FASP User Guide

FASP Developer Team

Version 1.3.7

Contents

Chapter 1

Introduction

1.1 What is FASP

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

We believe that there is no one-size-for-all solution method for discrete linear systems from different applications. And, efficient iterative solvers can be constructed by taking the properties of partial differential equations (PDEs) and discretizations into account. In this project, we plan to construct a pool of discrete problems arising from systems of PDEs and efficient linear solvers for these problems. We mainly utilize the methodology of Auxiliary Space Preconditioning (ASP) [17] to construct efficient linear solvers. Due to this reason, this software package is called "Fast Auxiliary Space Preconditioning" or FASP for short.

Our goal

The FASP project is not a traditional software project; instead, it is designed to support our effort to identify efficient algorithms and to build fast solvers for a set of PDE problems—FASP is designed for developing and testing new efficient solvers and preconditioners for discrete partial differential equations (PDEs) or systems of PDEs. The main components of the package are basic linear iterative methods, standard Krylov methods, geometric and algebraic multigrid methods, and incomplete factorization methods. Based on these standard techniques, we build efficient solvers, based on the framework of Auxiliary Space Preconditioning, for several complicated applications. For the moment, we have a few examples include the fluid dynamics, underground water simulation, the black oil model in reservoir simulation, and so on.

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. We tried and will continue to try to keep as many parts of the FASP project open

to public as possible. However, some of the applications contain contributions from and owned partially by other parties. We only discuss the kernel functions (open to public) in this user's guide.

Our strategy

We organize the development of FASP package in a *"Multilevel"* or *"Capitalism"* way:

- Stage 1. Fine level stage (or free market stage)

  (1) Collect problems and solvers. Allow similarities or even duplications, for example same solution algorithm, but different implementation. Keep all the record: problem description, solver code, test results, etc.

  (2) Try to find a minimal set of standard or rules. And then we let the market to evolve freely. The idea is to allow the market to be FREE.

- Stage 2. Coarse level stage (or state capitalism stage)

  (1) As FASP evolves, we might see, at certain time, that the market is out-of-control. This basically means the "fine level solver" or the "free market" is very successful and we should start to give more strict standard or regulation.

  (2) Write a professional-level software package for a set of chosen algorithms for particular problems.

1.2 What solvers you are going to get

We are currently interested in the theory and numerical solution of many PDE problems. Currently, we are mainly working on solving the following PDEs and PDE systems (this is not a complete list and it is still expanding):

- Poisson equation
- Reaction-diffusion equation
- Linear elasticity
- Brinkman equation
- Biharmonic equation
- Stokes and Navier-Stokes equations

- Fluid-structure interaction
- Oldryod-B and Johnson-Seglman equations
- Darcy's flow
- Black oil model and its generalizations
- H(curl)/H(div) systems
- Maxwell equation
- MHD equation

We intend to design solution algorithms and their implementation for all these problems with different discretizations. We have done a few of them but not all of them are publicly available at this moment.

1.3 How to use this guide

In this user's guide, we mainly describe how to use the existing solvers in FASP via a couple of simple tutorial problems. This user's guide is self-contained but does *not* provide details of the algorithms nor the implementation. Along this guide, we provide a reference manual[1] for technical details of the implementation. For the algorithms implemented, we will provide the references and we recommend the users to read them for better understanding of the code. Furthermore, since FASP is under heavy development, use this guide with caution because the code might have been changed before this document is updated.

1.4 How to obtain FASP

All the FASP packages are hosted on *BitBucket.org*[2] using Mercurial (Hg)[3]. A Hg client for GNU Linux, Mac OS X, or Windows can be downloaded from

http://mercurial.selenic.com/downloads/

There are also many other third-party clients which provides Hg services, for example: EasyMercurial[4] (cross platform) and SourceTree[5] (for Mac OS X only).

As a DVCS (Distributed Version Control System) source-control software, Hg is relatively new. But compared with other tools like Git, Hg is considered *friendlier* with a lower learning curve. This is despite the fact that Hg uses two distinct sets of commands and two distinct vocabularies for operations depending upon whether the repository is local or remote. Documentation for Hg is substantially better, including a book[6]. They've also had the advantage of trying the documentation

[1] Available online at http://fasp.sourceforge.net. It is also available in "faspsolver/doc/doc.zip".
[2] Official website: https://bitbucket.org/
[3] Official website: http://mercurial.selenic.com/
[4] Official website: http://easyhg.org
[5] Official website: http://www.sourcetreeapp.com
[6] The hgbook, http://hgbook.red-bean.com/

on a fairly savvy group of developers (Mozilla) who gave them lots of feedback that helped polish the rough edges.

Linux or OS X

First, you need to obtain a free copy of FASP kernel functions from our public Hg repository. If you are downloading FASP for the first time, you can clone the repository to your local machine:

```
"Download FASP kernel subroutines via HTTPS"

$ hg clone https://faspusers@bitbucket.org/fasp/faspsolver
```

For the moment, it requires a password to access the Hg repository. You may request an access passcode by sending an email to faspdev@gmail.com. Very soon, the FASP package will be open to public completely.

After a long pause[7], you should have obtained "faspsolver" in your current directory successfully. If you have already cloned the repository before, you can just pull a new version and update your local version with it: Go to your local "faspsolver" directory and then

```
"Pull a new version from BitBucket"
$ hg pull

"Update you local version to the new version"
$ hg update
```

Windows

If you are using Windows, you may want to install TortoiseHg. After installing it, the TortoiseHg menu has been merged into the right-click menu of Windows Explore. You could download FASP copy from BitBucket.org. Choose "TortoiseHg" --> "Clone" in the pop-up menu, the source address is

```
https://faspusers@bitbucket.org/fasp/faspsovler
```

Then press "Clone" and you will obtain "faspsolver" in the directory you set.

1.5 How to build FASP

FASP has been tested on Linux (Cent OS, Debian, Fedora, RedHat, Ubuntu), OS X (Leopard, Snow Leopard, Lion) and Windows (XP, Win 7) with a couple of compliers including GCC, G++, ICC, VC++, GFORTRAN, G95, IFORT.

[7] In fact, a very long pause. This is because the initial clone with copy all the history data which is about 150MB in total. Depending on the speed of your network, it could take 15 minutes to one hour.

Linux or OS X

Now we give a simple instruction on how to compile FASP on Linux: To build the FASP library, just go to the "faspsolver" directory and type:

```
$ make config
$ make install
```

In order to make sure everything is OK, you can go to the "faspsolver/test" directory and try to run the test problem:

```
$ ./test.ex
```

If you need more help, you can use

```
$ make help
```

and you will get the following screen

```
=============================================================
||   Fast Auxiliary Space Preconditioners (FASP)    ||
=============================================================

$ make config           # Configure the building environment
$ make                  # Compile the library
$ make install          # install FASP libraries and related files
$ make docs             # Generate the FASP documentation with Doxygen
$ make headers          # Generate function decorations automatically
$ make uninstall        # Remove installed files by "make install"
$ make clean            # Remove obj files but retain configuration options
$ make distclean        # Clean and completely removes the build directory
$ make help             # Show this screen
```

To uninstall FASP and clean up the working directory, you can simply run

```
$ make uninstall
$ make distclean
```

To enable OpenMP support, you need to uncomment one line in "Makefile" and set "openmp" to be "yes".

```
# If you want to compile with OpenMP support, uncomment the next line:
#
# openmp=yes
```

Windows

We provide a Visual Studio 2008 (VS08) solution and a VS10 solution of FASP for Windows users. For example, you can just open "faspsolver/vs08/faspsolver-vs08.sln" if you are using VS08 as your default developing environment. Then a single-click at the "Build Solution" on the menu or "F7" key will give you all the FASP libraries and the test programs in "faspsolver/test/".

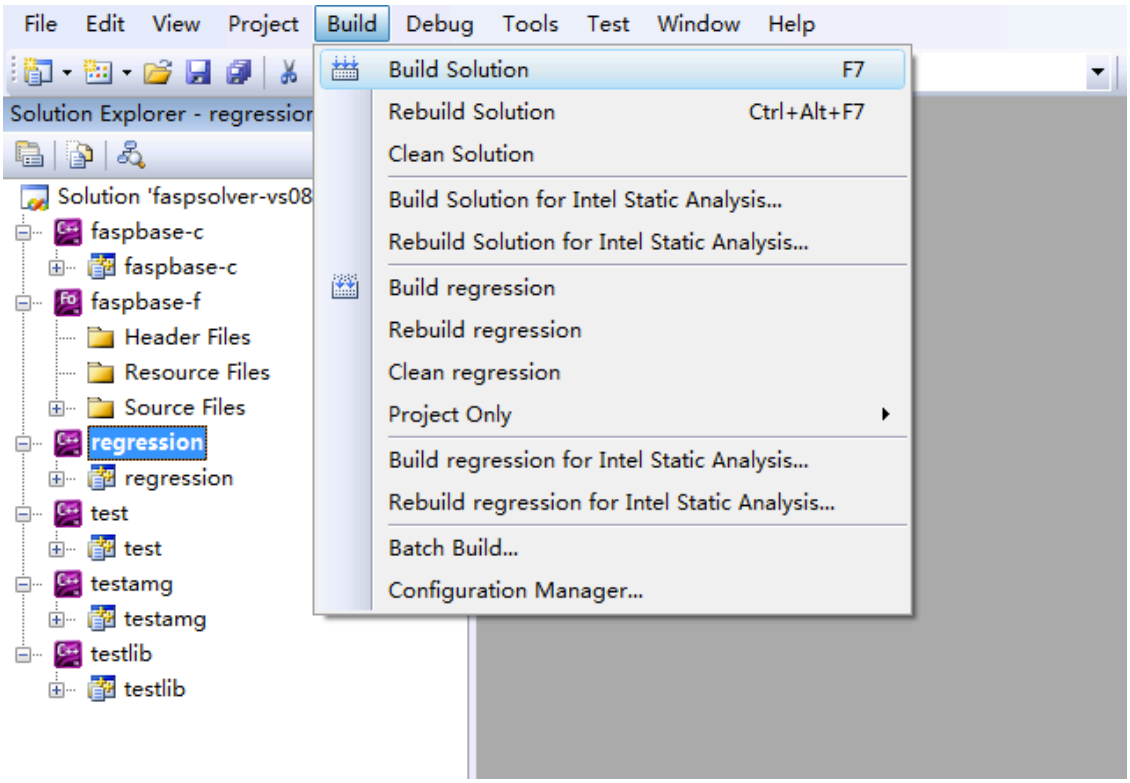

Figure 1.1: Build FASP using Visual Studio 2008.

