtlvl > PRINT_NONE ) {
        ITS_FINAL(iter, MaxIt, relres1);
        fasp_gettime(&solve_end);
        fasp_cputime("Nonlinear AMLI solve", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( iter > MaxIt )
        return ERROR_SOLVER_MAXIT;
    else
        return iter;
}

void fasp_famg_solve (AMG_data   *mgl,
                      AMG_param  *param)
{
    dCSRmat     *ptrA = &mgl[0].A;
    dvector     *b = &mgl[0].b, *x = &mgl[0].x, *r = &mgl[0].w;

    const SHORT  prtlvl = param->print_level;
    const REAL   sumb   = fasp_blas_dvec_norm2(b); // L2norm(b)

    // local variables
    REAL         solve_start, solve_end;
    REAL         relres1 = 1.0, absres;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nrow = %d, ncol = %d, nnz = %d\n",
           mgl[0].A.row, mgl[0].A.col, mgl[0].A.nnz);
#endif

    fasp_gettime(&solve_start);

    // If b = 0, set x = 0 to be a trivial solution
    if ( sumb <= SMALLREAL ) fasp_dvec_set(x->row, x, 0.0);

    // Call one full multigrid cycle
    fasp_solver_fmgcycle(mgl, param);

    // Form residual r = b-A*x
    fasp_dvec_cp(b, r);
    fasp_blas_dcsr_aAxpy(-1.0, ptrA, x->val, r->val);

    // Compute norms of r and convergence factor
    absres  = fasp_blas_dvec_norm2(r);     // residual ||r||
    relres1 = absres/MAX(SMALLREAL,sumb); // relative residual ||r||/||b||

    if ( prtlvl > PRINT_NONE ) {
        printf("FMG finishes with relative residual %e.\n", relres1);
        fasp_gettime(&solve_end);
        fasp_cputime("FMG solve", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.173 PreSTR.c File Reference

Preconditioners for dSTRmat matrices.

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

Functions

- void fasp_precond_dstr_diag (REAL ∗r, REAL ∗z, void ∗data)

  Diagonal preconditioner z=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 ItrSmootherSTR.c

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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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=inv(D)∗r.

Parameters

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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