You need a C/C++ complier and a Fortran compiler together with Visual Studio to build FASP. You can use either Microsoft Visual C++ or Intel C compiler, together with Intel Fortran compiler.

If you are using other versions of Visual Studio (like VS05 or VS12), do NOT convert the "VS08" solution file to your VS version because the FASP files might be cleaned up (removed) by Visual Studio automatically. You have to create another solution to build all the libraries and test programs by yourselves.

If you need to build a VS solution by yourselves, you should create 5 projects:

1. "faspbase-c" contains all the ".c" and ".inl" files in the directory "./base/src/". You should add "./base/include" in Additional Directories. This project contains the core subroutines of faspsolver.

2. "faspbase-f" contains all the ".f" files in "./base/src/" and "./base/extra/sparsekit".

3. "testlib" contains all the ".c" files in "./test/src/". You should add "./test/include" in Additional Directories.

4. "test" is an executing program for test purpose in FASP. The source file is "./test/main/test.c".

5. "regression" is another executing program, which contains several methods to test the problems. The source file is "./test/main/regression.c".

NOTE: If you are using Visual C++, all the C files should be compiled as C++ code (by using the /TP compiling option).

After you successfully build the solution, you will get two static libraries named "faspbase-c-vs08.lib" and "faspbase-f-vs08.lib". You can use the "lib" command to wrap together as one single file (e.g. FASP.lib) for better portability. For example:

```
C:\FASP> lib /ltcg /out:FASP.lib faspbase-c-vs08.lib faspbase-f-vs08.lib
```

Using GUI based on TCL

You can also try to build FASP using the TCL graphical user interface. For example, in Linux or Mac OS X, you may

```
$ wish FASP_install.tcl
```

A graphical interface will pop up and the rest of the building process is straightforward.

External libraries

There are a few *optional* external libraries that you might want to use, including memory allocation routines, direct solvers, ILU methods, discretization packages, etc. FASP has interfaces to a couple of them which we often use, for example, UMFPack, SuperLU, MUMPS, dlmalloc, SparseKit.

Chapter 2

A Tutorial

In this chapter, we use a couple simple examples to demonstrate how to use the FASP package for solving existing linear systems which have been saved as disk files. All the examples can be found in "faspsolver/tutorial/". Here we only discuss the C version of these examples; interested users can read the F90 version of some of the examples. After you successfully build FASP (see §1.5), just go to the "faspsolver/tutorial/" directory and the compiled tutorial examples should be ready to be tried.

2.1 The first example

The first example is the simplest one that we can imagine: We read the stiffness matrix A and right-hand side b from disk files; then we solve $Ax = b$ using the classical AMG method [1, 12, 13]; see §3.7. The stiffness matrix A is symmetric positive definite (SPD), arising from the continuous piecewise linear finite element discretization of the Poisson equation

$$-\Delta u = f$$

(with the Dirichlet boundary condition) on a simple quasi-uniform triangulation of the bounded domain Ω.

```
1  /*! \file poisson-amg.c
2   *
3   *  \brief The first test example for FASP: using AMG to solve
4   *         the discrete Poisson equation from P1 finite element.
5   *         C version.
6   *
7   *  \note  AMG example for FASP: C version
8   *
9   *  Solving the Poisson equation (P1 FEM) with AMG
10  */
11
12 #include "fasp.h"
13 #include "fasp_functs.h"
14
15 /**
16  * \fn int main (int argc, const char * argv[])
17  *
18  * \brief This is the main function for the first example.
19  *
```

```
 * \author Chensong Zhang
 * \date   12/21/2011
 *
 * Modified by Chensong Zhang on 09/22/2012
 */
int main (int argc, const char * argv[])
{
    input_param     inparam;  // parameters from input files
    AMG_param       amgparam; // parameters for AMG

    printf("\n=========================================");
    printf("\n||   FASP: AMG example -- C version   ||");
    printf("\n=========================================\n\n");

    // Step 0. Set parameters
    // Read input and AMG parameters from a disk file
    // In this example, we read everything from a disk file:
    //          "ini/amg.dat"
    // See the reference manual for details of the parameters.
    fasp_param_init("ini/amg.dat",&inparam,NULL,&amgparam,NULL,NULL);

    // Set local parameters using the input values
    const int print_level = inparam.print_level;

    // Step 1. Get stiffness matrix and right-hand side
    // Read A and b -- P1 FE discretization for Poisson. The location
    // of the data files is given in "ini/amg.dat".
    dCSRmat A;
    dvector b, x;
    char filename1[512], *datafile1;
    char filename2[512], *datafile2;

    // Read the stiffness matrix from matFE.dat
    strncpy(filename1,inparam.workdir,128);
    datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);

    // Read the RHS from rhsFE.dat
    strncpy(filename2,inparam.workdir,128);
    datafile2="rhs_FE.dat"; strcat(filename2,datafile2);

    fasp_dcsrvec2_read(filename1,filename2,&A,&b);

    // Step 2. Print problem size and AMG parameters
    if (print_level>PRINT_NONE) {
        printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
        printf("b: n = %d\n", b.row);
        fasp_param_amg_print(&amgparam);
    }

    // Step 3. Solve the system with AMG as an iterative solver
    // Set the initial guess to be zero and then solve it
    // with AMG method as an iterative procedure
    fasp_dvec_alloc(A.row, &x);
    fasp_dvec_set(A.row, &x, 0.0);

    fasp_solver_amg(&A, &b, &x, &amgparam);

    // Step 4. Clean up memory
    fasp_dcsr_free(&A);
    fasp_dvec_free(&b);
    fasp_dvec_free(&x);

    return SUCCESS;
}
```

```
84
85 /*----------------------------------*/
86 /*--        End of File           --*/
87 /*----------------------------------*/
```

Since this is the first example, we will explain it in some detail:

- Line 1 tells the Doxygen documentation system that the filename is "poisson-amg.c". Line 3–5 tells the Doxygen what is the purpose of this file (function).
- Line 12–13 includes the main FASP header file "fasp.h" and FASP function decoration header "fasp_functs.h". These two headers shall be included in all files that requires FASP subroutines. Please also be noted that the function decorations in "fasp_functs.h" is automatically generated from the source files and should NOT be modified by an enduser.
- Line 39 reads solver parameters from "tutorial/ini/amg.dat"; see more discussions in §2.4. In this disk file, we can set the location of the data files, type of solvers, maximal number of iteration numbers, convergence tolerance, and many other parameters for iterative solvers. Note that, in this function, we have initialize "amgparam" at the same time.
- Line 47 defines a sparse matrix A in the compressed sparse row (CSR) format. Line 48 defines two vectors: the right-hand side b and the numerical solution x. We refer to §3.1 for definitions of vectors and general sparse matrices.
- Line 60 reads the matrix and the right-hand side from two disk files. Line 49–58 defines the filenames of them.
- Line 63–67 prints basic information of coefficient matrix, right-hand side, and solver parameters.
- Line 72–73 allocates memory for the solution vector x and set its initial value to be all zero.
- Line 75 solves $Ax = b$ using the AMG method. Type the AMG method and other parameters have been given in "amgparam" at Line 24; see §3.7.
- Line 78–80 frees up memory allocated for A, b, and x.