r	Pointer to the vector needs preconditioning
z	Pointer to preconditioned vector
data	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.

```

#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
```

```
/*--  Declare Private Functions  --*/
/*----------------------------------*/

static inline void fasp_darray_cp_nc3 (const REAL *x, REAL *y);
static inline void fasp_darray_cp_nc5 (const REAL *x, REAL *y);
static inline void fasp_darray_cp_nc7 (const REAL *x, REAL *y);

/*----------------------------------*/
/*--      Public Functions       --*/
/*----------------------------------*/

void fasp_precond_dstr_diag (REAL *r,
                             REAL *z,
                             void *data)
{
    const precond_diag_str *diag = (precond_diag_str *)data;
    const REAL *diagptr = diag->diag.val;
    const INT   nc = diag->nc, nc2 = nc*nc;
    const INT   m = diag->diag.row/nc2;

    INT i;
    for ( i=0; i<m; ++i ) {
        fasp_blas_smat_mxv(&(diagptr[i*nc2]),&(r[i*nc]),&(z[i*nc]),nc);
    }
}

void fasp_precond_dstr_ilu0 (REAL *r,
                             REAL *z,
                             void *data)
{
    INT i, ic, ic2;
    REAL *zz,*zr,*tc;
    INT nline, nplane;

    dSTRmat *ILU_data=(dSTRmat *)data;
    INT m=ILU_data->ngrid;
    INT nc=ILU_data->nc;
    INT nc2=nc*nc;
    INT nx=ILU_data->nx;
    INT ny=ILU_data->ny;
    INT nz=ILU_data->nz;
    INT nxy=ILU_data->nxy;
    INT size=m*nc;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    if (nx == 1) {
        nline = ny;
        nplane = m;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = m;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = m;
    }
    else {
        nline = nx;
        nplane = nxy;
    }

    tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

    zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

    zr=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

    // copy residual r to zr, to save r
    memcpy(zr,r,(size)*sizeof(REAL));

    if (nc == 1) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        zz[0]=zr[0];
        for (i=1;i<m;++i) {
            zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
            if (i>=nline) zz[i]=zz[i]-ILU_data->offdiag[2][i-nline]*zz[i-nline];
            if (i>=nplane) zz[i]=zz[i]-ILU_data->offdiag[4][i-nplane]*zz[i-nplane];
        }

        // backward sweep:  solve upper matrix equation U*z=zz
        z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        for (i=m-2;i>=0;i--) {
            zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            if (i<m-nline) zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline];
            if (i<m-nplane) zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nplane];
            z[i]=zz[i]*ILU_data->diag[i];
        }

    } // end if (nc == 1)

    else if (nc == 3) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;

            fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 3)

    else if (nc == 5) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;

            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 5)


    else if (nc == 7) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;

            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 7)

    else {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
        for (i=1;i<m;++i) {
            ic=i*nc;

            fasp_blas_smat_mxv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
            fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));

            if (i>=nline) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));

        } // end for (i=1; i<m; ++i)

        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);

        for (i=m-2;i>=0;i--) {
            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
            fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);

        }// end for (i=m-2;i>=0;i--)
    } // end else

    fasp_mem_free(zr); zr = NULL;
    fasp_mem_free(zz); zz = NULL;
    fasp_mem_free(tc); tc = NULL;

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

void fasp_precond_dstr_ilu1 (REAL *r,
                             REAL *z,
                             void *data)
{
    REAL *zz,*zr,*tc;

    dSTRmat *ILU_data=(dSTRmat *)data;
    INT i,ic, ic2;
    INT m=ILU_data->ngrid;
    INT nc=ILU_data->nc;
    INT nc2=nc*nc;
    INT nx=ILU_data->nx;
    INT ny=ILU_data->ny;
    INT nz=ILU_data->nz;
    INT nxy=ILU_data->nxy;
    INT size=m*nc;
    INT nline, nplane;

    if (nx == 1) {
        nline = ny;
        nplane = m;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = m;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = m;
    }
    else {
        nline = nx;
```

```
            nplane = nxy;
        }

        tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

        zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

        zr=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

        // copy residual r to zr, to save r
        for (i=0;i<size;++i) zr[i]=r[i];
        if (nc == 1) {
            // forward sweep:  solve unit lower matrix equation L*zz=zr
            zz[0]=zr[0];
            for (i=1;i<m;++i) {

                zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                if (i>=nline-1)
                    zz[i]=zz[i]-ILU_data->offdiag[2][i-nline+1]*zz[i-nline+1];

                if (i>=nline)
                    zz[i]=zz[i]-ILU_data->offdiag[4][i-nline]*zz[i-nline];
                if (i>=nplane-nline)
                    zz[i]=zz[i]-ILU_data->offdiag[6][i-nplane+nline]*zz[i-nplane+nline];
                if (i>=nplane-1)
                    zz[i]=zz[i]-ILU_data->offdiag[8][i-nplane+1]*zz[i-nplane+1];
                if (i>=nplane)
                    zz[i]=zz[i]-ILU_data->offdiag[10][i-nplane]*zz[i-nplane];
            }

            // backward sweep:  solve upper matrix equation U*z=zz

            z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            for (i=m-2;i>=0;i--) {

                zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                if (i+nline-1<m)
                    zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline-1];
                if (i+nline<m)
                    zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nline];
                if (i+nplane-nline<m)
                    zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nplane-nline];
                if (i+nplane-1<m)
                    zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nplane-1];
                if (i+nplane<m)
                    zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nplane];

                z[i]=ILU_data->diag[i]*zz[i];

            }

        }     // end if (nc == 1)

        else if (nc == 3) {

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));

            for (i=1;i<m;++i) {
                ic=i*nc;

                //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));

                if (i>=nline-1) {
                    //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
```

```
fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }

            if (i>=nplane-1) {
                // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];

fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);

                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }

            fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
        }

        // backward sweep:  solve upper matrix equation U*z=zz

        // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        fasp_blas_smat_mxv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {
            ic=i*nc;
            ic2=ic*nc;

            //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));

            if (i+nline-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i+nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i+nplane-nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);

                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i+nplane-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i+nplane<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            //z[i]=ILU_data->diag[i]*zz[i];
            fasp_blas_smat_mxv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for (i=m-2;i>=0;i--)

    }  // end if (nc == 3)

    else if (nc == 5) {

        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;

            //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));

            if (i>=nline-1) {
                //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }

            if (i>=nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }

            if (i>=nplane-nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }

            if (i>=nplane-1) {
                // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }

            fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
        }

        // backward sweep:  solve upper matrix equation U*z=zz

        // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        fasp_blas_smat_mxv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {
            ic=i*nc;
            ic2=ic*nc;

            //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));

            if (i+nline-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i+nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i+nplane-nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i+nplane-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i+nplane<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            //z[i]=ILU_data->diag[i]*zz[i];
            fasp_blas_smat_mxv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for (i=m-2;i>=0;i--)

    }  // end if (nc == 5)

    else if (nc == 7) {

        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;

            //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));

            if (i>=nline-1) {
                //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);

                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }

            if (i>=nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);

                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }

            if (i>=nplane-nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];

fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }

            if (i>=nplane-1) {
                // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];

fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);

                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }

            fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
        }

        // backward sweep:  solve upper matrix equation U*z=zz

        // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        fasp_blas_smat_mxv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {
            ic=i*nc;
            ic2=ic*nc;

            //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));

            if (i+nline-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i+nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i+nplane-nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i+nplane-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i+nplane<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            //z[i]=ILU_data->diag[i]*zz[i];
            fasp_blas_smat_mxv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for (i=m-2;i>=0;i--)

    }  // end if (nc == 7)

    else {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp(nc,&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;
            //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
            fasp_blas_smat_mxv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
            fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));

            if (i>=nline-1) {
                //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nplane-nline) {
                //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
fasp_blas_smat_mxv(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nplane-1) {
                // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
fasp_blas_smat_mxv(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }
            fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
        }

        // backward sweep:  solve upper matrix equation U*z=zz

        // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        fasp_blas_smat_mxv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);

        for (i=m-2;i>=0;i--) {
            ic=i*nc;
            ic2=ic*nc;
            //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            fasp_blas_smat_mxv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
            fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));

            if (i+nline-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            if (i+nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            if (i+nplane-nline<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            if (i+nplane-1<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            if (i+nplane<m) {
                //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
            }

            //z[i]=ILU_data->diag[i]*zz[i];
            fasp_blas_smat_mxv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);
        }
    }  // end else

    fasp_mem_free(zr); zr = NULL;
    fasp_mem_free(zz); zz = NULL;
    fasp_mem_free(tc); tc = NULL;

    return;
}

void fasp_precond_dstr_ilu0_forward (REAL *r,
                                     REAL *z,
                                     void *data)
{
    INT i, ic;
    REAL *zz,*zr,*tc;
    INT nline, nplane;

    dSTRmat *ILU_data=(dSTRmat *)data;
    INT m=ILU_data->ngrid;
    INT nc=ILU_data->nc;
    INT nc2=nc*nc;
    INT nx=ILU_data->nx;
    INT ny=ILU_data->ny;
    INT nz=ILU_data->nz;
    INT nxy=ILU_data->nxy;
    INT size=m*nc;

    if (nx == 1) {
        nline = ny;
        nplane = m;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = m;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = m;
    }
    else {
        nline = nx;
        nplane = nxy;
    }

    tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

    zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

    zr=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

    // copy residual r to zr, to save r
    memcpy(zr,r,(size)*sizeof(REAL));
    if (nc == 1) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        zz[0]=zr[0];
        for (i=1;i<m;++i) {
            zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
            if (i>=nline) zz[i]=zz[i]-ILU_data->offdiag[2][i-nline]*zz[i-nline];
            if (i>=nplane) zz[i]=zz[i]-ILU_data->offdiag[4][i-nplane]*zz[i-nplane];
        }
    } // end if (nc == 1)

    else if (nc == 3) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;
            fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

    } // end else if (nc == 3)

    else if (nc == 5) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;
            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

    } // end else if (nc == 5)


    else if (nc == 7) {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));

        for (i=1;i<m;++i) {
            ic=i*nc;
            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            if (i>=nline) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }
            if (i>=nplane) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
            }
            fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
        } // end for (i=1;i<m;++i)

    } // end else if (nc == 7)


    else {
        // forward sweep:  solve unit lower matrix equation L*zz=zr
        fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
        for (i=1;i<m;++i) {
            ic=i*nc;
            fasp_blas_smat_mxv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
            fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));

            if (i>=nline) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[2][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            if (i>=nplane) {
                fasp_blas_smat_mxv(&(ILU_data->offdiag[4][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
            }

            fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));

        } // end for (i=1; i<m; ++i)

    } // end else

    memcpy(z,zz,(size)*sizeof(REAL));

    fasp_mem_free(zr); zr = NULL;
    fasp_mem_free(zz); zz = NULL;
    fasp_mem_free(tc); tc = NULL;

    return;
}

void fasp_precond_dstr_ilu0_backward (REAL *r,
                                      REAL *z,
                                      void *data)
{
    INT i, ic, ic2;
    REAL *zz,*tc;
    INT nline, nplane;

    dSTRmat *ILU_data=(dSTRmat *)data;
    INT m=ILU_data->ngrid;
    INT nc=ILU_data->nc;
    INT nc2=nc*nc;
    INT nx=ILU_data->nx;
    INT ny=ILU_data->ny;
    INT nz=ILU_data->nz;
    INT nxy=ILU_data->nxy;
    INT size=m*nc;

    if (nx == 1) {
        nline = ny;
        nplane = m;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = m;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = m;
    }
    else {
        nline = nx;
        nplane = nxy;
    }

    tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

    zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

    // copy residual r to zr, to save r
    memcpy(zz,r,(size)*sizeof(REAL));
    if (nc == 1) {
        // backward sweep:  solve upper matrix equation U*z=zz

        z[m-1]=zz[m-1]*ILU_data->diag[m-1];
        for (i=m-2;i>=0;i--) {
            zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
            if (i<m-nline) zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline];
            if (i<m-nplane) zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nplane];
            z[i]=zz[i]*ILU_data->diag[i];
        }

    } // end if (nc == 1)

    else if (nc == 3) {
        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 3)

    else if (nc == 5) {
        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 5)

    else if (nc == 7) {
        // backward sweep:  solve upper matrix equation U*z=zz
        fasp_blas_smat_mxv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

        for (i=m-2;i>=0;i--) {

            ic=i*nc;
            ic2=i*nc2;

            fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
            fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));

            if (i<m-nline) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            if (i<m-nplane) {
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
            }

            fasp_blas_smat_mxv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
        } // end for for (i=m-2;i>=0;i--)

    } // end else if (nc == 7)


    else
        {
            // backward sweep:  solve upper matrix equation U*z=zz
            fasp_blas_smat_mxv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);

            for (i=m-2;i>=0;i--) {
                ic=i*nc;
                ic2=i*nc2;

                fasp_blas_smat_mxv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));

                if (i<m-nline) {
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                if (i<m-nplane) {
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nplane)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                fasp_blas_smat_mxv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);

            }// end for (i=m-2;i>=0;i--)
        } // end else

    fasp_mem_free(zz); zz = NULL;
    fasp_mem_free(tc); tc = NULL;

    return;
}

void fasp_precond_dstr_ilu1_forward (REAL *r,
                                     REAL *z,
                                     void *data)
{
    REAL *zz,*zr,*tc;

    dSTRmat *ILU_data=(dSTRmat *)data;
    INT i,ic;
    INT m=ILU_data->ngrid;
    INT nc=ILU_data->nc;
    INT nc2=nc*nc;
    INT nx=ILU_data->nx;
    INT ny=ILU_data->ny;
    INT nz=ILU_data->nz;
    INT nxy=ILU_data->nxy;
    INT size=m*nc;
    INT nline, nplane;

    if (nx == 1) {
        nline = ny;
        nplane = m;
    }
    else if (ny == 1) {
        nline = nx;
        nplane = m;
    }
    else if (nz == 1) {
        nline = nx;
        nplane = m;
    }
    else {
        nline = nx;
```

```
            nplane = nxy;
        }

        tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

        zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

        zr=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

        // copy residual r to zr, to save r
        //for (i=0;i<size;++i) zr[i]=r[i];
        memcpy(zr,r,(size)*sizeof(REAL));
        if (nc == 1) {
            // forward sweep:  solve unit lower matrix equation L*zz=zr
            zz[0]=zr[0];
            for (i=1;i<m;++i) {

                zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                if (i>=nline-1)
                    zz[i]=zz[i]-ILU_data->offdiag[2][i-nline+1]*zz[i-nline+1];

                if (i>=nline)
                    zz[i]=zz[i]-ILU_data->offdiag[4][i-nline]*zz[i-nline];
                if (i>=nplane-nline)
                    zz[i]=zz[i]-ILU_data->offdiag[6][i-nplane+nline]*zz[i-nplane+nline];
                if (i>=nplane-1)
                    zz[i]=zz[i]-ILU_data->offdiag[8][i-nplane+1]*zz[i-nplane+1];
                if (i>=nplane)
                    zz[i]=zz[i]-ILU_data->offdiag[10][i-nplane]*zz[i-nplane];
            }

        }     // end if (nc == 1)

        else if (nc == 3) {

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp_nc3(&(zr[0]),&(zz[0]));

            for (i=1;i<m;++i) {
                ic=i*nc;
                //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));

                if (i>=nline-1) {
                    //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-1) {
                    // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                if (i>=nplane) {
                    //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zr[ic]));
                }

                fasp_darray_cp_nc3(&(zr[ic]),&(zz[ic]));
            }

        }  // end if (nc == 3)

        else if (nc == 5) {

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp_nc5(&(zr[0]),&(zz[0]));

            for (i=1;i<m;++i) {
                ic=i*nc;
                //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));

                if (i>=nline-1) {
                    //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);

                    fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
                }

                if (i>=nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];

    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-1) {
                    // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];

    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                    fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
                }

                if (i>=nplane) {
                    //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);

                    fasp_blas_darray_axpy_nc5(-1,tc,&(zr[ic]));
                }

                fasp_darray_cp_nc5(&(zr[ic]),&(zz[ic]));
            }

        }  // end if (nc == 5)

        else if (nc == 7) {

            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp_nc7(&(zr[0]),&(zz[0]));

            for (i=1;i<m;++i) {
                ic=i*nc;
                //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));

                if (i>=nline-1) {
                    //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc);

                    fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
                }

                if (i>=nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
                }

                if (i>=nplane-1) {
                    // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
                }

                if (i>=nplane) {
                    //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zr[ic]));
                }

                fasp_darray_cp_nc7(&(zr[ic]),&(zz[ic]));
            }

        }  // end if (nc == 7)

        else {
            // forward sweep:  solve unit lower matrix equation L*zz=zr
            fasp_darray_cp(nc,&(zr[0]),&(zz[0]));
            for (i=1;i<m;++i) {
                ic=i*nc;
                //zz[i]=zr[i]-ILU_data->offdiag[0][i-1]*zz[i-1];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[0][(i-1)*nc2]),&(zz[(i-1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));

                if (i>=nline-1) {
                    //zz[i]=zz[i]-ILU_data->offdiag[2][i-nx+1]*zz[i-nx+1];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[2][(i-nline+1)*nc2]),&(zz[(i-nline+1)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
                }

                if (i>=nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[4][i-nx]*zz[i-nx];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[4][(i-nline)*nc2]),&(zz[(i-nline)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
                }

                if (i>=nplane-nline) {
                    //zz[i]=zz[i]-ILU_data->offdiag[6][i-nxy+nx]*zz[i-nxy+nx];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[6][(i-nplane+nline)*nc2]),&(zz[(i-nplane+nline)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
                }

                if (i>=nplane-1) {
                    // zz[i]=zz[i]-ILU_data->offdiag[8][i-nxy+1]*zz[i-nxy+1];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[8][(i-nplane+1)*nc2]),&(zz[(i-nplane+1)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
                }

                if (i>=nplane) {
                    //zz[i]=zz[i]-ILU_data->offdiag[10][i-nxy]*zz[i-nxy];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[10][(i-nplane)*nc2]),&(zz[(i-nplane)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zr[ic]));
                }
                fasp_darray_cp(nc,&(zr[ic]),&(zz[ic]));
            }
        }  // end else

        memcpy(z,zz,(size)*sizeof(REAL));

        fasp_mem_free(zr); zr = NULL;
        fasp_mem_free(zz); zz = NULL;
        fasp_mem_free(tc); tc = NULL;

        return;
    }

    void fasp_precond_dstr_ilu1_backward (REAL *r,
                                          REAL *z,
                                          void *data)
    {
        REAL *zz,*tc;

        dSTRmat *ILU_data=(dSTRmat *)data;
        INT i,ic, ic2;
        INT m=ILU_data->ngrid;
        INT nc=ILU_data->nc;
        INT nc2=nc*nc;
        INT nx=ILU_data->nx;
        INT ny=ILU_data->ny;
        INT nz=ILU_data->nz;
        INT nxy=ILU_data->nxy;
        INT size=m*nc;
        INT nline, nplane;

        if (nx == 1) {
            nline = ny;
            nplane = m;
        }
        else if (ny == 1) {
            nline = nx;
            nplane = m;
        }
        else if (nz == 1) {
            nline = nx;
            nplane = m;
        }
        else {
            nline = nx;
            nplane = nxy;
        }

        tc=(REAL*)fasp_mem_calloc(nc, sizeof(REAL));

        zz=(REAL*)fasp_mem_calloc(size, sizeof(REAL));

        // copy residual r to zr, to save r
        //for (i=0;i<size;++i) zr[i]=r[i];
        memcpy(zz,r,(size)*sizeof(REAL));
        if (nc == 1) {
            // backward sweep:  solve upper matrix equation U*z=zz

            z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            for (i=m-2;i>=0;i--) {

                zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                if (i+nline-1<m)
                    zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nline-1];
                if (i+nline<m)
                    zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nline];
                if (i+nplane-nline<m)
                    zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nplane-nline];
                if (i+nplane-1<m)
                    zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nplane-1];
                if (i+nplane<m)
                    zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nplane];

                z[i]=ILU_data->diag[i]*zz[i];

            }

        }     // end if (nc == 1)

        else if (nc == 3) {
            // backward sweep:  solve upper matrix equation U*z=zz

            // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            fasp_blas_smat_mxv_nc3(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

            for (i=m-2;i>=0;i--) {
                ic=i*nc;
                ic2=ic*nc;

                //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
                fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));

                if (i+nline-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
                }

                if (i+nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                    fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
                }

                if (i+nplane-nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
                }

                if (i+nplane-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
                }

                if (i+nplane<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                    fasp_blas_smat_mxv_nc3(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                    fasp_blas_darray_axpy_nc3(-1,tc,&(zz[ic]));
                }

                //z[i]=ILU_data->diag[i]*zz[i];
                fasp_blas_smat_mxv_nc3(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
            } // end for (i=m-2;i>=0;i--)

        }  // end if (nc == 3)

        else if (nc == 5) {
            // backward sweep:  solve upper matrix equation U*z=zz

            // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            fasp_blas_smat_mxv_nc5(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

            for (i=m-2;i>=0;i--) {
                ic=i*nc;
                ic2=ic*nc;

                //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
                fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));

                if (i+nline-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
                }

                if (i+nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                    fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
                }

                if (i+nplane-nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
                }

                if (i+nplane-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
                }

                if (i+nplane<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                    fasp_blas_smat_mxv_nc5(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                    fasp_blas_darray_axpy_nc5(-1,tc,&(zz[ic]));
                }

                //z[i]=ILU_data->diag[i]*zz[i];
                fasp_blas_smat_mxv_nc5(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
            } // end for (i=m-2;i>=0;i--)

        }  // end if (nc == 5)

        else if (nc == 7) {
            // backward sweep:  solve upper matrix equation U*z=zz

            // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            fasp_blas_smat_mxv_nc7(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]));

            for (i=m-2;i>=0;i--) {
                ic=i*nc;
                ic2=ic*nc;

                //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc);
                fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));

                if (i+nline-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
                }

                if (i+nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
                }

                if (i+nplane-nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc);

                    fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
                }

                if (i+nplane-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
                }

                if (i+nplane<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                    fasp_blas_smat_mxv_nc7(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc);
                    fasp_blas_darray_axpy_nc7(-1,tc,&(zz[ic]));
                }

                //z[i]=ILU_data->diag[i]*zz[i];
                fasp_blas_smat_mxv_nc7(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]));
            } // end for (i=m-2;i>=0;i--)

        }  // end if (nc == 7)

        else {
            // backward sweep:  solve upper matrix equation U*z=zz

            // z[m-1]=zz[m-1]*ILU_data->diag[m-1];
            fasp_blas_smat_mxv(&(ILU_data->diag[(m-1)*nc2]),&(zz[(m-1)*nc]),&(z[(m-1)*nc]),nc);

            for (i=m-2;i>=0;i--) {
                ic=i*nc;
                ic2=ic*nc;
                //zz[i]=zz[i]-ILU_data->offdiag[1][i]*z[i+1];
                fasp_blas_smat_mxv(&(ILU_data->offdiag[1][ic2]),&(z[(i+1)*nc]),tc,nc);
                fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));

                if (i+nline-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[3][i]*z[i+nx-1];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[3][ic2]),&(z[(i+nline-1)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                if (i+nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[5][i]*z[i+nx];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[5][ic2]),&(z[(i+nline)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                if (i+nplane-nline<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[7][i]*z[i+nxy-nx];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[7][ic2]),&(z[(i+nplane-nline)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                if (i+nplane-1<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[9][i]*z[i+nxy-1];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[9][ic2]),&(z[(i+nplane-1)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }

                if (i+nplane<m) {
                    //zz[i]=zz[i]-ILU_data->offdiag[11][i]*z[i+nxy];
                    fasp_blas_smat_mxv(&(ILU_data->offdiag[11][ic2]),&(z[(i+nplane)*nc]),tc,nc);
                    fasp_blas_darray_axpy(nc,-1,tc,&(zz[ic]));
                }
                //z[i]=ILU_data->diag[i]*zz[i];
                fasp_blas_smat_mxv(&(ILU_data->diag[ic2]),&(zz[ic]),&(z[ic]),nc);
            }
        }  // end else

        fasp_mem_free(zz); zz = NULL;
        fasp_mem_free(tc); tc = NULL;

        return;
    }

    void fasp_precond_dstr_blockgs (REAL *r,
                                    REAL *z,
                                    void *data)
    {
        precond_data_str *predata=(precond_data_str *)data;
        dSTRmat *A = predata->A_str;
        dvector *diaginv = predata->diaginv;
        ivector *pivot = predata->pivot;
        ivector *order = predata->order;
        ivector *neigh = predata->neigh;

        INT i;
        const INT nc = A->nc;
        const INT ngrid = A->ngrid;
        const INT n   = nc*ngrid;  // whole size

        dvector zz, rr;
        zz.row=rr.row=n; zz.val=z; rr.val=r;
        fasp_dvec_set(n,&zz,0.0);

        for (i=0; i<1; ++i)
            fasp_smoother_dstr_swz(A, &rr, &zz, diaginv, pivot, neigh, order);
    }

    /*---------------------------------*/
    /*--      Private Functions      --*/
    /*---------------------------------*/

    static inline void fasp_darray_cp_nc3 (const REAL  *x,
                                           REAL        *y)
    {
        memcpy(y, x, 3*sizeof(REAL));
    }

    static inline void fasp_darray_cp_nc5 (const REAL  *x,
                                           REAL        *y)
    {
        memcpy(y, x, 5*sizeof(REAL));
    }

    static inline void fasp_darray_cp_nc7 (const REAL  *x,
                                           REAL        *y)
    {
        memcpy(y, x, 7*sizeof(REAL));
    }

    /*---------------------------------*/
    /*--        End of File          --*/
    /*---------------------------------*/
```