To run this example, we can simply type (the default parameters have been in "tutorial/ini/amg.dat").

```
$ ./poisson-amg-c.ex
```

A sample output is given as follows (note that the actual output depends on the solver parameters and might be different than what you see here):

```
============================================
||   FASP: AMG example -- C version   ||
============================================

fasp_dcsrvec2_read: reading file ../data/csrmat_FE.dat...
fasp_dcsrvec2_read: reading file ../data/rhs_FE.dat...
A: m = 3969, n = 3969, nnz = 27281
b: n = 3969
```

```
       Parameters in AMG_param
-----------------------------------------------------
AMG print level:                    3
AMG max num of iter:                100
AMG type:                           1
AMG tolerance:                      1.00e-08
AMG max levels:                     20
AMG cycle type:                     1
AMG scaling of coarse correction:   0
AMG smoother type:                  2
AMG smoother order:                 1
AMG num of presmoothing:            2
AMG num of postsmoothing:           2
AMG coarsening type:                1
AMG interpolation type:             1
AMG dof on coarsest grid:           500
AMG strong threshold:               0.6000
AMG truncation threshold:           0.4000
AMG max row sum:                    0.9000
AMG aggressive levels:              0
AMG aggressive path:                1
-----------------------------------------------------


Calling classical AMG ...
-----------------------------------------------------
  Level     Num of rows      Num of nonzeros
-----------------------------------------------------
    0            3969             27281
    1            1985             28523
    2             961             20519
    3             481             13153
-----------------------------------------------------
AMG grid complexity     = 1.863
AMG operator complexity = 3.280
Classical AMG setup costs 0.0123 seconds.
-------------------------------------------------------------------
It Num |    ||r||/||b||    |      ||r||       |  Conv. Factor
-------------------------------------------------------------------
     0 |  1.000000e+00   |  7.514358e+00  |     -.--
     1 |  5.160015e-04   |  3.877420e-03  |     0.0005
     2 |  1.895524e-06   |  1.424365e-05  |     0.0037
     3 |  7.554832e-09   |  5.676972e-08  |     0.0040
Number of iterations = 3 with relative residual 7.554832e-09.
AMG solve costs 0.0189 seconds.
AMG totally costs 0.0316 seconds.
```

2.2 The second example

In the second example, we modify the previous example slightly and solve the Poisson equation using iterative methods.

```
1  /*! \file poisson-its.c
2   *  \brief The second test example for FASP: using ITS to solve
3   *         the discrete Poisson equation from P1 finite element.
4   *
5   *  \note  ITS example for FASP: C version
6   *
7   *  Solving the Poisson equation (P1 FEM) with iterative methods
8   */
```

```
9
10 #include "fasp.h"
11 #include "fasp_functs.h"
12
13 /**
14  * \fn int main (int argc, const char * argv[])
15  *
16  * \brief This is the main function for the second example.
17  *
18  * \author Feiteng Huang
19  * \date   04/13/2012
20  *
21  * Modified by Chensong Zhang on 09/22/2012
22  */
23 int main (int argc, const char * argv[])
24 {
25     input_param          inparam;  // parameters from input files
26     itsolver_param       itparam;  // parameters for itsolver
27
28     printf("\n=========================================");
29     printf("\n||   FASP: ITS example -- C version   ||");
30     printf("\n=========================================\n\n");
31
32     // Step 0. Set parameters
33     // Read input and AMG parameters from a disk file
34     // In this example, we read everything from a disk file:
35     //          "ini/its.dat"
36     // See the reference manual for details of the parameters.
37     fasp_param_init("ini/its.dat",&inparam,&itparam,NULL,NULL,NULL);
38
39     // Set local parameters
40     const int print_level = inparam.print_level;
41
42     // Step 1. Get stiffness matrix and right-hand side
43     // Read A and b -- P1 FE discretization for Poisson. The location
44     // of the data files is given in "ini/its.dat".
45     dCSRmat A;
46     dvector b, x;
47     char filename1[512], *datafile1;
48     char filename2[512], *datafile2;
49
50     // Read the stiffness matrix from matFE.dat
51     strncpy(filename1,inparam.workdir,128);
52     datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);
53
54     // Read the RHS from rhsFE.dat
55     strncpy(filename2,inparam.workdir,128);
56     datafile2="rhs_FE.dat"; strcat(filename2,datafile2);
57
58     fasp_dcsrvec2_read(filename1,filename2,&A,&b);
59
60     // Step 2. Print problem size and ITS parameters
61     if (print_level>PRINT_NONE) {
62         printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
63         printf("b: n = %d\n", b.row);
64         fasp_param_solver_print(&itparam);
65     }
66
67     // Step 3. Solve the system with ITS as an iterative solver
68     // Set the initial guess to be zero and then solve it using standard
69     // iterative methods, without applying any preconditioners
70     fasp_dvec_alloc(A.row, &x);
71     fasp_dvec_set(A.row,&x,0.0);
72
```

```
73      fasp_solver_dcsr_itsolver(&A, &b, &x, NULL, &itparam);
74
75      // Step 4. Clean up memory
76      fasp_dcsr_free(&A);
77      fasp_dvec_free(&b);
78      fasp_dvec_free(&x);
79
80      return SUCCESS;
81  }
82
83  /*---------------------------------*/
84  /*--        End of File          --*/
85  /*---------------------------------*/
```

This example is very similar to the first example and we briefly explain the differences:

- Line 37 is slightly different than Line 39 in the previous example—Instead of initializing "amgparam", we initialize "itparam". This is because we are calling general interface of Krylov subspace methods [14] and the parameters are recorded in "itparam".
- Line 70–71 allocates memory for the solution vector x and set its initial value to be all zero.
- Line 73 solves $Ax = b$ using the general interface for Krylov subspace methods. Type the iterative method and other parameters have been specified in "itparam"; see §3.5 for details.

To run this example, we can simply type (the default parameters have been in "tutorial/ini/its.dat").

```
$ ./poisson-its-c.ex
```

2.3 The third example

This example is slightly longer and is a modification of the previous one. In this example, we wish to demonstrate how to setup a simple preconditioner for the preconditioned conjugate gradient (PCG) method.

```
1   /*! \file poisson-pcg.c
2    *  \brief The third test example for FASP: using PCG to solve
3    *         the discrete Poisson equation from P1 finite element.
4    *         C version.
5    *
6    *  \note  PCG example for FASP: C version
7    *
8    *  Solving the Poisson equation (P1 FEM) with PCG methods
9    */
10
11  #include "fasp.h"
12  #include "fasp_functs.h"
13
14  /**
15   * \fn int main (int argc, const char * argv[])
16   *
17   * \brief This is the main function for the third example.
18   *
19   * \author Feiteng Huang
20   * \date   05/17/2012
21   *
```

```
 * Modified by Chensong Zhang on 09/22/2012
 */
int main (int argc, const char * argv[])
{
    input_param          inparam;  // parameters from input files
    itsolver_param       itparam;  // parameters for itsolver
    AMG_param            amgparam; // parameters for AMG
    ILU_param            iluparam; // parameters for ILU

    printf("\n=========================================");
    printf("\n||   FASP: PCG example -- C version   ||");
    printf("\n=========================================\n\n");

    // Step 0. Set parameters
    // Read input and precondition parameters from a disk file
    // In this example, we read everything from a disk file:
    //          "ini/pcg.dat"
    // See the reference manual for details of the parameters.
    fasp_param_init("ini/pcg.dat",&inparam,&itparam,&amgparam,&iluparam,NULL);

    // Set local parameters
    const SHORT print_level = itparam.print_level;
    const SHORT pc_type     = itparam.precond_type;
    const SHORT stop_type   = itparam.stop_type;
    const INT   maxit       = itparam.maxit;
    const REAL  tol         = itparam.tol;

    // Step 1. Get stiffness matrix and right-hand side
    // Read A and b -- P1 FE discretization for Poisson. The location
    // of the data files is given in "ini/pcg.dat".
    dCSRmat A;
    dvector b, x;
    char filename1[512], *datafile1;
    char filename2[512], *datafile2;

    // Read the stiffness matrix from matFE.dat
    strncpy(filename1,inparam.workdir,128);
    datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);

    // Read the RHS from rhsFE.dat
    strncpy(filename2,inparam.workdir,128);
    datafile2="rhs_FE.dat"; strcat(filename2,datafile2);

    fasp_dcsrvec2_read(filename1,filename2,&A,&b);

    // Step 2. Print problem size and PCG parameters
    if (print_level>PRINT_NONE) {
        printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
        printf("b: n = %d\n", b.row);
        fasp_param_solver_print(&itparam);
    }

    // Setp 3. Setup preconditioner
    // Preconditioiner type is determined by pc_type
    precond *pc = fasp_precond_setup(pc_type, &amgparam, &iluparam, &A);

    // Step 4. Solve the system with PCG as an iterative solver
    // Set the initial guess to be zero and then solve it using pcg solver
    // Note that we call PCG interface directly. There is another way which
    // calls the abstract iterative method interface; see possion-its.c for
    // more details.
    fasp_dvec_alloc(A.row, &x);
    fasp_dvec_set(A.row, &x, 0.0);

```

```
86      fasp_solver_dcsr_pcg(&A, &b, &x, pc, tol, maxit, stop_type, print_level);
87
88      // Step 5. Clean up memory
89      if (pc_type!=PREC_NULL) fasp_mem_free(pc->data);
90      fasp_dcsr_free(&A);
91      fasp_dvec_free(&b);
92      fasp_dvec_free(&x);
93
94      return SUCCESS;
95  }
96
97  /*----------------------------------*/
98  /*--        End of File           --*/
99  /*----------------------------------*/
```

This example is very similar to the first example and we now briefly explain it:

- Line 40 reads parameters from "tutorial/ini/pcg.dat". In this example, we need parameters for iterative methods, AMG preconditioner, and ILU preconditioner. The type of the preconditioner is also set in "pcg.dat" and recorded in "pc_type"; see Line 30 in "pcg.dat".
- Line 76 sets up the desired preconditioner and prepare it for the preconditioned iterative methods.
- Line 86 calls PCG to solve $Ax = b$. One can also call the general iterative method interface as in the previous example.
- Line 89 cleans up auxiliary data associated with the preconditioner in use if necessary.