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

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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
x	Pointer to dvector: the unknowns
param	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) {

        case SA_AMG:  // Smoothed Aggregation AMG setup
            status = fasp_amg_setup_sa(mgl, param);
            break;

        case UA_AMG:  // Unsmoothed Aggregation AMG setup
            status = fasp_amg_setup_ua(mgl, param);
            break;

        default:  // Classical AMG setup
            status = fasp_amg_setup_rs(mgl, param);
            break;
    }

    // Step 2:  AMG solve phase
    if (status == FASP_SUCCESS) { // call a multilevel cycle

        switch (cycle_type) {

            case AMLI_CYCLE:  // AMLI-cycle
                iter = fasp_amg_solve_amli(mgl, param);
                break;

            case NL_AMLI_CYCLE:  // Nonlinear AMLI-cycle
                iter = fasp_amg_solve_namli(mgl, param);
                break;

            default:  // V,W-cycles or hybrid cycles (determined by param)
                iter = fasp_amg_solve(mgl, param);
                break;
        }

        fasp_dvec_cp(&mgl[0].x, x);

    }

    else { // call a backup solver

        if (prtlvl > PRINT_MIN) {
            printf("### WARNING: AMG setup failed!\n");
            printf("### WARNING: Use a backup solver instead!\n");
        }
        fasp_solver_dcsr_spgmres(A, b, x, NULL, tol, maxit, 20, 1, prtlvl);
    }

    // clean-up memory
    fasp_amg_data_free(mgl, param);

    // print out CPU time if needed
    if (prtlvl > PRINT_NONE) {
        fasp_gettime(&AMG_end);
        fasp_cputime("AMG totally", AMG_end - AMG_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

/*---------------------------------*/
/*--        End of File        --*/
/*---------------------------------*/
```

9.177 SolBLC.c File Reference

Iterative solvers for dBLCmat matrices.

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

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,
              precond * pc,
              ITS_param * itparam )
```

Solve Ax = b by standard Krylov methods.

Parameters

A	Pointer to the coeff matrix in dBLCmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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

A	Pointer to the coeff matrix in dBLCmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dBLCmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	Pointer to parameters for AMG solvers
A_diag	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

A	Pointer to the coeff matrix in dBLCmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	Pointer to parameters for AMG solvers
A_diag	Digonal blocks of A

Returns

Iteration number if converges; ERROR otherwise.

Author

Xiaozhe Hu

Date

07/06/2014

Warning

Only works for 4 by 4 block dCSRmat problems!! – Xiaozhe Hu

Definition 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

A	Pointer to the coeff matrix in dBLCmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
NumLayers	Number of layers used for sweeping preconditioner
Ai	Pointer to the coeff matrix for the preconditioner in dBLCmat format
local_A	Pointer to the local coeff matrices in the dCSRmat format
local_index	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 SolBLC.c

Go to the documentation of this file.

```
#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dblc_itsolver (dBLCmat    *A,
                               dvector    *b,
                               dvector    *x,
                               precond    *pc,
                               ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type = itparam->stop_type;
    const SHORT restart = itparam->restart;
    const INT   MaxIt = itparam->maxit;
    const REAL  tol = itparam->tol;

    REAL  solve_start, solve_end;
    INT   iter = ERROR_SOLVER_TYPE;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    switch (itsolver_type) {

        case SOLVER_BiCGstab:
            iter=fasp_solver_dblc_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_MinRes:
            iter=fasp_solver_dblc_pminres(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter=fasp_solver_dblc_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter=fasp_solver_dblc_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VFGMRES:
            iter=fasp_solver_dblc_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl >= PRINT_MIN) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Iterative method", solve_end - solve_start);
    }
```

```

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_dblc_krylov (dBLCmat    *A,
                             dvector    *b,
                             dvector    *x,
                             ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;

    INT status = FASP_SUCCESS;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // solver part
    fasp_gettime(&solve_start);

    status = fasp_solver_dblc_itsolver(A,b,x,NULL,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dblc_krylov_block3 (dBLCmat    *A,
                                    dvector    *b,
                                    dvector    *x,
                                    ITS_param  *itparam,
                                    AMG_param  *amgparam,
                                    dCSRmat    *A_diag)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT precond_type = itparam->precond_type;

    INT status = FASP_SUCCESS;
    REAL setup_start, setup_end;
    REAL solve_start, solve_end;

    const SHORT max_levels = amgparam->max_levels;
    INT m, n, nnz, i;

    AMG_data **mgl = NULL;

#if WITH_UMFPACK
    void **LU_diag = (void **)fasp_mem_calloc(3, sizeof(void *));
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /* setup preconditioner */
    fasp_gettime(&solve_start);
    fasp_gettime(&setup_start);

    /* diagonal blocks are solved exactly */
    if ( precond_type > 20 && precond_type < 30 ) {
#if WITH_UMFPACK
        // Need to sort the diagonal blocks for UMFPACK format
        dCSRmat A_tran;

        for (i=0; i<3; i++){

            A_tran = fasp_dcsr_create(A_diag[i].row, A_diag[i].col, A_diag[i].nnz);
            fasp_dcsr_transz(&A_diag[i], NULL, &A_tran);
            fasp_dcsr_cp(&A_tran, &A_diag[i]);

            printf("Factorization for %d-th diagnol:  \n", i);
            LU_diag[i] = fasp_umfpack_factorize(&A_diag[i], prtlvl);

        }

        fasp_dcsr_free(&A_tran);
#endif
    }

    /* diagonal blocks are solved by AMG */
    else if ( precond_type > 30 && precond_type < 40 ) {

        mgl = (AMG_data **)fasp_mem_calloc(3, sizeof(AMG_data *));

        for (i=0; i<3; i++){

            mgl[i] = fasp_amg_data_create(max_levels);
            m = A_diag[i].row; n = A_diag[i].col; nnz = A_diag[i].nnz;
            mgl[i][0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(&A_diag[i],&mgl[i][0].A);
            mgl[i][0].b=fasp_dvec_create(n); mgl[i][0].x=fasp_dvec_create(n);

            switch (amgparam->AMG_type) {
                case SA_AMG:  // Smoothed Aggregation AMG
                    status = fasp_amg_setup_sa(mgl[i], amgparam); break;
                case UA_AMG:  // Unsmoothed Aggregation AMG
                    status = fasp_amg_setup_ua(mgl[i], amgparam); break;
                default:  // Classical AMG
                    status = fasp_amg_setup_rs(mgl[i], amgparam); break;
            }

            fasp_chkerr(status, __FUNCTION__);
        }

    }

    else {
        fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
    }

    precond_data_blc precdata;
    precdata.Ablc = A;
    precdata.A_diag = A_diag;
    precdata.r = fasp_dvec_create(b->row);

    /* diagonal blocks are solved exactly */
    if ( precond_type > 20 && precond_type < 30 ) {
#if WITH_UMFPACK
        precdata.LU_diag = LU_diag;
#endif
    }
    /* diagonal blocks are solved by AMG */
    else if ( precond_type > 30 && precond_type < 40 ) {
        precdata.amgparam = amgparam;
        precdata.mgl = mgl;
    }
    else {
        fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
    }

    precond prec; prec.data = &precdata;

    switch (precond_type) {
        case 21:
            prec.fct = fasp_precond_dblc_diag_3; break;

        case 22:
            prec.fct = fasp_precond_dblc_lower_3; break;

        case 23:
            prec.fct = fasp_precond_dblc_upper_3; break;

        case 24:
            prec.fct = fasp_precond_dblc_SGS_3; break;

        case 31:
            prec.fct = fasp_precond_dblc_diag_3_amg; break;

        case 32:
            prec.fct = fasp_precond_dblc_lower_3_amg; break;

        case 33:
            prec.fct = fasp_precond_dblc_upper_3_amg; break;

        case 34:
            prec.fct = fasp_precond_dblc_SGS_3_amg; break;

        default:
            fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__); break;
    }

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&setup_end);
        fasp_cputime("Setup totally", setup_end - setup_start);
    }

    // solve part
    status = fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

    // clean up
    /* diagonal blocks are solved exactly */
    if ( precond_type > 20 && precond_type < 30 ) {
#if WITH_UMFPACK
        for (i=0; i<3; i++) fasp_umfpack_free_numeric(LU_diag[i]);
#endif
    }
    /* diagonal blocks are solved by AMG */
    else if ( precond_type > 30 && precond_type < 40 ) {
        for (i=0; i<3; i++) fasp_amg_data_free(mgl[i], amgparam);
        fasp_mem_free(mgl); mgl = NULL;
    }
    else {
        fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dblc_krylov_block4 (dBLCmat    *A,
                                    dvector    *b,
                                    dvector    *x,
                                    ITS_param  *itparam,
                                    AMG_param  *amgparam,
                                    dCSRmat    *A_diag)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT precond_type = itparam->precond_type;

    INT status = FASP_SUCCESS;
    REAL setup_start, setup_end;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /* setup preconditioner */
    fasp_gettime(&solve_start);
    fasp_gettime(&setup_start);

#if WITH_UMFPACK
    void **LU_diag = (void **)fasp_mem_calloc(4, sizeof(void *));
    INT i;
#endif

    /* diagonal blocks are solved exactly */
    if ( precond_type > 20 && precond_type < 30 ) {

#if WITH_UMFPACK
        // Need to sort the matrices local_A for UMFPACK format
        dCSRmat A_tran;

        for (i=0; i<4; i++){

            A_tran = fasp_dcsr_create(A_diag[i].row, A_diag[i].col, A_diag[i].nnz);
            fasp_dcsr_transz(&A_diag[i], NULL, &A_tran);
            fasp_dcsr_cp(&A_tran, &A_diag[i]);

            printf("Factorization for %d-th diagnol:  \n", i);
            LU_diag[i] = fasp_umfpack_factorize(&A_diag[i], prtlvl);

        }

        fasp_dcsr_free(&A_tran);
#endif

    }
    else {
        fasp_chkerr(ERROR_SOLVER_PRECTYPE, __FUNCTION__);
    }

    precond_data_blc precdata;

    precdata.Ablc = A;
    precdata.A_diag = A_diag;
#if WITH_UMFPACK
    precdata.LU_diag = LU_diag;
#endif
    precdata.r = fasp_dvec_create(b->row);

    precond prec; prec.data = &precdata;

    switch (precond_type)
    {
        case 21:
            prec.fct = fasp_precond_dblc_diag_4;
            break;

        case 22:
            prec.fct = fasp_precond_dblc_lower_4;
            break;
    }

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&setup_end);
        fasp_cputime("Setup totally", setup_end - setup_start);
    }

    // solver part
    status=fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

    // clean
#if WITH_UMFPACK
    for (i=0; i<4; i++) fasp_umfpack_free_numeric(LU_diag[i]);
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

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)
{
    const SHORT prtlvl = itparam->print_level;

    INT status = FASP_SUCCESS;
    REAL setup_start, setup_end;
    REAL solve_start, solve_end;

    void **local_LU = NULL;
```

```

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    /* setup preconditioner */
    fasp_gettime(&solve_start);
    fasp_gettime(&setup_start);

#if WITH_UMFPACK
    // Need to sort the matrices local_A for UMFPACK format
    INT l;
    dCSRmat A_tran;
    local_LU = (void **)fasp_mem_calloc(NumLayers, sizeof(void *));

    for ( l=0; l<NumLayers; l++ ) {

        A_tran = fasp_dcsr_create(local_A[l].row, local_A[l].col, local_A[l].nnz);
        fasp_dcsr_transz(&local_A[l], NULL, &A_tran);
        fasp_dcsr_cp(&A_tran, &local_A[l]);

        printf("Factorization for layer %d:  \n", l);
        local_LU[l] = fasp_umfpack_factorize(&local_A[l], prtlvl);

    }

    fasp_dcsr_free(&A_tran);
#endif

    precond_data_sweeping precdata;
    precdata.NumLayers = NumLayers;
    precdata.A = A;
    precdata.Ai = Ai;
    precdata.local_A = local_A;
    precdata.local_LU = local_LU;
    precdata.local_index = local_index;
    precdata.r = fasp_dvec_create(b->row);
    precdata.w = (REAL *)fasp_mem_calloc(10*b->row,sizeof(REAL));

    precond prec; prec.data = &precdata;
    prec.fct = fasp_precond_dblc_sweeping;

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&setup_end);
        fasp_cputime("Setup totally", setup_end - setup_start);
    }

    /* solver part */
    status = fasp_solver_dblc_itsolver(A,b,x, &prec,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

    // clean
#if WITH_UMFPACK
    for (l=0; l<NumLayers; l++) fasp_umfpack_free_numeric(local_LU[l]);
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.179 SolBSR.c File Reference

Iterative solvers for dBSRmat matrices.

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

```
#include "fasp_functs.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, BlaILUSetupBSR.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,
            precond * pc,
            ITS_param * itparam )
```

Solve Ax=b by preconditioned Krylov methods for BSR matrices.

Parameters

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	Pointer to parameters of AMG

Parameters

A_nk	Pointer to the coeff matrix for near kernel space in dBSRmat format
P_nk	Pointer to the prolongation for near kernel space in dBSRmat format
R_nk	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
iluparam	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

A	Pointer to the coeff matrix in dBSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	Pointer to parameters of AMG
nk_dim	Dimension of the near kernel spaces
nk	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.

```c

#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dbsr_itsolver (dBSRmat    *A,
                               dvector    *b,
                               dvector    *x,
                               precond    *pc,
                               ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type = itparam->stop_type;
    const SHORT restart = itparam->restart;
    const INT   MaxIt = itparam->maxit;
    const REAL  tol = itparam->tol;

    // Local variables
    INT iter = ERROR_SOLVER_TYPE;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    switch (itsolver_type) {

        case SOLVER_CG:
            iter = fasp_solver_dbsr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_BiCGstab:
            iter = fasp_solver_dbsr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter = fasp_solver_dbsr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter = fasp_solver_dbsr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VFGMRES:
            iter = fasp_solver_dbsr_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl > PRINT_MIN) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Iterative method", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_dbsr_krylov (dBSRmat    *A,
                             dvector    *b,
                             dvector    *x,
                             ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    INT status = FASP_SUCCESS;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // solver part
    fasp_gettime(&solve_start);

    status=fasp_solver_dbsr_itsolver(A,b,x,NULL,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl > PRINT_NONE )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dbsr_krylov_diag (dBSRmat    *A,
                                  dvector    *b,
                                  dvector    *x,
                                  ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    INT status = FASP_SUCCESS;
    REAL solve_start, solve_end;

    INT nb=A->nb,i,k;
    INT nb2=nb*nb;
    INT ROW=A->ROW;

#ifdef _OPENMP
    // variables for OpenMP
    INT myid, mybegin, myend;
    INT nthreads = fasp_get_num_threads();
#endif
    // setup preconditioner
    precond_diag_bsr diag;
    fasp_dvec_alloc(ROW*nb2, &(diag.diag));

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // get all the diagonal sub-blocks
#ifdef _OPENMP
    if (ROW > OPENMP_HOLDS) {
#pragma omp parallel for private(myid, mybegin, myend, i, k)
        for (myid=0; myid<nthreads; ++myid) {
            fasp_get_start_end(myid, nthreads, ROW, &mybegin, &myend);
            for (i = mybegin; i < myend; ++i) {
                for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
                    if (A->JA[k] == i)
                        memcpy(diag.diag.val+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
                }
            }
        }
    }
    else {
#endif
        for (i = 0; i < ROW; ++i) {
            for (k = A->IA[i]; k < A->IA[i+1]; ++k) {
                if (A->JA[k] == i)
                    memcpy(diag.diag.val+i*nb2, A->val+k*nb2, nb2*sizeof(REAL));
            }
        }
#ifdef _OPENMP
    }
#endif

    diag.nb=nb;

#ifdef _OPENMP
#pragma omp parallel for if(ROW>OPENMP_HOLDS)
#endif
    for (i=0; i<ROW; ++i){
        fasp_smat_inv(&(diag.diag.val[i*nb2]), nb);
    }

    precond *pc = (precond *)fasp_mem_calloc(1,sizeof(precond));
    pc->data = &diag;
    pc->fct  = fasp_precond_dbsr_diag;

    // solver part
    fasp_gettime(&solve_start);

    status=fasp_solver_dbsr_itsolver(A,b,x,pc,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl > PRINT_NONE )
        fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);

    // clean up
    fasp_dvec_free(&(diag.diag));

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dbsr_krylov_ilu (dBSRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 ITS_param  *itparam,
                                 ILU_param  *iluparam)
{
    const SHORT prtlvl = itparam->print_level;
    REAL solve_start, solve_end;
    INT status = FASP_SUCCESS;

    ILU_data LU;
    precond pc;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->ROW, A->COL, A->NNZ);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // ILU setup for whole matrix
    if ( (status = fasp_ilu_dbsr_setup(A, &LU, iluparam)) < 0 ) goto FINISHED;

    // check iludata
    if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;

    // set preconditioner
    pc.data = &LU; pc.fct = fasp_precond_dbsr_ilu;

    // solve
    status = fasp_solver_dbsr_itsolver(A, b, x, &pc, itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl > PRINT_NONE )
        fasp_cputime("ILUk_Krylov method totally", solve_end - solve_start);

FINISHED:
    fasp_ilu_data_free(&LU);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dbsr_krylov_amg (dBSRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 ITS_param  *itparam,
                                 AMG_param  *amgparam)
{
    //--------------------------------------------------------------
    // Part 1:  prepare
    // --------------------------------------------------------------

    const SHORT prtlvl = itparam->print_level;
    const SHORT max_levels = amgparam->max_levels;

    // return variable
    INT status = FASP_SUCCESS;

    // data of AMG
    AMG_data_bsr *mgl = fasp_amg_data_bsr_create(max_levels);

    // timing
    REAL setup_start, setup_end, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    //--------------------------------------------------------------
    //Part 2:  set up the preconditioner
    //--------------------------------------------------------------
    fasp_gettime(&setup_start);

    // initialize A, b, x for mgl[0]
    mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
    mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
    mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);

    fasp_dbsr_cp(A, &(mgl[0].A));

    switch (amgparam->AMG_type) {

        case SA_AMG:  // Smoothed Aggregation AMG
            status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;

        default:
            status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;

    }

    if (status < 0) goto FINISHED;

    precond_data_bsr precdata;
    precdata.print_level = amgparam->print_level;
    precdata.maxit = amgparam->maxit;
    precdata.tol = amgparam->tol;
    precdata.cycle_type = amgparam->cycle_type;
    precdata.smoother = amgparam->smoother;
    precdata.presmooth_iter = amgparam->presmooth_iter;
    precdata.postsmooth_iter = amgparam->postsmooth_iter;
    precdata.coarsening_type = amgparam->coarsening_type;
    precdata.relaxation = amgparam->relaxation;
    precdata.coarse_scaling = amgparam->coarse_scaling;
    precdata.amli_degree = amgparam->amli_degree;
    precdata.amli_coef = amgparam->amli_coef;
    precdata.tentative_smooth = amgparam->tentative_smooth;
    precdata.max_levels = mgl[0].num_levels;
    precdata.mgl_data = mgl;
    precdata.A = A;

    precond prec;
    prec.data = &precdata;
    switch (amgparam->cycle_type) {
        case NL_AMLI_CYCLE:  // Nonlinear AMLI AMG
            prec.fct = fasp_precond_dbsr_namli; break;
        default:  // V,W-Cycle AMG
            prec.fct = fasp_precond_dbsr_amg; break;
    }

    fasp_gettime(&setup_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("BSR AMG setup", setup_end - setup_start);

    //--------------------------------------------------------------
    // Part 3:  solver
    //--------------------------------------------------------------
    status = fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("BSR Krylov method", solve_end - setup_start);

FINISHED:
    fasp_amg_data_bsr_free(mgl);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if ( status == ERROR_ALLOC_MEM ) goto MEMORY_ERROR;
    return status;

MEMORY_ERROR:
    printf("### ERROR: Cannot allocate memory!  [%s]\n", __FUNCTION__);
    exit(status);
}

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)
{
    //--------------------------------------------------------------
    // Part 1:  prepare
    // --------------------------------------------------------------
    // parameters of iterative method
    const SHORT prtlvl = itparam->print_level;
    const SHORT max_levels = amgparam->max_levels;

    // return variable
    INT status = FASP_SUCCESS;

    // data of AMG
    AMG_data_bsr *mgl=fasp_amg_data_bsr_create(max_levels);

    // timing
    REAL setup_start, setup_end, setup_time;
    REAL solve_start, solve_end, solve_time;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    //------------------------------------------------------------
    //Part 2:  set up the preconditioner
    //------------------------------------------------------------
    fasp_gettime(&setup_start);

    // initialize A, b, x for mgl[0]
    mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
    fasp_dbsr_cp(A, &(mgl[0].A));
    mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
    mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);

    // near kernel space
    mgl[0].A_nk = NULL;
    mgl[0].P_nk = P_nk;
    mgl[0].R_nk = R_nk;

    switch (amgparam->AMG_type) {

        case SA_AMG:  // Smoothed Aggregation AMG
            status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;

        default:
            status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;

    }

    if (status < 0) goto FINISHED;

    precond_data_bsr precdata;
    precdata.print_level = amgparam->print_level;
    precdata.maxit = amgparam->maxit;
    precdata.tol = amgparam->tol;
    precdata.cycle_type = amgparam->cycle_type;
    precdata.smoother = amgparam->smoother;
    precdata.presmooth_iter = amgparam->presmooth_iter;
    precdata.postsmooth_iter = amgparam->postsmooth_iter;
    precdata.coarsening_type = amgparam->coarsening_type;
    precdata.relaxation = amgparam->relaxation;
    precdata.coarse_scaling = amgparam->coarse_scaling;
    precdata.amli_degree = amgparam->amli_degree;
    precdata.amli_coef = amgparam->amli_coef;
    precdata.tentative_smooth = amgparam->tentative_smooth;
    precdata.max_levels = mgl[0].num_levels;
    precdata.mgl_data = mgl;
    precdata.A = A;

#if WITH_UMFPACK // use UMFPACK directly
    dCSRmat A_tran;
    A_tran = fasp_dcsr_create(A_nk->row, A_nk->col, A_nk->nnz);
    fasp_dcsr_transz(A_nk, NULL, &A_tran);
    // It is equivalent to do transpose and then sort
    //     fasp_dcsr_trans(A_nk, &A_tran);
    //     fasp_dcsr_sort(&A_tran);
    precdata.A_nk = &A_tran;
#else
    precdata.A_nk = A_nk;
#endif

    precdata.P_nk = P_nk;
    precdata.R_nk = R_nk;

    if (status < 0) goto FINISHED;

    precond prec;
    prec.data = &precdata;

    prec.fct = fasp_precond_dbsr_amg_nk;

    fasp_gettime(&setup_end);

    setup_time = setup_end - setup_start;

    if ( prtlvl >= PRINT_MIN ) fasp_cputime("BSR AMG setup", setup_time);

    //---------------------------------------------------------------
    // Part 3:  solver
    //---------------------------------------------------------------
    fasp_gettime(&solve_start);

    status=fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);

    fasp_gettime(&solve_end);

    solve_time = solve_end - solve_start;

    if ( prtlvl >= PRINT_MIN ) {
        fasp_cputime("BSR Krylov method", setup_time+solve_time);
    }

FINISHED:
    fasp_amg_data_bsr_free(mgl);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

#if WITH_UMFPACK // use UMFPACK directly
    fasp_dcsr_free(&A_tran);
#endif
    if (status == ERROR_ALLOC_MEM) goto MEMORY_ERROR;
    return status;

MEMORY_ERROR:
    printf("### ERROR: Cannot allocate memory!  [%s]\n", __FUNCTION__);
    exit(status);
}

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)
{
    //---------------------------------------------------------------
    // Part 1:  prepare
    // ----------------------------------------------------------------
    const SHORT prtlvl = itparam->print_level;
    const SHORT max_levels = amgparam->max_levels;