To run this example, we can simply type (the default parameters have been in "tutorial/ini/pcg.dat").

```
$ ./poisson-pcg-c.ex
```

2.4 Set parameters

In the previous examples, we have seen how to read input parameters from disk files. Now we take a brief look inside those ini files. We take "tutorial/ini/amg.dat" as an example:

```
1   %-------------------------------------------------%
2   % input parameters                                %
3   % lines starting with % are comments              %
4   % must have spaces around the equal sign "="      %
5   %-------------------------------------------------%
6
7   workdir = ../data/    % work directory, no more than 128 characters
8   print_level = 3       % How much information to print out
9
10  %-------------------------------------------------%
11  % parameters for multilevel iteration             %
12  %-------------------------------------------------%
13
14  AMG_type                  = C      % C classic AMG
15                                     % SA smoothed aggregation
16                                     % UA unsmoothed aggregation
17  AMG_cycle_type            = V      % V V-cycle | W W-cycle
```

```
18                                  % A AMLI-cycle | NA Nonlinear AMLI-cycleA
19  AMG_tol                 = 1e-8  % tolerance for AMG
20  AMG_maxit               = 100   % number of AMG iterations
21  AMG_levels              = 20    % max number of levels
22  AMG_coarse_dof          = 500   % max number of coarse degrees of freedom
23  AMG_coarse_scaling      = OFF   % switch of scaling of the coarse grid correction
24  AMG_amli_degree         = 2     % degree of the polynomial used by AMLI cycle
25  AMG_nl_amli_krylov_type = 6     % Krylov method in NLAMLI cycle: 6 FGMRES | 7 GCG
26
27  %-------------------------------------------------%
28  % parameters for AMG smoothing                    %
29  %-------------------------------------------------%
30
31  AMG_smoother            = GS    % GS | JACOBI | SGS
32                                  % SOR | SSOR | GSOR | SGSOR | POLY
33  AMG_ILU_levels          = 0     % number of levels using ILU smoother
34  AMG_schwarz_levels      = 0     % number of levels using Schwarz smoother
35  AMG_relaxation          = 1.1   % relaxation parameter for SOR smoother
36  AMG_polynomial_degree   = 3     % degree of the polynomial smoother
37  AMG_presmooth_iter      = 2     % number of presmoothing sweeps
38  AMG_postsmooth_iter     = 2     % number of postsmoothing sweeps
39
40  %-------------------------------------------------%
41  % parameters for classical AMG SETUP              %
42  %-------------------------------------------------%
43
44  AMG_coarsening_type     = 1     % 1 Modified RS
45                                  % 3 Compatible Relaxation
46                                  % 4 Aggressive
47  AMG_interpolation_type  = 1     % 1 Direct | 2 Standard | 3 Energy-min
48  AMG_strong_threshold    = 0.6   % Strong threshold
49  AMG_truncation_threshold = 0.4  % Truncation threshold
50  AMG_max_row_sum         = 0.9   % Max row sum
51
52  %-------------------------------------------------%
53  % parameters for aggregation-type AMG SETUP       %
54  %-------------------------------------------------%
55
56  AMG_strong_coupled      = 0.08  % Strong coupled threshold
57  AMG_max_aggregation     = 20    % Max size of aggregations
58  AMG_tentative_smooth    = 0.67  % Smoothing factor for tentative prolongation
59  AMG_smooth_filter       = OFF   % Switch for filtered matrix for smoothing
```

We now briefly discuss the parameters above: This example is very similar to the first example and we now briefly explain it:

- Line 7 sets the working directory, which should contain data files for the matrices (and right-hand side vectors when necessary).
- Line 8 sets the level of output for FASP routines. It should range from 0 to 10 with 0 means no output and 10 means output everything possible.
- Line 14–25 sets the basic parameters for multilevel iterations. For example, type of AMG, type of multilevel cycles, number of maximal levels, etc.
- Line 31–38 sets the type of smoothers, number of smoothing sweeps, etc.
- Line 44–50 sets the parameters for the setup phase of the classical AMG method (§3.7).
- Line 56–59 gives the parameters for the setup phase of the aggregation-base AMG methods (§3.7).

You can do a very simple experiment and change the AMG type from the classical AMG to smoothed aggregation AMG by revise Line 14 to

```
AMG_type                = SA
```

Then you run "poisson-amg-c.ex" one more time and will get

```
=============================================
||   FASP: AMG example -- C version   ||
=============================================

fasp_dcsrvec2_read: reading file ../data/csrmat_FE.dat...
fasp_dcsrvec2_read: reading file ../data/rhs_FE.dat...
A: m = 3969, n = 3969, nnz = 27281
b: n = 3969

       Parameters in AMG_param
-----------------------------------------------
AMG print level:                    3
AMG max num of iter:                100
AMG type:                           2
AMG tolerance:                      1.00e-08
AMG max levels:                     20
AMG cycle type:                     1
AMG scaling of coarse correction:   0
AMG smoother type:                  2
AMG smoother order:                 1
AMG num of presmoothing:            2
AMG num of postsmoothing:           2
Aggregation AMG strong coupling:    0.0800
Aggregation AMG max aggregation:    20
Aggregation AMG tentative smooth:   0.6700
Aggregation AMG smooth filter:      0
-----------------------------------------------


Calling SA AMG ...
-----------------------------------------------
  Level     Num of rows      Num of nonzeros
-----------------------------------------------
    0            3969             27281
    1             541              6531
    2              41               421
-----------------------------------------------
AMG grid complexity     = 1.147
AMG operator complexity = 1.255
Smoothed aggregation setup costs 0.0072 seconds.
-------------------------------------------------------------------
It Num |    ||r||/||b||    |      ||r||       |  Conv. Factor
-------------------------------------------------------------------
     0 |  1.000000e+00   |  7.514358e+00  |      -.--
     1 |  4.345463e-02   |  3.265336e-01  |      0.0435
     2 |  8.041967e-03   |  6.043022e-02  |      0.1851
     3 |  3.808810e-03   |  2.862076e-02  |      0.4736
     4 |  1.838990e-03   |  1.381883e-02  |      0.4828
     5 |  8.675952e-04   |  6.519421e-03  |      0.4718
     6 |  4.089274e-04   |  3.072827e-03  |      0.4713
     7 |  1.939823e-04   |  1.457653e-03  |      0.4744
     8 |  9.276723e-05   |  6.970862e-04  |      0.4782
     9 |  4.471799e-05   |  3.360270e-04  |      0.4820
    10 |  2.171249e-05   |  1.631554e-04  |      0.4855
    11 |  1.060934e-05   |  7.972239e-05  |      0.4886
```

```
    12 |   5.212246e-06   |   3.916668e-05   |      0.4913
    13 |   2.572464e-06   |   1.933042e-05   |      0.4935
    14 |   1.274466e-06   |   9.576797e-06   |      0.4954
    15 |   6.333891e-07   |   4.759512e-06   |      0.4970
    16 |   3.155926e-07   |   2.371476e-06   |      0.4983
    17 |   1.575755e-07   |   1.184079e-06   |      0.4993
    18 |   7.881043e-08   |   5.922098e-07   |      0.5001
    19 |   3.947044e-08   |   2.965950e-07   |      0.5008
    20 |   1.978978e-08   |   1.487075e-07   |      0.5014
    21 |   9.931176e-09   |   7.462641e-08   |      0.5018
Number of iterations = 21 with relative residual 9.931176e-09.
AMG solve costs 0.0252 seconds.
AMG totally costs 0.0327 seconds.
```

You can compare this with the sample results in §2.1.

The input parameters allowed in FASP are not limited to the ones listed in this example. A list of possible iterative methods and preconditioners can be found in "base/include/messages.h"; see §4.3. For more parameters and their ranges, we refer to the Reference Manual.

Chapter 3

Basic Usage

In this chapter, we discuss the basic data structures and important building blocks which will be useful later for constructing auxiliary space preconditioners for systems of PDEs in Chapter 4. In particular, we will discuss vectors, sparse matrices, iterative methods, and multigrid methods.

3.1 Vectors and sparse matrices

The most important data structures for iterative methods are probably vectors and sparse matrices. In this section, we first discuss the data structures for vectors and matrices in FASP; and then we discuss BLAS for sparse matrices.

Vectors

The data structure for vectors is very simple. It only contains the length of the vector and an array which contains the entries of this vector.

```
/**
 * \struct dvector
 * \brief Vector with n entries of REAL type.
 */
typedef struct dvector{

    //! number of rows
    INT row;
    //! actual vector entries
    REAL *val;

} dvector; /**< Vector of REAL type */
```

Sparse matrices

On the other hand, sparse matrices for PDE applications are very complicated. It depends on the particular applications, discretization methods, as well as solution algorithms. In FASP, there are several types of sparse matrices, COO, CSR, CSRL, BSR, and CSR Block, etc. The presentation closely follows ideas from Pissanetzky [11].