    // local variable
    INT i;

    // return variable
    INT status = FASP_SUCCESS;

    // data of AMG
    AMG_data_bsr *mgl=fasp_amg_data_bsr_create(max_levels);

    // timing
    REAL setup_start, setup_end, setup_time;
    REAL solve_start, solve_end, solve_time;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    //---------------------------------------------------------------
    //Part 2:  set up the preconditioner
    //---------------------------------------------------------------
    fasp_gettime(&setup_start);

    // initialize A, b, x for mgl[0]
    mgl[0].A = fasp_dbsr_create(A->ROW, A->COL, A->NNZ, A->nb, A->storage_manner);
    fasp_dbsr_cp(A, &(mgl[0].A));
    mgl[0].b = fasp_dvec_create(mgl[0].A.ROW*mgl[0].A.nb);
    mgl[0].x = fasp_dvec_create(mgl[0].A.COL*mgl[0].A.nb);

    /*------------------------*/
    /*-- setup null spaces --*/
    /*------------------------*/

    // null space for whole Jacobian
    mgl[0].near_kernel_dim   = nk_dim;
    mgl[0].near_kernel_basis = (REAL **)fasp_mem_calloc(mgl->near_kernel_dim, sizeof(REAL*));

    for ( i=0; i < mgl->near_kernel_dim; ++i ) mgl[0].near_kernel_basis[i] = nk[i].val;

    switch (amgparam->AMG_type) {

        case SA_AMG:  // Smoothed Aggregation AMG
            status = fasp_amg_setup_sa_bsr(mgl, amgparam); break;

        default:
            status = fasp_amg_setup_ua_bsr(mgl, amgparam); break;

    }

    if (status < 0) goto FINISHED;

    precond_data_bsr precdata;
    precdata.print_level = amgparam->print_level;
    precdata.maxit = amgparam->maxit;
    precdata.tol = amgparam->tol;
    precdata.cycle_type = amgparam->cycle_type;
    precdata.smoother = amgparam->smoother;
    precdata.presmooth_iter = amgparam->presmooth_iter;
    precdata.postsmooth_iter = amgparam->postsmooth_iter;
    precdata.coarsening_type = amgparam->coarsening_type;
    precdata.relaxation = amgparam->relaxation;
    precdata.coarse_scaling = amgparam->coarse_scaling;
    precdata.amli_degree = amgparam->amli_degree;
    precdata.amli_coef = amgparam->amli_coef;
    precdata.tentative_smooth = amgparam->tentative_smooth;
    precdata.max_levels = mgl[0].num_levels;
    precdata.mgl_data = mgl;
    precdata.A = A;

    if (status < 0) goto FINISHED;

    precond prec;
    prec.data = &precdata;
    switch (amgparam->cycle_type) {
        case NL_AMLI_CYCLE:  // Nonlinear AMLI AMG
            prec.fct = fasp_precond_dbsr_namli;
            break;
        default:  // V,W-Cycle AMG
            prec.fct = fasp_precond_dbsr_amg;
            break;
    }

    fasp_gettime(&setup_end);

    setup_time = setup_end - setup_start;

    if ( prtlvl >= PRINT_MIN ) fasp_cputime("BSR AMG setup", setup_time);

    //--------------------------------------------------------------
    // Part 3:  solver
    //--------------------------------------------------------------
    fasp_gettime(&solve_start);

    status=fasp_solver_dbsr_itsolver(A,b,x,&prec,itparam);

    fasp_gettime(&solve_end);

    solve_time = solve_end - solve_start;

    if ( prtlvl >= PRINT_MIN ) {
        fasp_cputime("BSR Krylov method", setup_time+solve_time);
    }

FINISHED:
    fasp_amg_data_bsr_free(mgl);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    if (status == ERROR_ALLOC_MEM) goto MEMORY_ERROR;
    return status;

MEMORY_ERROR:
    printf("### ERROR: Cannot allocate memory!  [%s]\n", __FUNCTION__);
    exit(status);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, KryPgcg.c, KryPgcr.c, KryPgmres.c, KryPminres.c, KryPvfgmres.c, KryPvgmres.c, PreAMGSetupRS.c, PreAMGSetupSA.c, PreAMGSetupUA.c, PreCSR.c, and PreDataInit.c

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,
             precond * pc,
             ITS_param * itparam )
```

Solve Ax=b by preconditioned Krylov methods for CSR matrices.

Note

This is an abstract interface for iterative methods.

Parameters

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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,
             precond * 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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	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

A	Pointer to the coeff matrix in dCSRmat format

Parameters

b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
amgparam	Pointer to parameters for AMG methods
A_nk	Pointer to the coeff matrix of near kernel space in dCSRmat format
P_nk	Pointer to the prolongation of near kernel space in dCSRmat format
R_nk	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
iluparam	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
iluparam	Pointer to parameters for ILU
M	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dCSRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
schparam	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.

```
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dcsr_itsolver (dCSRmat    *A,
                               dvector    *b,
                               dvector    *x,
                               precond    *pc,
                               ITS_param  *itparam)
{
    const SHORT prtlvl        = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type     = itparam->stop_type;
    const SHORT restart       = itparam->restart;
    const INT   MaxIt         = itparam->maxit;
    const REAL  tol           = itparam->tol;

    /* Local Variables */
    REAL solve_start, solve_end;
    INT iter;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    /* check matrix data */
    fasp_check_dCSRmat(A);

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    /* Choose a desirable Krylov iterative solver */
    switch ( itsolver_type ) {
        case SOLVER_CG:
            iter = fasp_solver_dcsr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_BiCGstab:
            iter = fasp_solver_dcsr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_MinRes:
            iter = fasp_solver_dcsr_pminres(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter = fasp_solver_dcsr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter = fasp_solver_dcsr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VFGMRES:
            iter = fasp_solver_dcsr_pvfgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_GCG:
            iter = fasp_solver_dcsr_pgcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GCR:
            iter = fasp_solver_dcsr_pgcr(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Iterative method", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_dcsr_itsolver_s (dCSRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 precond    *pc,
                                 ITS_param  *itparam)
{
    const SHORT prtlvl        = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type     = itparam->stop_type;
    const SHORT restart       = itparam->restart;
    const INT   MaxIt         = itparam->maxit;
    const REAL  tol           = itparam->tol;

    /* Local Variables */
    REAL solve_start, solve_end;
    INT iter;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    /* check matrix data */
    fasp_check_dCSRmat(A);

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    /* Choose a desirable Krylov iterative solver */
    switch ( itsolver_type ) {
        case SOLVER_CG:
            iter = fasp_solver_dcsr_spcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_BiCGstab:
            iter = fasp_solver_dcsr_spbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_MinRes:
            iter = fasp_solver_dcsr_spminres(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter = fasp_solver_dcsr_spgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter = fasp_solver_dcsr_spvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Iterative method", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_dcsr_krylov (dCSRmat    *A,
                             dvector    *b,
                             dvector    *x,
                             ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    status = fasp_solver_dcsr_itsolver(A,b,x,NULL,itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Krylov method totally", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dcsr_krylov_s (dCSRmat    *A,
                               dvector    *b,
                               dvector    *x,
                               ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    status = fasp_solver_dcsr_itsolver_s (A,b,x,NULL,itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Krylov method totally", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dcsr_krylov_diag (dCSRmat    *A,
                                  dvector    *b,
                                  dvector    *x,
                                  ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    INT       status = FASP_SUCCESS;
    REAL      solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // setup preconditioner
    dvector diag; fasp_dcsr_getdiag(0,A,&diag);

    precond pc;
    pc.data = &diag;
    pc.fct  = fasp_precond_diag;

    // call iterative solver
    status = fasp_solver_dcsr_itsolver(A,b,x,&pc,itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);
    }

    fasp_dvec_free(&diag);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
```

```
#endif

    return status;
}

INT fasp_solver_dcsr_krylov_swz (dCSRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 ITS_param  *itparam,
                                 SWZ_param  *schparam)
{
    SWZ_param swzparam;
    swzparam.SWZ_mmsize    = schparam->SWZ_mmsize;
    swzparam.SWZ_maxlvl    = schparam->SWZ_maxlvl;
    swzparam.SWZ_type      = schparam->SWZ_type;
    swzparam.SWZ_blksolver = schparam->SWZ_blksolver;

    const SHORT prtlvl = itparam->print_level;

    REAL setup_start, setup_end;
    REAL solve_start, solve_end;
    INT status = FASP_SUCCESS;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);
    fasp_gettime(&setup_start);

    // setup preconditioner
    SWZ_data SWZ_data;

    // symmetrize the matrix (get rid of this later)
    SWZ_data.A = fasp_dcsr_sympart(A);

    // construct Schwarz precondtioner
    fasp_dcsr_shift (&SWZ_data.A, 1);
    fasp_swz_dcsr_setup (&SWZ_data, &swzparam);

    fasp_gettime (&setup_end);
    printf("SWZ_Krylov method setup %f seconds.\n", setup_end - setup_start);

    precond prec;
    prec.data = &SWZ_data;
    prec.fct  = fasp_precond_swz;

    // solver part
    status = fasp_solver_dcsr_itsolver(A,b,x,&prec,itparam);

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&solve_end);
        fasp_cputime("SWZ_Krylov method totally", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    fasp_swz_data_free(&SWZ_data);

    return status;
}

INT fasp_solver_dcsr_krylov_amg (dCSRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 ITS_param  *itparam,
                                 AMG_param  *amgparam)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT max_levels = amgparam->max_levels;
    const INT nnz = A->nnz, m = A->row, n = A->col;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end;

#if MULTI_COLOR_ORDER
```

```
    A->color = 0;
    A->IC = NULL;
    A->ICMAP = NULL;
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // initialize A, b, x for mgl[0]
    AMG_data *mgl=fasp_amg_data_create(max_levels);
    mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
    mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);

    // setup preconditioner
    switch (amgparam->AMG_type) {

        case SA_AMG:  // Smoothed Aggregation AMG
            status = fasp_amg_setup_sa(mgl, amgparam); break;

        case UA_AMG:  // Unsmoothed Aggregation AMG
            status = fasp_amg_setup_ua(mgl, amgparam); break;

        default:  // Classical AMG
            status = fasp_amg_setup_rs(mgl, amgparam);

    }

#if DEBUG_MODE > 1
    fasp_mem_usage();
#endif

    if (status < 0) goto FINISHED;

    // setup preconditioner
    precond_data pcdata;
    fasp_param_amg_to_prec(&pcdata,amgparam);
    pcdata.max_levels = mgl[0].num_levels;
    pcdata.mgl_data = mgl;

    precond pc; pc.data = &pcdata;

    if (itparam->precond_type == PREC_FMG) {
        pc.fct = fasp_precond_famg; // Full AMG
    }
    else {
        switch (amgparam->cycle_type) {
            case AMLI_CYCLE:  // AMLI cycle
                pc.fct = fasp_precond_amli; break;
            case NL_AMLI_CYCLE:  // Nonlinear AMLI
                pc.fct = fasp_precond_namli; break;
            default:  // V,W-cycles or hybrid cycles
                pc.fct = fasp_precond_amg;
        }
    }

    // call iterative solver
    status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        fasp_cputime("AMG_Krylov method totally", solve_end - solve_start);
    }

FINISHED:
    fasp_amg_data_free(mgl, amgparam);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dcsr_krylov_ilu (dCSRmat     *A,
                                 dvector     *b,
                                 dvector     *x,
                                     ITS_param  *itparam,
                                     ILU_param  *iluparam)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end, solve_time;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // ILU setup for whole matrix
    ILU_data LU;
    if ( (status = fasp_ilu_dcsr_setup(A, &LU, iluparam)) < 0 ) goto FINISHED;

    // check iludata
    if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;

    // set preconditioner
    precond pc;
    pc.data = &LU;
    pc.fct  = fasp_precond_ilu;

    // call iterative solver
    status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        solve_time = solve_end - solve_start;

        switch (iluparam->ILU_type) {
            case ILUt:
                fasp_cputime("ILUt_Krylov method totally", solve_time);
                break;
            case ILUtp:
                fasp_cputime("ILUtp_Krylov method totally", solve_time);
                break;
            default:  // ILUk
                fasp_cputime("ILUk_Krylov method totally", solve_time);
        }
    }

FINISHED:
    fasp_ilu_data_free(&LU);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dcsr_krylov_ilu_M (dCSRmat    *A,
                                   dvector    *b,
                                   dvector    *x,
                                   ITS_param  *itparam,
                                   ILU_param  *iluparam,
                                   dCSRmat    *M)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    REAL solve_start, solve_end, solve_time;
    INT status = FASP_SUCCESS;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // ILU setup for M
    ILU_data LU;
    if ( (status = fasp_ilu_dcsr_setup(M, &LU, iluparam)) < 0 ) goto FINISHED;

    // check iludata
    if ( (status = fasp_mem_iludata_check(&LU)) < 0 ) goto FINISHED;

    // set precondtioner
    precond pc;
    pc.data = &LU;
    pc.fct  = fasp_precond_ilu;

    // call iterative solver
    status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        solve_time = solve_end - solve_start;

        switch (iluparam->ILU_type) {
            case ILUt:
                fasp_cputime("ILUt_Krylov method", solve_time);
                break;
            case ILUtp:
                fasp_cputime("ILUtp_Krylov method", solve_time);
                break;
            default:  // ILUk
                fasp_cputime("ILUk_Krylov method", solve_time);
        }
    }

FINISHED:
    fasp_ilu_data_free(&LU);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

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)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT max_levels = amgparam->max_levels;
    const INT nnz = A->nnz, m = A->row, n = A->col;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end, solve_time;

#if MULTI_COLOR_ORDER
    A->color = 0;
    A->IC = NULL;
    A->ICMAP = NULL;
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: matrix size:  %d %d %d\n", A->row, A->col, A->nnz);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    // initialize A, b, x for mgl[0]
    AMG_data *mgl=fasp_amg_data_create(max_levels);
    mgl[0].A=fasp_dcsr_create(m,n,nnz); fasp_dcsr_cp(A,&mgl[0].A);
    mgl[0].b=fasp_dvec_create(n); mgl[0].x=fasp_dvec_create(n);

    // setup preconditioner
    switch (amgparam->AMG_type) {

        case SA_AMG:  // Smoothed Aggregation AMG
            status = fasp_amg_setup_sa(mgl, amgparam); break;

        case UA_AMG:  // Unsmoothed Aggregation AMG
            status = fasp_amg_setup_ua(mgl, amgparam); break;

        default:  // Classical AMG
            status = fasp_amg_setup_rs(mgl, amgparam);

    }

#if DEBUG_MODE > 1
    fasp_mem_usage();
#endif

    if (status < 0) goto FINISHED;

    // setup preconditioner
    precond_data pcdata;
    fasp_param_amg_to_prec(&pcdata,amgparam);
    pcdata.max_levels = mgl[0].num_levels;
    pcdata.mgl_data = mgl;

    // near kernel space
#if WITH_UMFPACK // use UMFPACK directly
    dCSRmat A_tran;
    A_tran = fasp_dcsr_create(A_nk->row, A_nk->col, A_nk->nnz);
    fasp_dcsr_transz(A_nk, NULL, &A_tran);
    // It is equivalent to do transpose and then sort
    //     fasp_dcsr_trans(A_nk, &A_tran);
    //     fasp_dcsr_sort(&A_tran);
    pcdata.A_nk = &A_tran;
#else
    pcdata.A_nk = A_nk;
#endif

    pcdata.P_nk = P_nk;
    pcdata.R_nk = R_nk;

    precond pc; pc.data = &pcdata;
    pc.fct = fasp_precond_amg_nk;

    // call iterative solver
    status = fasp_solver_dcsr_itsolver(A, b, x, &pc, itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        solve_time = solve_end - solve_start;
        fasp_cputime("AMG_NK_Krylov method", solve_time);
    }

FINISHED:
    fasp_amg_data_free(mgl, amgparam);

#if WITH_UMFPACK // use UMFPACK directly
    fasp_dcsr_free(&A_tran);
#endif

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.183 SolFAMG.c File Reference

Full AMG method as an iterative solver.

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

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

A	Pointer to dCSRmat: the coefficient matrix
b	Pointer to dvector: the right hand side
x	Pointer to dvector: the unknowns
param	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>
```

```

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

void fasp_solver_famg (const dCSRmat  *A,
                       const dvector  *b,
                       dvector        *x,
                       AMG_param      *param)
{
    const SHORT   max_levels  = param->max_levels;
    const SHORT   prtlvl      = param->print_level;
    const SHORT   amg_type    = param->AMG_type;
    const INT     nnz = A->nnz, m = A->row, n = A->col;

    // local variables
    AMG_data *    mgl = fasp_amg_data_create(max_levels);
    REAL          FMG_start = 0, FMG_end;

#if DEBUG_MODE > 0
    printf("###DEBUG: %s ......  [begin]\n", __FUNCTION__);
    printf("###DEBUG: nr=%d, nc=%d, nnz=%d\n", m, n, nnz);
#endif

    if ( prtlvl > PRINT_NONE ) fasp_gettime(&FMG_start);

    // check matrix data
    fasp_check_dCSRmat(A);

    // Step 0:  initialize mgl[0] with A, b and x
    mgl[0].A = fasp_dcsr_create(m,n,nnz);
    fasp_dcsr_cp(A,&mgl[0].A);

    mgl[0].b = fasp_dvec_create(n);
    fasp_dvec_cp(b,&mgl[0].b);

    mgl[0].x = fasp_dvec_create(n);
    fasp_dvec_cp(x,&mgl[0].x);

    // Step 1:  AMG setup phase
    switch (amg_type) {

        case SA_AMG:
            // Smoothed Aggregation AMG setup phase
            fasp_amg_setup_sa(mgl, param); break;

        case UA_AMG:
            // Unsmoothed Aggregation AMG setup phase
            fasp_amg_setup_ua(mgl, param); break;

        default:
            // Classical AMG setup phase
            fasp_amg_setup_rs(mgl, param); break;

    }

    // Step 2:  FAMG solve phase
    fasp_famg_solve(mgl, param);

    // Step 3:  Save solution vector and return
    fasp_dvec_cp(&mgl[0].x, x);

    // clean-up memory
    fasp_amg_data_free(mgl, param);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&FMG_end);
        fasp_cputime("FAMG totally", FMG_end - FMG_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

```

```
/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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.

Definition in file SolGMGPoisson.c.

9.185.2 Function Documentation

9.185.2.1 fasp_poisson_fgmg1d()

```
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)

Parameters

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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_fgmg2d()

```
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)

Parameters

u	Pointer to the vector of dofs

Parameters

b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	Number of grids in Y direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	NUmber of grids in y direction
nz	NUmber of grids in z direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	Number of grids in y direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	Number of grids in y direction
nz	Number of grids in z direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	Number of grids in y direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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