In this section, we use the following sparse matrix as an example to explain different formats for sparse matrices:

Example 3.1.1 *Consider the following* 4×5 *matrix with 12 non-zero entries*

$$\begin{pmatrix} 1 & 1.5 & 0 & 0 & 12 \\ 0 & 1 & 6 & 7 & 1 \\ 3 & 0 & 6 & 0 & 0 \\ 1 & 0 & 2 & 0 & 5 \end{pmatrix}$$

(i) COO format

The coordinate (COO) format or IJ format is the simplest sparse matrix format.

```
/**
 * \struct dCOOmat
 * \brief Sparse matrix of REAL type in COO (or IJ) format.
 *
 * Coordinate Format (I,J,A)
 *
 * \note The starting index of A is 0.
 */
typedef struct dCOOmat{

    //! row number of matrix A, m
    INT row;
    //! column of matrix A, n
    INT col;
    //! number of nonzero entries
    INT nnz;
    //! integer array of row indices, the size is nnz
    INT *I;
    //! integer array of column indices, the size is nnz
    INT *J;
    //! nonzero entries of A
    REAL *val;

} dCOOmat; /**< Sparse matrix of REAL type in COO format */
```

So it clear that the sparse matrix in Example 3.1.1 in COO format is stored as:

```
row = 4
col = 5
nnz = 12

 I J  val
------------
 0 0   1.0
 0 1   1.5
 0 4  12.0
 1 1   1.0
 1 2   6.0
 1 3   7.0
 1 4   1.0
 .......
```

Although the COO format is easy to understand or use, it wastes storage space and has little advantages in sparse BLAS operations.

NOTE: In FASP, the indices always start from 0, instead of from 1. This is often the source of problems related to vectors and matrices.

(ii) CSR format

The most commonly used data structure for sparse matrices nowadays is probably the so-called *compressed sparse row* (CSR) format, according to Saad [14]. The compressed row storage format of a matrix $A \in \mathbb{R}^{n\times m}$ (n rows and m columns) consists of three arrays, as follows:

1. An integer array of row pointers of size n+1;
2. An integer array of column indexes of size nnz;
3. An array of actual matrix entries.

In FASP, we define:

```
/**
 * \struct dCSRmat
 * \brief Sparse matrix of REAL type in CSR format.
 *
 * CSR Format (IA,JA,A) in REAL
 *
 * \note The starting index of A is 0.
 */
typedef struct dCSRmat{

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

} dCSRmat; /**< Sparse matrix of REAL type in CSR format */
```

The matrix (only nonzero elements) is stored in the array val row after row, in a way that i-th row begins at $val(IA(i))$ and ends at $val(IA(i+1)-1)$. In the same way, $JA(IA(i)$ to $JA(IA(i+1)-1)$ will contain the column indexes of the non-zeros in row i. Thus IA is of size $n+1$ (number of rows in val plus one), JA and val are of size equal to the number of non-zeroes. The total number of non-zeroes is equal to $IA(n+1)-1$.

NOTE: When the sparse matrix A is a boolean (i.e. all entries are either 0 or 1), the actual non-zeroes are not stored because it is understood that, if it is nonzero, it could only be 1 and there is no need to store it.

The matrix in Example 3.1.1 in CSR format is represented in the following way:

- IA is of size 5 and
$$IA = \| \ 0 \ \| \ 3 \ \| \ 7 \ \| \ 9 \ \| \ 12 \ \|$$
- JA is of size $IA(5) - 1 = 12$
$$JA = \| \ 0 \ | \ 1 \ | \ 4 \ \| \ 1 \ | \ 3 \ | \ 2 \ | \ 4 \ \| \ 0 \ | \ 2 \ \| \ 2 \ | \ 4 \ | \ 0 \ \|$$
- val is of the same size as JA and
$$val = \| \ 1. \ | \ 1.5 \ | \ 12. \ \| \ 1. \ | \ 7. \ | \ 6. \ | \ 1. \ \| \ 3. \ | \ 6. \ \| \ 2. \ | \ 5. \ | \ 1. \ \|$$

Here we use double vertical bars to separate rows and single vertical bars to separate values.

NOTE: The indices in JA and entries of val does NOT have to be ordered as seen in this example. Sometimes they are sorted in ascending order in each row. More often, the diagonal entries are stored in the first position in each row and the rest are sorted in ascending order.

Below is a "non-numeric" example.

Example 3.1.2 *Consider the following sparse matrix:*

$$\begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & a_{32} & 0 & a_{34} \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix}$$

For this matrix, we have that the number of non-zeros $nnz = 10$*. Furthermore, the three arrays of in the CSR format are:*

$$IA = \| \ 0 \ \| \ 2 \ \| \ 5 \ \| \ 7 \ \|,$$
$$JA = \| \ 0 \ | \ 2 \ \| \ 1 \ | \ 2 \ | \ 3 \ \| \ 1 \ | \ 3 \ \| \ 0 \ | \ 1 \ | \ 2 \ \|,$$

and

$$val = \| \ a_{11} \ | \ a_{13} \ \| \ a_{22} \ | \ a_{23} \ | \ a_{24} \ \| \ a_{32} \ | \ a_{34} \ \| \ a_{41} \ | \ a_{42} \ | \ a_{43} \ \|.$$

NOTE: The CSR format presents challenges to sparse matrix-vector product mainly because of the high cache missing rate due to indirect memory access and irregular access pattern. In order to reduce the cache missing rate, we introduce an improved data format, CSRL.

(iii) CSRL format

CSRL matrix format [9] groups rows with same number of nonzeros together and improves cache hitting rate.

```
1 /*!
2  * \struct dCSRLmat
3  * \brief Sparse matrix of REAL type in CSRL format.
4  */
5 typedef struct dCSRLmat{
```

```
6
7       //! number of rows
8       INT row;
9       //! number of cols
10      INT col;
11      //! number of nonzero entries
12      INT nnz;
13      //! number of different values in i-th row, i=0:nrows-1
14      INT dif;
15      //! nz_diff[i]: the i-th different value in 'nzrow'
16      INT *nz_diff;
17      //! row index of the matrix (length-grouped): rows with same nnz are together
18      INT *index;
19      //! j in {start[i],...,start[i+1]-1} means nz_diff[i] nnz in index[j]-row
20      INT *start;
21      //! column indices of all the nonzeros
22      INT *ja;
23      //! values of all the nonzero entries
24      REAL *val;
25
26  } dCSRLmat; /**< Sparse matrix of REAL type in CSRL format */
```

3.2 Block sparse matrices

For PDE applications, we often need to solve systems of partial differential equations. Many iterative methods and preconditioners could take advantages of the structure of PDE systems and improve efficiency. So we often need to use semi-structured (block) sparse data structures to store the coefficient matrix arising from PDE systems.

Depending on different applications and different solving algorithms, we can use two types of block matrices: dBSRmat (or BSR Block Compressed Sparse Row) and block_dCSRmat (CSR Block or Block of CSR matrices).

For more details as well as other specialized block matrices, readers are referred to the header file "base/include/fasp_block.h".

As an example, we consider the following matrix, which have been used in §3.1 for the CSR format. We add structure to this matrix and divide it as a 2×2 block matrix:

Example 3.2.1

$$\left(\begin{array}{cc|cc} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ \hline 0 & a_{32} & 0 & a_{34} \\ a_{41} & a_{42} & a_{43} & 0 \end{array}\right)$$

(i) BSR format

This format is a standard data structure for storing block sparse matrices which has been used by the Intel MKL library.

```
1   /**
2    * \struct dBSRmat
```

```
 * \brief 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
 *
 * \note Some of the following entries are capitalized to stress that they are
 *       for blocks!
 *
 */
typedef struct dBSRmat{


    //! number of rows of sub-blocks in matrix A, M
    INT ROW;
    //! number of cols of sub-blocks in matrix A, N
    INT COL;
    //! number of nonzero sub-blocks in matrix A, NNZ
    INT NNZ;
    //! dimension of each sub-block
    INT nb; // for the moment, allow nb*nb full block
    //! storage manner for each sub-block
    INT storage_manner; // 0: row-major order, 1: column-major order

    //! 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).
    REAL *val;

    //! integer array of row pointers, the size is ROW+1
    INT *IA;

    //! 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.
    INT *JA;

} dBSRmat; /**< Matrix of REAL type in BSR format */
```

For the matrix in Example 3.2.1, we have that the number of block rows $ROW = 2$, the number of block columns $COL = 2$, and the number of block nonzeros $NNZ = 4$. The block size is $nb = 2$. We can choose different storage manners for storing the small blocks. Suppose that we set it to be 0, i.e. row-major format. Then the three arrays of in the BSR format are:

$$IA = \| \ 0 \ \| \ 8 \ \| \ 16 \ \|,$$

$$JA = \| \ 0 \ | \ 1 \ \| \ 0 \ | \ 1 \ \|,$$

and

$$\begin{aligned} val = \ & \| \ a_{11} \ | \ 0 \ | \ 0 \ | \ a_{22} \ \| \ a_{13} \ | \ 0 \ | \ a_{23} \ | \ a_{24} \ \| \\ & \| \ 0 \ | \ a_{32} \ | \ a_{41} \ | \ a_{42} \ \| \ 0 \ | \ a_{34} \ | \ a_{43} \ | \ 0 \ \|. \end{aligned}$$

We immediately notice that this format might be not be the best choice for this particular matrix due to all the blocks are nonzero blocks, i.e., contain nonzero entries. However, for PDE applications, this does not usually happen.