u	Pointer to the vector of dofs
b	Pointer to the vector of right hand side
nx	Number of grids in x direction
ny	Number of grids in y direction
nz	Number of grids in z direction
maxlevel	Maximum levels of the multigrid
rtol	Relative tolerance to judge convergence
prtlvl	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.

```

#include <time.h>
#include <math.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "PreGMG.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_poisson_gmg1d (REAL         *u,
                        REAL         *b,
                        const INT     nx,
                        const INT     maxlevel,
                        const REAL    rtol,
                        const SHORT   prtlvl)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL      *u0, *r0, *b0;
    REAL       norm_r, norm_r0, norm_r1, factor, error = BIGREAL;
    INT        i, *level, count = 0;
    REAL       AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", nx+1);
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = nx+1;
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(level[i]-level[i-1]+1)/2;
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));

    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);

    fasp_darray_cp(nx, u, u0);
    fasp_darray_cp(nx, b, b0);

    // compute initial l2 norm of residue
    fasp_darray_set(level[1], r0, 0.0);
    residual1d(u0, b0, r0, 0, level);
    norm_r0 = l2norm(r0, level, 0);
    norm_r1 = norm_r0;
    if (norm_r0 < atol) goto FINISHED;

    if ( prtlvl > PRINT_SOME ){
        printf("-----------------------------------------------------------\n");
        printf("It Num |   ||r||/||b||   |     ||r||      |  Conv.  Factor\n");
        printf("-----------------------------------------------------------\n");
    }

    // GMG solver of V-cycle
    while (count < max_itr_num) {
        count++;
        mg1d(u0, b0, level, 0, maxlevel);
        residual1d(u0, b0, r0, 0, level);
        norm_r = l2norm(r0, level, 0);
        factor = norm_r/norm_r1;
        error = norm_r / norm_r0;
        norm_r1 = norm_r;
        if ( prtlvl > PRINT_SOME ){
            printf("%6d | %13.6e   | %13.6e  | %10.4f\n",count,error,norm_r,factor);
        }
        if (error < rtol || norm_r < atol) break;
    }

    if ( prtlvl > PRINT_NONE ){
        if (count >= max_itr_num) {
            printf("### WARNING: V-cycle failed to converge.\n");
        }
        else {
            printf("Num of V-cycle's:  %d, Relative Residual = %e.\n", count, error);
        }
    }

    // Update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG totally", AMG_end - AMG_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

FINISHED:
    free(level);
    free(r0);
    free(u0);
    free(b0);

    return count;
}

INT fasp_poisson_gmg2d (REAL         *u,
                        REAL         *b,
                        const INT     nx,
                        const INT     ny,
                        const INT     maxlevel,
                        const REAL    rtol,
                        const SHORT   prtlvl)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL *u0, *b0, *r0;
    REAL norm_r, norm_r0, norm_r1, factor, error = BIGREAL;
    INT i, k, count = 0, *nxk, *nyk, *level;
    REAL AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
```

```
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
    }

    // set nxk, nyk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));
    nxk[0] = nx+1; nyk[0] = ny+1;
    for (k=1;k<maxlevel;k++) {
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0
    u0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel+1]*sizeof(REAL));

    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);

    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    residual2d(u0, b0, r0, 0, level, nxk, nyk);
    norm_r0 = l2norm(r0, level, 0);
    norm_r1 = norm_r0;
    if (norm_r0 < atol) goto FINISHED;

    if ( prtlvl > PRINT_SOME ){
        printf("-----------------------------------------------------------\n");
        printf("It Num |   ||r||/||b||   |     ||r||      |  Conv. Factor\n");
        printf("-----------------------------------------------------------\n");
    }

    // GMG solver of V-cycle
    while ( count < max_itr_num ) {
        count++;
        mg2d(u0, b0, level, 0, maxlevel, nxk, nyk);
        residual2d(u0, b0, r0, 0, level, nxk, nyk);
        norm_r = l2norm(r0, level, 0);
        error = norm_r / norm_r0;
        factor = norm_r/norm_r1;
        norm_r1 = norm_r;
        if ( prtlvl > PRINT_SOME ){
            printf("%6d | %13.6e   | %13.6e  | %10.4f\n",count,error,norm_r,factor);
        }
        if ( error < rtol || norm_r < atol ) break;
    }

    if ( prtlvl > PRINT_NONE ){
        if (count >= max_itr_num) {
            printf("### WARNING: V-cycle failed to converge.\n");
        }
        else {
            printf("Num of V-cycle's:  %d, Relative Residual = %e.\n", count, error);
        }
    }

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG totally", AMG_end - AMG_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(u0);
    free(b0);
    free(r0);

    return count;
}

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)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL       *u0, *r0, *b0;
    REAL       norm_r,norm_r0,norm_r1, factor, error = BIGREAL;
    INT        i, k, count = 0, *nxk, *nyk, *nzk, *level;
    REAL       AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
           nx, ny, nz, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
    }

    // set nxk, nyk, nzk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));
    nzk = (INT *)malloc(maxlevel*sizeof(INT));
    nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
    for(k=1;k<maxlevel;k++){
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
        nzk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
    norm_r0 = l2norm(r0, level, 0);
    norm_r1 = norm_r0;
    if (norm_r0 < atol) goto FINISHED;

    if ( prtlvl > PRINT_SOME ){
        printf("-----------------------------------------------------------\n");
        printf("It Num |   ||r||/||b||   |     ||r||      |  Conv.  Factor\n");
        printf("-----------------------------------------------------------\n");
    }

    // GMG solver of V-cycle
    while (count < max_itr_num) {
        count++;
        mg3d(u0, b0, level, 0, maxlevel, nxk, nyk, nzk);
        residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
        norm_r = l2norm(r0, level, 0);
        factor = norm_r/norm_r1;
        error = norm_r / norm_r0;
        norm_r1 = norm_r;
        if ( prtlvl > PRINT_SOME ){
            printf("%6d | %13.6e   | %13.6e  | %10.4f\n",count,error,norm_r,factor);
        }
        if (error < rtol || norm_r < atol) break;
    }

    if ( prtlvl > PRINT_NONE ){
        if (count >= max_itr_num) {
            printf("### WARNING: V-cycle failed to converge.\n");
        }
        else {
            printf("Num of V-cycle's:  %d, Relative Residual = %e.\n", count, error);
        }
    }

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG totally", AMG_end - AMG_start);
    }

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(nzk);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return count;
}

void fasp_poisson_fgmg1d (REAL         *u,
                          REAL         *b,
                          const INT     nx,
                          const INT     maxlevel,
                          const REAL    rtol,
                          const SHORT   prtlvl)
{
    const REAL  atol = 1.0E-15;
    REAL        *u0, *r0, *b0;
    REAL        norm_r0, norm_r;
    INT         *level;
    REAL        AMG_start = 0, AMG_end;
    int         i;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1));
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = nx+1;
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(level[i]-level[i-1]+1)/2;
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(nx, u, u0);
    fasp_darray_cp(nx, b, b0);

    // compute initial l2 norm of residue
    fasp_darray_set(level[1], r0, 0.0);
    residual1d(u0, b0, r0, 0, level);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    //  Full GMG solver
    fmg1d(u0, b0, level, maxlevel, nx);

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out Relative Residual and CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("FGMG totally", AMG_end - AMG_start);
        residual1d(u0, b0, r0, 0, level);
        norm_r = l2norm(r0, level, 0);
        printf("Relative Residual = %e.\n", norm_r/norm_r0);
    }

FINISHED:
    free(level);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

void fasp_poisson_fgmg2d (REAL         *u,
                          REAL         *b,
                          const INT     nx,
                          const INT     ny,
                          const INT     maxlevel,
                          const REAL    rtol,
                          const SHORT   prtlvl)
{
    const REAL atol = 1.0E-15;
    REAL       *u0, *r0, *b0;
    REAL       norm_r0, norm_r;
    INT        *nxk, *nyk, *level;
    int        i, k;
    REAL       AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
    }

    // set nxk, nyk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));

    nxk[0] = nx+1; nyk[0] = ny+1;
    for(k=1;k<maxlevel;k++) {
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel] + 1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    fasp_darray_set(level[1], r0, 0.0);
    residual2d(u0, b0, r0, 0, level, nxk, nyk);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    // FMG solver
    fmg2d(u0, b0, level, maxlevel, nxk, nyk);

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out Relative Residual and CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("FGMG totally", AMG_end - AMG_start);
        residual2d(u0, b0, r0, 0, level, nxk, nyk);
        norm_r = l2norm(r0, level, 0);
        printf("Relative Residual = %e.\n", norm_r/norm_r0);
    }

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

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)
{
    const REAL  atol = 1.0E-15;
    REAL        *u0, *r0, *b0;
    REAL        norm_r0, norm_r;
    INT         *nxk, *nyk, *nzk, *level;
    int         i, k;
    REAL        AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
           nx, ny, nz, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
    }
    // set nxk, nyk, nzk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));
    nzk = (INT *)malloc(maxlevel*sizeof(INT));

    nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
    for(k=1;k<maxlevel;k++){
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
        nzk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    // FMG
    fmg3d(u0, b0, level, maxlevel, nxk, nyk, nzk);

    // update u
    fasp_darray_cp(level[1], u0, u);

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("FGMG totally", AMG_end - AMG_start);
        residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
        norm_r = l2norm(r0, level, 0);
        printf("Relative Residual = %e.\n", norm_r/norm_r0);
    }

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(nzk);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return;
}

INT fasp_poisson_gmgcg1d (REAL         *u,
                          REAL         *b,
                          const INT     nx,
                          const INT     maxlevel,
                          const REAL    rtol,
                          const SHORT   prtlvl)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL        *u0, *r0, *b0;
    REAL        norm_r0;
    INT         *level;
    int         i, iter = 0;
    REAL        AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, maxlevel=%d\n", nx, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1));
    }
    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = nx+1;
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(level[i]-level[i-1]+1)/2;
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(nx, u, u0);
    fasp_darray_cp(nx, b, b0);

    // compute initial l2 norm of residue
    fasp_darray_set(level[1], r0, 0.0);
    residual1d(u, b, r0, 0, level);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    // Preconditioned CG method
    iter = pcg1d(u0, b0, level, maxlevel, nx, rtol, max_itr_num, prtlvl);

    // Update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
    }

FINISHED:
    free(level);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_poisson_gmgcg2d (REAL         *u,
                          REAL         *b,
                          const INT     nx,
                          const INT     ny,
                          const INT     maxlevel,
                          const REAL    rtol,
                          const SHORT   prtlvl)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL        *u0,*r0,*b0;
    REAL        norm_r0;
    INT         *nxk, *nyk, *level;
    int         i, k, iter = 0;
    REAL        AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, maxlevel=%d\n", nx, ny, maxlevel);
#endif
```

```

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1));
    }
    // set nxk, nyk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));

    nxk[0] = nx+1; nyk[0] = ny+1;
    for (k=1;k<maxlevel;k++) {
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0, r0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    fasp_darray_set(level[1], r0, 0.0);
    residual2d(u0, b0, r0, 0, level, nxk, nyk);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    // Preconditioned CG method
    iter = pcg2d(u0, b0, level, maxlevel, nxk,
                 nyk, rtol, max_itr_num, prtlvl);

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
    }

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

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)
{
    const REAL atol = 1.0E-15;
    const INT  max_itr_num = 100;

    REAL       *u0,*r0,*b0;
    REAL       norm_r0;
    INT        *nxk, *nyk, *nzk, *level;
    int        i, k, iter = 0;
    REAL       AMG_start = 0, AMG_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: nx=%d, ny=%d, nz=%d, maxlevel=%d\n",
           nx, ny, nz, maxlevel);
#endif

    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_start);
        printf("Num of DOF's:  %d\n", (nx+1)*(ny+1)*(nz+1));
    }

    // set nxk, nyk, nzk
    nxk = (INT *)malloc(maxlevel*sizeof(INT));
    nyk = (INT *)malloc(maxlevel*sizeof(INT));
    nzk = (INT *)malloc(maxlevel*sizeof(INT));

    nxk[0] = nx+1; nyk[0] = ny+1; nzk[0] = nz+1;
    for (k = 1; k < maxlevel; k++ ) {
        nxk[k] = (int) (nxk[k-1]+1)/2;
        nyk[k] = (int) (nyk[k-1]+1)/2;
        nzk[k] = (int) (nyk[k-1]+1)/2;
    }

    // set level
    level = (INT *)malloc((maxlevel+2)*sizeof(INT));
    level[0] = 0; level[1] = (nx+1)*(ny+1)*(nz+1);
    for (i = 1; i < maxlevel; i++) {
        level[i+1] = level[i]+(nx/pow(2.0,i)+1)*(ny/pow(2.0,i)+1)*(nz/pow(2.0,i)+1);
    }
    level[maxlevel+1] = level[maxlevel]+1;

    // set u0, b0
    u0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    b0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    r0 = (REAL *)malloc(level[maxlevel]*sizeof(REAL));
    fasp_darray_set(level[maxlevel], u0, 0.0);
    fasp_darray_set(level[maxlevel], b0, 0.0);
    fasp_darray_set(level[maxlevel], r0, 0.0);
    fasp_darray_cp(level[1], u, u0);
    fasp_darray_cp(level[1], b, b0);

    // compute initial l2 norm of residue
    residual3d(u0, b0, r0, 0, level, nxk, nyk, nzk);
    norm_r0 = l2norm(r0, level, 0);
    if (norm_r0 < atol) goto FINISHED;

    // Preconditioned CG method
    iter = pcg3d(u0, b0, level, maxlevel, nxk,
                 nyk, nzk, rtol, max_itr_num, prtlvl);

    // update u
    fasp_darray_cp(level[1], u0, u);

    // print out CPU time if needed
    if ( prtlvl > PRINT_NONE ) {
        fasp_gettime(&AMG_end);
        fasp_cputime("GMG_PCG totally", AMG_end - AMG_start);
    }

FINISHED:
    free(level);
    free(nxk);
    free(nyk);
    free(nzk);
    free(r0);
    free(u0);
    free(b0);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

/*---------------------------------*/
/*--        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) itsovlers.

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

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,
            precond * pc,
            ITS_param * itparam )
```

Solve Ax=b by preconditioned Krylov methods for CSR matrices.

Note

This is an abstract interface for iterative methods.

Parameters

mf	Pointer to mxv_matfree MatFree spmv operation
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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

mf	Pointer to mxv_matfree MatFree spmv operation
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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) itsovlers.

Parameters

matrix_format	matrix format
mf	Pointer to mxv_matfree MatFree spmv operation
A	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.

```
#include <time.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#include "fasp.h"
#include "fasp_functs.h"
#include "fasp_block.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"
#include "BlaSpmvMatFree.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_itsolver (mxv_matfree  *mf,
                          dvector      *b,
                          dvector      *x,
                          precond      *pc,
                          ITS_param    *itparam)
{
    const SHORT prtlvl        = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type     = itparam->stop_type;
    const INT   restart       = itparam->restart;
    const INT   MaxIt         = itparam->maxit;
    const REAL  tol           = itparam->tol;

    /* Local Variables */
    REAL solve_start, solve_end, solve_time;
    INT iter = ERROR_SOLVER_TYPE;

    fasp_gettime(&solve_start);

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    /* Choose a desirable Krylov iterative solver */
    switch ( itsolver_type ) {

        case SOLVER_CG:
            iter = fasp_solver_pcg(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_BiCGstab:
            iter = fasp_solver_pbcgs(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_MinRes:
            iter = fasp_solver_pminres(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter = fasp_solver_pgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter = fasp_solver_pvgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VFGMRES:
            iter = fasp_solver_pvfgmres(mf, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_GCG:
            iter = fasp_solver_pgcg(mf, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl >= PRINT_SOME) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        solve_time = solve_end - solve_start;
        fasp_cputime("Iterative method", solve_time);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_krylov (mxv_matfree  *mf,
                        dvector      *b,
                        dvector      *x,
                        ITS_param    *itparam)
{
    const SHORT prtlvl = itparam->print_level;

    /* Local Variables */
    INT      status = FASP_SUCCESS;
    REAL     solve_start, solve_end, solve_time;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    status = fasp_solver_itsolver(mf,b,x,NULL,itparam);

    if ( prtlvl >= PRINT_MIN ) {
        fasp_gettime(&solve_end);
        solve_time = solve_end - solve_start;
        fasp_cputime("Krylov method totally", solve_time);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

void fasp_solver_matfree_init (INT           matrix_format,
                               mxv_matfree  *mf,
                               void         *A)
{
    switch ( matrix_format ) {

        case MAT_CSR:
            mf->fct = fasp_blas_mxv_csr;
            break;

        case MAT_BSR:
            mf->fct = fasp_blas_mxv_bsr;
            break;

        case MAT_STR:
            mf->fct = fasp_blas_mxv_str;
            break;

        case MAT_BLC:
            mf->fct = fasp_blas_mxv_blc;
            break;

        case MAT_CSRL:
            mf->fct = fasp_blas_mxv_csrl;
            break;

        default:
            printf("### ERROR: Unknown matrix format %d!\n", matrix_format);
            exit(ERROR_DATA_STRUCTURE);

    }

    mf->data = A;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

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, BlaILUSetupSTR.c, BlaSparseSTR.c, ItrSmootherSTR.c, KryPbcgs.c, KryPcg.c, KryPgmres.c, KryPvgmres.c, and PreSTR.c

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,
            precond * pc,
            ITS_param * itparam )
```

Solve Ax=b by standard Krylov methods.

Parameters

A	Pointer to the coeff matrix in dSTRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
pc	Pointer to the preconditioning action
itparam	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

A	Pointer to the coeff matrix in dSTRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dSTRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format

Parameters

itparam	Pointer to parameters for iterative solvers
neigh	Pointer to neighbor vector
order	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

A	Pointer to the coeff matrix in dSTRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	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

A	Pointer to the coeff matrix in dSTRmat format
b	Pointer to the right hand side in dvector format
x	Pointer to the approx solution in dvector format
itparam	Pointer to parameters for iterative solvers
iluparam	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.

```

#include <math.h>
#include <time.h>

#include "fasp.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--  Declare Private Functions  --*/
/*---------------------------------*/

#include "KryUtil.inl"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

INT fasp_solver_dstr_itsolver (dSTRmat    *A,
                               dvector    *b,
                               dvector    *x,
                               precond    *pc,
                               ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    const SHORT itsolver_type = itparam->itsolver_type;
    const SHORT stop_type = itparam->stop_type;
    const SHORT restart = itparam->restart;
    const INT   MaxIt = itparam->maxit;
    const REAL  tol = itparam->tol;

    // local variables
    INT iter = ERROR_SOLVER_TYPE;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
    printf("### DEBUG: rhs/sol size:  %d %d\n", b->row, x->row);
#endif

    fasp_gettime(&solve_start);

    /* Safe-guard checks on parameters */
    ITS_CHECK ( MaxIt, tol );

    switch (itsolver_type) {

        case SOLVER_CG:
            iter=fasp_solver_dstr_pcg(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_BiCGstab:
            iter=fasp_solver_dstr_pbcgs(A, b, x, pc, tol, MaxIt, stop_type, prtlvl);
            break;

        case SOLVER_GMRES:
            iter=fasp_solver_dstr_pgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        case SOLVER_VGMRES:
            iter=fasp_solver_dstr_pvgmres(A, b, x, pc, tol, MaxIt, restart, stop_type, prtlvl);
            break;

        default:
            printf("### ERROR: Unknown iterative solver type %d!  [%s]\n",
                   itsolver_type, __FUNCTION__);
            return ERROR_SOLVER_TYPE;

    }

    if ( (prtlvl > PRINT_MIN) && (iter >= 0) ) {
        fasp_gettime(&solve_end);
        fasp_cputime("Iterative method", solve_end - solve_start);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return iter;
}

INT fasp_solver_dstr_krylov (dSTRmat    *A,
                             dvector    *b,
                             dvector    *x,
                             ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    INT status = FASP_SUCCESS;
    REAL solve_start, solve_end;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // solver part
    fasp_gettime(&solve_start);

    status=fasp_solver_dstr_itsolver(A,b,x,NULL,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Krylov method totally", solve_end - solve_start);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
```

```
}

INT fasp_solver_dstr_krylov_diag (dSTRmat    *A,
                                  dvector    *b,
                                  dvector    *x,
                                  ITS_param  *itparam)
{
    const SHORT prtlvl = itparam->print_level;
    const INT ngrid = A->ngrid;

    INT status = FASP_SUCCESS;
    REAL solve_start, solve_end;
    INT nc = A->nc, nc2 = nc*nc, i;

    fasp_gettime(&solve_start);

    // setup preconditioner
    precond_diag_str diag;
    fasp_dvec_alloc(ngrid*nc2, &(diag.diag));
    fasp_darray_cp(ngrid*nc2, A->diag, diag.diag.val);

    diag.nc = nc;

    for (i=0;i<ngrid;++i) fasp_smat_inv(&(diag.diag.val[i*nc2]),nc);

    precond *pc = (precond *)fasp_mem_calloc(1,sizeof(precond));

    pc->data = &diag;
    pc->fct  = fasp_precond_dstr_diag;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // solver part
    status=fasp_solver_dstr_itsolver(A,b,x,pc,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN )
        fasp_cputime("Diag_Krylov method totally", solve_end - solve_start);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dstr_krylov_ilu (dSTRmat    *A,
                                 dvector    *b,
                                 dvector    *x,
                                 ITS_param  *itparam,
                                 ILU_param  *iluparam)
{
    const SHORT prtlvl = itparam->print_level;
    const INT ILU_lfil = iluparam->ILU_lfil;

    INT status = FASP_SUCCESS;
    REAL setup_start, setup_end, setup_time;
    REAL solve_start, solve_end, solve_time;

    //set up
    dSTRmat LU;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    fasp_gettime(&setup_start);

    if (ILU_lfil == 0) {
        fasp_ilu_dstr_setup0(A,&LU);
    }
    else if (ILU_lfil == 1) {
        fasp_ilu_dstr_setup1(A,&LU);
    }
    else  {
        printf("### ERROR: Illegal level of ILU fill-in (>1)!  [%s]\n", __FUNCTION__);
        return ERROR_MISC;
    }

    fasp_gettime(&setup_end);

    setup_time = setup_end - setup_start;

    if ( prtlvl > PRINT_NONE )
        printf("Structrued ILU(%d) setup costs %f seconds.\n", ILU_lfil, setup_time);

    precond pc; pc.data=&LU;
    if (ILU_lfil == 0) {
        pc.fct = fasp_precond_dstr_ilu0;
    }
    else if (ILU_lfil == 1) {
        pc.fct = fasp_precond_dstr_ilu1;
    }
    else {
        printf("### ERROR: Illegal level of ILU fill-in (>1)!  [%s]\n", __FUNCTION__);
        return ERROR_MISC;
    }

    // solver part
    fasp_gettime(&solve_start);

    status=fasp_solver_dstr_itsolver(A,b,x,&pc,itparam);

    fasp_gettime(&solve_end);

    if ( prtlvl >= PRINT_MIN ) {
        solve_time = solve_end - solve_start;
        printf("Iterative solver costs %f seconds.\n", solve_time);
        fasp_cputime("ILU_Krylov method totally", setup_time+solve_time);
    }

    fasp_dstr_free(&LU);

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

INT fasp_solver_dstr_krylov_blockgs (dSTRmat    *A,
                                     dvector    *b,
                                     dvector    *x,
                                     ITS_param  *itparam,
                                     ivector    *neigh,
                                     ivector    *order)
{
    // Parameter for iterative method
    const SHORT prtlvl = itparam->print_level;

    // Information about matrices
    INT ngrid = A->ngrid;

    // return parameter
    INT status = FASP_SUCCESS;

    // local parameter
    REAL setup_start, setup_end, setup_time = 0;
    REAL solve_start, solve_end, solve_time = 0;

    dvector *diaginv;
    ivector *pivot;

#if DEBUG_MODE > 0
    printf("### DEBUG: [-Begin-] %s ...\n", __FUNCTION__);
#endif

    // setup preconditioner
    fasp_gettime(&setup_start);

    diaginv = (dvector *)fasp_mem_calloc(ngrid, sizeof(dvector));
    pivot = (ivector *)fasp_mem_calloc(ngrid, sizeof(ivector));
    fasp_generate_diaginv_block(A, neigh, diaginv, pivot);

    precond_data_str pcdata;
    pcdata.A_str = A;
    pcdata.diaginv = diaginv;
    pcdata.pivot = pivot;
    pcdata.order = order;
    pcdata.neigh = neigh;

    precond pc; pc.data = &pcdata; pc.fct = fasp_precond_dstr_blockgs;

    fasp_gettime(&setup_end);

    if ( prtlvl > PRINT_NONE ) {
        setup_time = setup_end - setup_start;
        printf("Preconditioner setup costs %f seconds.\n", setup_time);
    }

    // solver part
    fasp_gettime(&solve_start);

    status = fasp_solver_dstr_itsolver(A,b,x,&pc,itparam);

    fasp_gettime(&solve_end);
    solve_time = solve_end - solve_start;

    if ( prtlvl >= PRINT_MIN ) {
        fasp_cputime("Iterative solver", solve_time);
        fasp_cputime("BlockGS_Krylov method totally", setup_time + solve_time);
    }

#if DEBUG_MODE > 0
    printf("### DEBUG: [--End--] %s ...\n", __FUNCTION__);
#endif

    return status;
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

9.191 SolWrapper.c File Reference

Wrappers for accessing functions by advanced users.

```
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.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.

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

n	Number of cols of A
nnz	Number of nonzeros of A
nb	Size of each small block
ia	IA of A in CSR format
ja	JA of A in CSR format
a	VAL of A in CSR format
b	RHS vector
u	Solution vector
tol	Tolerance for iterative solvers
maxit	Max number of iterations
ptrlvl	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

n	Number of cols of A
nnz	Number of nonzeros of A
nb	Size of each small block
ia	IA of A in BSR format
ja	JA of A in BSR format
a	VAL of A in BSR format
b	RHS vector
u	Solution vector
tol	Tolerance for iterative solvers
maxit	Max number of iterations
ptrlvl	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

n	Number of cols of A
nnz	Number of nonzeros of A
ia	IA of A in CSR format
ja	JA of A in CSR format
a	VAL of A in CSR format
b	RHS vector
u	Solution vector
tol	Tolerance for iterative solvers
maxit	Max number of iterations
ptrlvl	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

n	Number of cols of A
nnz	Number of nonzeros of A

Parameters

ia	IA of A in CSR format
ja	JA of A in CSR format
a	VAL of A in CSR format
b	RHS vector
u	Solution vector
tol	Tolerance for iterative solvers
maxit	Max number of iterations
ptrlvl	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

n	Number of cols of A
nnz	Number of nonzeros of A
ia	IA of A in CSR format
ja	JA of A in CSR format
a	VAL of A in CSR format
b	RHS vector
u	Solution vector
tol	Tolerance for iterative solvers
maxit	Max number of iterations
ptrlvl	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

n	Number of cols of A
nnz	Number of nonzeros of A
ia	IA of A in CSR format
ja	JA of A in CSR format
a	VAL of A in CSR format
b	RHS vector
u	Solution vector
ptrlvl	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.

```
#include "fasp.h"
#include "fasp_block.h"
#include "fasp_functs.h"

/*---------------------------------*/
/*--      Public Functions       --*/
/*---------------------------------*/

```

```
void fasp_fwrapper_dcsr_pardiso_(INT* n, INT* nnz, INT* ia, INT* ja, REAL* a, REAL* b,
                                 REAL* u, INT* ptrlvl)
{
    dCSRmat mat;      // coefficient matrix
    dvector rhs, sol; // right-hand-side, solution

    // set up coefficient matrix
    mat.row = *n;
    mat.col = *n;
    mat.nnz = *nnz;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n;
    rhs.val = b;
    sol.row = *n;
    sol.val = u;

    fasp_dcsr_sort(&mat);

    fasp_solver_pardiso(&mat, &rhs, &sol, *ptrlvl);
}

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)
{
    dCSRmat   mat;      // coefficient matrix
    dvector   rhs, sol; // right-hand-side, solution
    AMG_param amgparam; // parameters for AMG

    // setup AMG parameters
    fasp_param_amg_init(&amgparam);

    amgparam.tol         = *tol;
    amgparam.print_level = *ptrlvl;
    amgparam.maxit       = *maxit;

    // set up coefficient matrix
    mat.row = *n;
    mat.col = *n;
    mat.nnz = *nnz;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n;
    rhs.val = b;
    sol.row = *n;
    sol.val = u;

    fasp_solver_amg(&mat, &rhs, &sol, &amgparam);
}

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)
{
    dCSRmat   mat;      // coefficient matrix
    dvector   rhs, sol; // right-hand-side, solution
    ILU_param iluparam; // parameters for ILU
    ITS_param itsparam; // parameters for itsolver

    // setup ILU parameters
    fasp_param_ilu_init(&iluparam);

    iluparam.print_level = *ptrlvl;

    // setup Krylov method parameters
    fasp_param_solver_init(&itsparam);

    itsparam.itsolver_type = SOLVER_VFGMRES;
    itsparam.tol           = *tol;
    itsparam.maxit         = *maxit;
    itsparam.print_level   = *ptrlvl;

    // set up coefficient matrix
    mat.row = *n;
    mat.col = *n;
    mat.nnz = *nnz;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n;
    rhs.val = b;
    sol.row = *n;
    sol.val = u;

    fasp_solver_dcsr_krylov_ilu(&mat, &rhs, &sol, &itsparam, &iluparam);
}

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)
{
    dCSRmat     mat;      // coefficient matrix
    dvector     rhs, sol; // right-hand-side, solution
    input_param inparam;  // parameters from input files
    AMG_param   amgparam; // parameters for AMG
    ITS_param   itsparam; // parameters for itsolver
    ILU_param   iluparam; // parameters for ILU

    char* inputfile = "ini/amg.dat"; // Added for fasp4ns 2022.04.08 --zcs
    fasp_param_input(inputfile, &inparam);
    fasp_param_init(&inparam, &itsparam, &amgparam, &iluparam, NULL);

    itsparam.tol         = *tol;
    itsparam.maxit       = *maxit;
    itsparam.print_level = *ptrlvl;

    // set up coefficient matrix
    mat.row = *n;
    mat.col = *n;
    mat.nnz = *nnz;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n;
    rhs.val = b;
    sol.row = *n;
    sol.val = u;

    fasp_solver_dcsr_krylov_amg(&mat, &rhs, &sol, &itsparam, &amgparam);
}

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)
{
    dBSRmat mat;      // coefficient matrix in BSR format
    dvector rhs, sol; // right-hand-side, solution

    ILU_param iluparam; // parameters for ILU
    ITS_param itsparam; // parameters for itsolver

    // setup ILU parameters
    fasp_param_ilu_init(&iluparam);
    iluparam.ILU_lfil    = 0;
    iluparam.print_level = *ptrlvl;

    // setup Krylov method parameters
    fasp_param_solver_init(&itsparam);

    itsparam.itsolver_type = SOLVER_VFGMRES;
    itsparam.tol           = *tol;
    itsparam.maxit         = *maxit;
    itsparam.print_level   = *ptrlvl;

    // set up coefficient matrix
    mat.ROW = *n;
    mat.COL = *n;
    mat.NNZ = *nnz;
    mat.nb  = *nb;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n * *nb;
    rhs.val = b;
    sol.row = *n * *nb;
    sol.val = u;

    // solve
    fasp_solver_dbsr_krylov_ilu(&mat, &rhs, &sol, &itsparam, &iluparam);
}

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)
{
    dBSRmat mat;      // coefficient matrix in CSR format
    dvector rhs, sol; // right-hand-side, solution

    AMG_param amgparam; // parameters for AMG
    ITS_param itsparam; // parameters for itsolver

    // setup AMG parameters
    fasp_param_amg_init(&amgparam);
    amgparam.AMG_type    = UA_AMG;
    amgparam.print_level = *ptrlvl;

    // setup Krylov method parameters
    fasp_param_solver_init(&itsparam);
    itsparam.tol           = *tol;
    itsparam.print_level   = *ptrlvl;
    itsparam.maxit         = *maxit;
    itsparam.itsolver_type = SOLVER_VFGMRES;

    // set up coefficient matrix
    mat.ROW = *n;
    mat.COL = *n;
    mat.NNZ = *nnz;
    mat.nb  = *nb;
    mat.IA  = ia;
    mat.JA  = ja;
    mat.val = a;

    rhs.row = *n * *nb;
    rhs.val = b;
    sol.row = *n * *nb;
    sol.val = u;

    // solve
    fasp_solver_dbsr_krylov_amg(&mat, &rhs, &sol, &itsparam, &amgparam);
}

/*---------------------------------*/
/*--        End of File          --*/
/*---------------------------------*/
```

Index