(ii) CSR Block format

This format is simple and is derived from the dCSRmat data structure. The following definition explains itself.

```
/**
 * \struct block_dCSRmat
 * \brief Block of REAL CSR matrix structure.
 *
 * CSR Block Format in REAL
 *
 * \note The starting index of A is 0.
 */
typedef struct block_dCSRmat{

    //! row number of blocks in A, m
    INT brow;
    //! column number of blocks A, n
    INT bcol;
    //! blocks of dCSRmat, point to blocks[brow][bcol]
    dCSRmat **blocks;

} block_dCSRmat; /**< Matrix of REAL type in Block BSR format */
```

3.3 I/O subroutines for sparse matrices

To be added.

3.4 Sparse BLAS

The matrix-vector multiplication: $y = Ax$ can be performed in the following simple way:

```
/**
 * \fn void fasp_blas_dcsr_mxv (dCSRmat *A, REAL *x, REAL *y)
 *
 * \brief Matrix-vector multiplication y = A*x
 *
 * \param A   Pointer to dCSRmat matrix A
 * \param x   Pointer to array x
 * \param y   Pointer to array y
 *
 * \author Chensong Zhang
 * \date   07/01/2009
 */
void fasp_blas_dcsr_mxv (dCSRmat *A,
                         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, beg, end;
    register REAL tmp;

    for ( i=0; i<m; ++i ) {
        tmp = 0.0;
```

```
26          beg = ia[i]; end = ia[i+1];
27          for ( k=beg; k<end; ++k ) tmp += aj[k]*x[ja[k]];
28          y[i] = tmp;
29      }
30  }
```

This is only a simple example for sparse matrix-vector multiplication (SpMV) kernel. Since we need many types of sparse matrices, there are various of versions of SpMV for different data structures. See the Reference Manual for more details.

3.5 Iterative methods

In FASP, there are a couple of standard preconditioned iterative methods [14] implemented, including preconditioned CG, BiCGstab, GMRES, Variable Restarting GMRES, Flexible GMRES, etc. In this section, we use the CSR matrix format as example to introduce how to call these iterative methods. To learn more details, we refer to the Reference Manual.

We first notice the abstract interface for the iterative methods is:

```
/**
 * \fn INT fasp_solver_dcsr_itsolver (dCSRmat *A, dvector *b, dvector *x,
 *                                    precond *pc, itsolver_param *itparam)
 *
 * \brief Solve Ax=b by preconditioned Krylov methods for CSR matrices
 *
 * \param A         Pointer to the coeff matrix in dCSRmat format
 * \param b         Pointer to the right hand side in dvector format
 * \param x         Pointer to the approx solution in dvector format
 * \param pc        Pointer to the preconditioning action
 * \param itparam   Pointer to parameters for iterative solvers
 *
 * \return          Number of iterations if succeed
 *
 * \author Chensong Zhang
 * \date   09/25/2009
 *
 * \note This is an abstract interface for iterative methods.
 */
INT fasp_solver_dcsr_itsolver (dCSRmat *A,
                               dvector *b,
                               dvector *x,
                               precond *pc,
                               itsolver_param *itparam)
```

The names of the input arguments explain themselves mostly and they are explained in the Reference Manual in detail.

We briefly discuss how to call this function; and, once you understand PCG, you can easily call other iterative methods. The following code segment is taken from "base/src/itsolver_csr.c":

```
1  ... ...
2
3      // ILU setup for whole matrix
4      ILU_data LU;
5      if ( (status = fasp_ilu_dcsr_setup(A,&LU,iluparam))<0 ) goto FINISHED;
6
7      // check iludata
8      if ( (status = fasp_mem_iludata_check(&LU))<0 ) goto FINISHED;
```

```
9
10      // set preconditioner
11      precond pc;
12      pc.data = &LU;
13      pc.fct  = fasp_precond_ilu;
14
15      // call iterative solver
16      status = fasp_solver_dcsr_itsolver(A,b,x,&pc,itparam);
17
18  ... ...
```

Now we explain this code segment a little bit:

- Line 3–4 performs the setup phase for ILU method. The particular type of ILU method is determined by "iluparam"; see §2.4. Line 7 performs a simple memory check for ILU.
- Line 10–12 defines the preconditioner data structure "pc", which contains two parts: one is the actual preconditioning action "pc.fct", the other is the auxiliary data which is needed to perform the preconditioning action "pc.data".
- Line 15 calls iterative methods. "A" is the matrix in dCSRmat format; "b" and "x" are the right-hand side and the solution vectors, respectively. Similar to ILU setup, the type of iterative methods is determined by "itparam".

Apparently, we now left with no choice but introducing "itparam".

```
/**
 * \struct itsolver_param
 * \brief Parameters passed to iterative solvers.
 *
 */
typedef struct {

    SHORT itsolver_type; /**< solver type: see message.h */
    SHORT precond_type;  /**< preconditioner type: see message.h */
    SHORT stop_type;     /**< stopping criteria type */
    INT   maxit;         /**< max number of iterations */
    REAL  tol;           /**< convergence tolerance */
    INT   restart;       /**< number of steps for restarting: for GMRES etc */
    SHORT print_level;   /**< print level: 0--10 */

} itsolver_param; /**< Parameters for iterative solvers */
```

Possible "itsolver_type" includes:

```
/**
 * \brief Definition of solver types for iterative methods
 */
#define SOLVER_CG                1    /**< Conjugate Gradient */
#define SOLVER_BiCGstab          2    /**< Biconjugate Gradient Stabilized */
#define SOLVER_MinRes            3    /**< Minimal Residual */
#define SOLVER_GMRES             4    /**< Generalized Minimal Residual */
#define SOLVER_VGMRES            5    /**< Variable Restarting GMRES */
#define SOLVER_VFGMRES           6    /**< Variable Restarting Flexible GMRES */
#define SOLVER_GCG               7    /**< Generalized Conjugate Gradient */
#define SOLVER_AMG               21   /**< AMG as an iterative solver */
#define SOLVER_FMG               22   /**< Full AMG as an solver */
```

3.6 Geometric multigrid

To be added.

3.7 Algebraic multigrid

The classical algebraic multigrid method [13] is an important component in many of our auxiliary space preconditioners. Because of its user-friendly and scalability, AMG becomes increasingly popular in scientific and engineering computing, especially when GMG is difficult or not possible to be applied. Various of new AMG techniques [15, 16, 3, 6, 4, 8, 5, 18, 2, 10, 7] have emerged in recent years.

The following code segment is part of "base/src/amg.c" and it is a good example which shows how to call different AMG methods (classical AMG, smoothed aggregation, un-smoothed aggregation) and different multilevel iterative methods (V-cycle, W-cycle, AMLI-cycle, Nonlinear AMLI-cycle, etc).

```
... ...

    // param is a pointer to AMG_param
    const SHORT   max_levels  = param->max_levels;
    const SHORT   amg_type    = param->AMG_type;
    const SHORT   cycle_type  = param->cycle_type;

    // initialize mgl[0] with A, b, x
    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);       fasp_dvec_cp(b,&mgl[0].b);
    mgl[0].x = fasp_dvec_create(n);       fasp_dvec_cp(x,&mgl[0].x);

    // AMG setup phase
    switch (amg_type) {

    case SA_AMG: // Smoothed Aggregation AMG setup phase
        if ( (status=fasp_amg_setup_sa(mgl, param)) < 0 ) goto FINISHED;
        break;

    case UA_AMG: // Unsmoothed Aggregation AMG setup phase
        if ( (status=fasp_amg_setup_ua(mgl, param)) < 0 ) goto FINISHED;
        break;

    default: // Classical AMG setup phase
        if ( (status=fasp_amg_setup_rs(mgl, param)) < 0 ) goto FINISHED;
        break;

    }

    // AMG solve phase
    switch (cycle_type) {

    case AMLI_CYCLE: // call AMLI-cycle
        if ( (status=fasp_amg_solve_amli(mgl, param)) < 0 ) goto FINISHED;
        break;

    case NL_AMLI_CYCLE: // call Nonlinear AMLI-cycle
        if ( (status=fasp_amg_solve_nl_amli(mgl, param)) < 0 ) goto FINISHED;
        break;

```

```
42      default: // call classical V,W-cycles
43          if ( (status=fasp_amg_solve(mgl, param)) < 0 ) goto FINISHED;
44          break;
45
46      }
47
48  ... ...
```

The code above is very simple and we only wish to point out that:

- Line 4–6 reads some of the parameters from "AMG_param", which can be defined from a input file; see §2.4.
- Line 9–12 initializes the "AMG_data" with a copy of the coefficient matrix, the right-hand side, and the initial solution (it will store the final solution eventually).
- Line 17–27 calls three different AMG methods, determined by "amg_type".
- Line 34–44 calls three different multilevel iterative methods, determined by "cycle_type".

Parameters for AMG

There are a couple of controlling parameters for algebraic multigrid methods in FASP. Basically, there are four types of parameters for AMG—They control multilevel iterations, smoothing, classical AMG setup, and aggregation AMG setup. The following is a sample from "test/ini/input.dat" and a brief explanation of each parameter is given.

```
1  %-------------------------------------------------%
2  % parameters for multilevel iteration            %
3  %-------------------------------------------------%
4
5  AMG_type                  = C      % C classic AMG
6                                     % SA smoothed aggregation
7                                     % UA unsmoothed aggregation
8  AMG_cycle_type            = V      % V V-cycle | W W-cycle
9                                     % A AMLI-cycle | NA Nonlinear AMLI-cycleA
10 AMG_tol                   = 1e-8   % tolerance for AMG
11 AMG_maxit                 = 1      % number of AMG iterations
12 AMG_levels                = 20     % max number of levels
13 AMG_coarse_dof            = 100    % max number of coarse degrees of freedom
14 AMG_coarse_scaling        = OFF    % switch of scaling of the coarse grid correction
15 AMG_amli_degree           = 2      % degree of the polynomial used by AMLI cycle
16 AMG_nl_amli_krylov_type   = 6      % Krylov method in NLAMLI cycle: 6 FGMRES | 7 GCG
17
18 %-------------------------------------------------%
19 % parameters for AMG smoothing                   %
20 %-------------------------------------------------%
21
22 AMG_smoother              = GS     % GS | JACOBI | SGS
23                                    % SOR | SSOR | GSOR | SGSOR | POLY
24 AMG_ILU_levels            = 0      % number of levels using ILU smoother
25 AMG_schwarz_levels        = 0      % number of levels using Schwarz smoother
26 AMG_relaxation            = 1.1    % relaxation parameter for SOR smoother
27 AMG_polynomial_degree     = 3      % degree of the polynomial smoother
28 AMG_presmooth_iter        = 1      % number of presmoothing sweeps
29 AMG_postsmooth_iter       = 1      % number of postsmoothing sweeps
30
31 %-------------------------------------------------%
```

```
32 % parameters for classical AMG SETUP                %
33 %-------------------------------------------------------%
34
35 AMG_coarsening_type      = 1      % 1 Modified RS
36                                   % 3 Compatible Relaxation
37                                   % 4 Aggressive
38 AMG_interpolation_type   = 1      % 1 Direct | 2 Standard | 3 Energy-min
39 AMG_strong_threshold     = 0.25   % Strong threshold
40 AMG_truncation_threshold = 0.4    % Truncation threshold
41 AMG_max_row_sum          = 0.9    % Max row sum
42
43 %-------------------------------------------------------%
44 % parameters for aggregation-type AMG SETUP    %
45 %-------------------------------------------------------%
46
47 AMG_strong_coupled       = 0.08   % Strong coupled threshold
48 AMG_max_aggregation      = 20     % Max size of aggregations
49 AMG_tentative_smooth     = 0.67   % Smoothing factor for tentative prolongation
50 AMG_smooth_filter        = OFF    % Switch for filtered matrix for smoothing
```

NOTE: Here we can not discuss the details of these parameters as a full discussion requires more understand of the underlying algorithms which we have completely omitted. So to learn more about, we refer to the Reference Manual.

Chapter 4

More Advanced Usage

In this chapter, we discuss a few more advanced features of FASP. We will discuss parallel versions of FASP and its build-in features for debugging purposes. These features will be helpful for people who would like to develop on the top of FASP. For users who only wish to call a few standard solvers, they can skip this chapter.

4.1 An OpenMP example

OpenMP[1] (Open Multiprocessing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Some preliminary OpenMP support has been included since the very beginning of FASP. We consistently improves and expands OpenMP support as multiprocessor architectures become the dominant desktop computing environment.

NOTE: By default, OpenMP is disabled in FASP. In order to turn it on, you need to modify Makefile a little bit; see §1.5.

After you build FASP with "openmp=yes", OpenMP is turned on and the number of threads is determined by the environment variable OMP_NUM_THREADS. For example, to use 8 threads in sh/bash, you need to set:

```
$ export OMP_NUM_THREADS=8
```

Then you use 8 threads for computation.

4.2 A CUDA example

To be added.

[1] Official website: http://openmp.org/

4.3 Predefined constants

It is important to notice that there are several predefined constants in FASP. Using these macros makes the program more uniform. These constants are defined in "base/include/messages.h":

```
1  /*! \file messages.h
2   *  \brief Definition of all kinds of messages, including error messages,
3   *         solver types, etc.
4   *
5   *  \note  This is internal use only.
6   *
7   *---------------------------------------------------------------------------
8   *  Created  by Chensong Zhang on 03/20/2010.
9   *  Modified by Chensong Zhang on 12/06/2011.
10  *  Modified by Chensong Zhang on 12/25/2011.
11  *  Modified by Chensong Zhang on 04/22/2012.
12  *  Modified by Ludmil Zikatanov on 02/15/2013: CG -> SMOOTHER_CG.
13  *  Modified by Chensong Zhang on 02/16/2013: GS -> SMOOTHER_GS, etc.
14  *  Modified by Chensong Zhang on 04/09/2013: Add safe krylov methods.
15  *  Modified by Chensong Zhang on 09/22/2013: Clean up Doxygen.
16  *---------------------------------------------------------------------------
17  *
18  */
19
20 #ifndef __FASP_MESSAGES__          /*-- allow multiple inclusions --*/
21 #define __FASP_MESSAGES__
22
23 /**
24  * \brief Definition of logic type
25  */
26 #define TRUE                    1  /**< logic TRUE */
27 #define FALSE                   0  /**< logic FALSE */
28
29 /**
30  * \brief Definition of switch
31  */
32 #define ON                      1  /**< turn on certain parameter */
33 #define OFF                     0  /**< turn off certain parameter */
34
35 /**
36  * \brief Print level for all subroutines -- not including DEBUG output
37  */
38 #define PRINT_NONE              0  /**< slient: no printout at all */
39 #define PRINT_MIN               1  /**< quiet: minimal print, like convergence */
40 #define PRINT_SOME              2  /**< some: print cpu time, iteration number */
41 #define PRINT_MORE              4  /**< more: print more useful information */
42 #define PRINT_MOST              8  /**< most: maximal printouts, no disk files*/
43 #define PRINT_ALL              10  /**< everything: all printouts allowed */
44
45 /**
46  * \brief Definition of matrix format
47  **/
48 #define MAT_FREE                0  /**< matrix-free format: only mxv action */
49 #define MAT_CSR                 1  /**< compressed sparse row */
50 #define MAT_BSR                 2  /**< blockwise compressed sparse row */
51 #define MAT_STR                 3  /**< structured sparse matrix */
52 #define MAT_bCSR                4  /**< block matrix of CSR */
53 #define MAT_bBSR                5  /**< block matrix of BSR for borded systems */
54 #define MAT_CSRL                6  /**< modified CSR to reduce cache missing */
55 #define MAT_SymCSR              7  /**< symmetric CSR format */
56
57 /**
```

```
58  * \brief Definition of return status and error messages
59  */
60 #define ERROR_OPEN_FILE        -10  /**< fail to open a file */
61 #define ERROR_WRONG_FILE       -11  /**< input contains wrong format */
62 #define ERROR_INPUT_PAR        -13  /**< wrong input argument */
63 #define ERROR_REGRESS          -14  /**< regression test fail */
64 #define ERROR_NUM_BLOCKS       -18  /**< wrong number of blocks */
65 #define ERROR_MISC             -19  /**< other error */
66 //-----------------------------------------------------------------------------
67 #define ERROR_ALLOC_MEM        -20  /**< fail to allocate memory */
68 #define ERROR_DATA_STRUCTURE   -21  /**< matrix or vector structures */
69 #define ERROR_DATA_ZERODIAG    -22  /**< matrix has zero diagonal entries */
70 #define ERROR_DUMMY_VAR        -23  /**< unexpected input data */
71 //-----------------------------------------------------------------------------
72 #define ERROR_AMG_INTERP_TYPE -30  /**< unknown interpolation type */
73 #define ERROR_AMG_SMOOTH_TYPE -31  /**< unknown smoother type */
74 #define ERROR_AMG_COARSE_TYPE -32  /**< unknown coarsening type */
75 #define ERROR_AMG_COARSEING   -33  /**< coarsening step failed to complete */
76 //-----------------------------------------------------------------------------
77 #define ERROR_SOLVER_TYPE      -40  /**< unknown solver type */
78 #define ERROR_SOLVER_PRECTYPE -41  /**< unknow precond type */
79 #define ERROR_SOLVER_STAG      -42  /**< solver stagnates */
80 #define ERROR_SOLVER_SOLSTAG   -43  /**< solver's solution is too small */
81 #define ERROR_SOLVER_TOLSMALL -44  /**< solver's tolerance is too small */
82 #define ERROR_SOLVER_ILUSETUP -45  /**< ILU setup error */
83 #define ERROR_SOLVER_MISC      -46  /**< misc solver error during run time */
84 #define ERROR_SOLVER_MAXIT     -48  /**< maximal iteration number exceeded */
85 #define ERROR_SOLVER_EXIT      -49  /**< solver does not quit successfully */
86 //-----------------------------------------------------------------------------
87 #define ERROR_QUAD_TYPE        -60  /**< unknown quadrature type */
88 #define ERROR_QUAD_DIM         -61  /**< unsupported quadrature dim */
89 //-----------------------------------------------------------------------------
90 #define RUN_FAIL               -99  /**< general failure */
91 #define SUCCESS                  0  /**< return from funtion successfully */
92
93 /**
94  * \brief Definition of solver types for iterative methods
95  */
96 #define SOLVER_CG                1  /**< Conjugate Gradient */
97 #define SOLVER_BiCGstab          2  /**< Biconjugate Gradient Stabilized */
98 #define SOLVER_MinRes            3  /**< Minimal Residual */
99 #define SOLVER_GMRES             4  /**< Generalized Minimal Residual */
100 #define SOLVER_VGMRES            5  /**< Variable Restarting GMRES */
101 #define SOLVER_VFGMRES           6  /**< Variable Restarting Flexible GMRES */
102 #define SOLVER_GCG               7  /**< Generalized Conjugate Gradient */
103 //-----------------------------------------------------------------------------
104 #define SOLVER_SCG              11  /**< Conjugate Gradient with safe net */
105 #define SOLVER_SBiCGstab        12  /**< BiCGstab with safe net */
106 #define SOLVER_SMinRes          13  /**< MinRes with safe net */
107 #define SOLVER_SGMRES           14  /**< GMRes with safe net */
108 #define SOLVER_SVGMRES          15  /**< Variable-restart GMRES with safe net */
109 #define SOLVER_SVFGMRES         16  /**< Variable-restart FGMRES with safe net */
110 #define SOLVER_SGCG             17  /**< GCG with safe net */
111 //-----------------------------------------------------------------------------
112 #define SOLVER_AMG              21  /**< AMG as an iterative solver */
113 #define SOLVER_FMG              22  /**< Full AMG as an solver */
114
115 /**
116  * \brief Definition of solver types for direct methods (requires external libs)
117  */
118 #define SOLVER_SUPERLU          31  /**< SuperLU Direct Solver */
119 #define SOLVER_UMFPACK          32  /**< UMFPack Direct Solver */
120 #define SOLVER_MUMPS            33  /**< MUMPS   Direct Solver */
121
```

```
122 /**
123  * \brief Definition of iterative solver stopping criteria types
124  */
125 #define STOP_REL_RES            1  /**< relative residual ||r||/||b|| */
126 #define STOP_REL_PRECRES        2  /**< relative B-residual ||r||_B/||b||_B */
127 #define STOP_MOD_REL_RES        3  /**< modified relative residual ||r||/||x|| */
128
129 /**
130  * \brief Definition of preconditioner type for iterative methods
131  */
132 #define PREC_NULL               0  /**< with no precond */
133 #define PREC_DIAG               1  /**< with diagonal precond */
134 #define PREC_AMG                2  /**< with AMG precond */
135 #define PREC_FMG                3  /**< with full AMG precond */
136 #define PREC_ILU                4  /**< with ILU precond */
137 #define PREC_SCHWARZ            5  /**< with Schwarz preconditioner */
138
139 /**
140  * \brief Definition of AMG types
141  */
142 #define CLASSIC_AMG             1  /**< classic AMG */
143 #define SA_AMG                  2  /**< smoothed aggregation AMG */
144 #define UA_AMG                  3  /**< unsmoothed aggregation AMG */
145
146 /**
147  * \brief Definition of cycle types
148  */
149 #define V_CYCLE                 1  /**< V-cycle */
150 #define W_CYCLE                 2  /**< W-cycle */
151 #define AMLI_CYCLE              3  /**< AMLI-cycle */
152 #define NL_AMLI_CYCLE           4  /**< Nonlinear AMLI-cycle */
153
154 /**
155  * \brief Definition of standard smoother types
156  */
157 #define SMOOTHER_JACOBI         1  /**< Jacobi smoother */
158 #define SMOOTHER_GS             2  /**< Gauss-Seidel smoother */
159 #define SMOOTHER_SGS            3  /**< symm Gauss-Seidel smoother */
160 #define SMOOTHER_CG             4  /**< CG as a smoother */
161 #define SMOOTHER_SOR            5  /**< SOR smoother */
162 #define SMOOTHER_SSOR           6  /**< SSOR smoother */
163 #define SMOOTHER_GSOR           7  /**< GS + SOR smoother */
164 #define SMOOTHER_SGSOR          8  /**< SGS + SSOR smoother */
165 #define SMOOTHER_POLY           9  /**< Polynomial smoother */
166 #define SMOOTHER_L1DIAG        10  /**< L1 norm diagonal scaling smoother */
167
168 /**
169  * \brief Definition of specialized smoother types
170  */
171 #define SMOOTHER_BLKOIL        11  /**< Used in monolithic AMG for black-oil */
172
173 /**
174  * \brief Definition of coarsening types
175  */
176 #define COARSE_RS               1  /**< Classical coarsening */
177 #define COARSE_CR               3  /**< Compatible relaxation */
178 #define COARSE_AC               4  /**< Aggressive coarsening */
179
180 /**
181  * \brief Definition of interpolation types
182  */
183 #define INTERP_DIR              1  /**< Direct interpolation */
184 #define INTERP_STD              2  /**< Standard interpolation */
185 #define INTERP_ENG              3  /**< energy minimization interp in C */
```

```
186
187 /**
188  * \brief Type of vertices (dofs) for C/F splitting
189  */
190 #define UNPT                    -1  /**< undetermined points */
191 #define FGPT                     0  /**< fine grid points  */
192 #define CGPT                     1  /**< coarse grid points */
193 #define ISPT                     2  /**< isolated points */
194
195 /**
196  * \brief Definition of smoothing order
197  */
198 #define NO_ORDER                 0  /**< Natural order smoothing */
199 #define CF_ORDER                 1  /**< C/F order smoothing */
200
201 /**
202  * \brief Type of ordering for smoothers
203  */
204 #define USERDEFINED              0  /**< USERDEFINED order */
205 #define CPFIRST                  1  /**< C-points first order */
206 #define FPFIRST                 -1  /**< F-points first order */
207 #define ASCEND                  12  /**< Asscending order */
208 #define DESCEND                 21  /**< Dsscending order */
209
210 /**
211  * \brief Type of ILU methods
212  */
213 #define ILUk                     1  /**< ILUk */
214 #define ILUt                     2  /**< ILUt */
215 #define ILUtp                    3  /**< ILUtp */
216
217 #endif                              /* end if for __FASP_MESSAGES__ */
218
219 /*---------------------------------------*/
220 /*--        End of File              --*/
221 /*---------------------------------------*/
```

4.4 The debug environment

To be added.

Bibliography

[1] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for automatic multigrid solution with application to geodetic computations, Report. *Inst. Comp. Studies Colorado State Univ*, 109:110, 1982.

[2] J. Brannick and L. Zikatanov. Algebraic multigrid methods based on compatible relaxation and energy minimization. *Lecture Notes in Computational Science and Engineering*, 55:15, 2007.

[3] M. Brezina, A. Cleary, R. Falgout, V. Henson, J. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge. Algebraic multigrid based on element interpolation (AMGe). *SIAM Journal on Scientific Computing*, 22(5):1570–1592, 2000.

[4] T. Chartier, R. Falgout, V. Henson, J. Jones, T. Manteuffel, S. McCormick, J. Ruge, and P. Vassilevski. Spectral AMGe (ρAMGe). *SIAM J. Sci. Comput.*, 25(1):1–26, 2003.

[5] R. Falgout and P. Vassilevski. On generalizing the algebraic multigrid framework. *SIAM J. Numer. Anal.*, 42(4):1669–1693 (electronic), 2004.

[6] V. Henson and P. Vassilevski. Element-free AMGe: general algorithms for computing interpolation weights in AMG. *SIAM J. Sci. Comput.*, 23(2):629–650, 2001.

[7] X. Hu, P. S. Vassilevski, and J. Xu. Comparative convergence analysis of nonlinear amli-cycle multigrid. *SIAM J. Num. Anal*, 51(2):1349–1369, 2013.

[8] O. E. Livne. Coarsening by compatible relaxation. *Numer. Linear Algebra Appl.*, 11(2-3):205–227, 2004.

[9] J. Mellor-crummey and J. Garvin. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *International Journal of High Performance Computing Applications*, 18(2):225—-236, 2004.

[10] A. Muresan and Y. Notay. Analysis of aggregation-based multigrid. *SIAM Journal on Scientific Computing*, 30(2):1082–1103, 2008.

[11] S. Pissanetzky. *Sparse matrix technology*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1984.

[12] J. Ruge and K. Stuben. Efficient solution of finite difference and finite element equations. In D. J. Paddon and H. Holstein, editors, *Multigrid Methods for Integral and Differential Equations*, volume 3, pages 169–212. Clarendon Press, 1985.

[13] J. Ruge and K. Stüben. Algebraic multigrid. *Multigrid methods*, 3:73–130, 1987.

[14] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[15] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.

[16] W. Wan, T. Chan, and B. Smith. An energy-minimizing interpolation for robust multigrid methods. *SIAM Journal on Scientific Computing*, 21(4):1632–1649, 2000.

[17] J. Xu. Fast Poisson-Based Solvers for Linear and Nonlinear PDEs Jinchao Xu. In *Proceedings of the International Congress of Mathematicians*, pages 2886–2912, 2010.

[18] J. Xu and L. Zikatanov. On an energy minimizing basis for algebraic multigrid methods. *Computing and Visualization in Science*, 7(3):121–127, 2004.