User's Guide for FASP

FASP Developer Team

June 13, 2012

Contents

Chapter 1

Introduction

ch:intro

1.1 What is FASP

sec:goal

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

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

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 LPGL. Some of the

applications contain contributions from and owned partially by other parties. We wish to keep as many parts open to public as possible. We only discuss the kernel functions in this user's guide.

Our strategy

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

- Stage 1. Fine grid 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 grid 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 grid 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 you can expect

sec:idea

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 modifications
- H(curl)/H(div) systems
- Maxwell equation
- MHD equation
- QCD problem

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

sec:how

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 do not provide details of the algorithms nor the implementation. Along this guide, we provide a reference manual 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.

1.4 How to obtain FASP

sec:install

All the FASP packages are hosted on *BitBucket.org*[1] using Mercurial (Hg)[2]. 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[3]. They've also had the advantage of trying the documentation on a fairly savvy group of developers (Mozilla) who gave them lots of feedback that helped polish the rough edges.

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 core subroutines via HTTPS"

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

[1] Official website: https://bitbucket.org/
[2] Official website: http://mercurial.selenic.com/
[3] The hgbook, http://hgbook.red-bean.com/

For the moment, it requires a password to access the Hg repository and the password is "multigrid". Very soon it will be open to public completely.

After a long pause[4], 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
```

Compilation

ssec:lib

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.

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

```
$ make
```

In order to make sure everything is OK, you can go to the "test" directory and type:

```
$ make testall
```

Then you should obtain a set of test problems. If you need more help, you can use

```
$ make help
```

External libraries

ssec:lib

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, SuperLU, MUMPS, dlmalloc, SparseKit.

[4] In fact, a very long pause. This is because the initial clone with copy all the history data which is about 125MB in total. Depending on the speed of your network, it could take 15 minutes to one hour.

Chapter 2

A Tutorial

ch:tutor

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.

To build the tutorial examples, just go to this directory and type:

```
$ make
```

This command will build all the examples inside tutorial. You can use "make help" to get more details.

2.1 The first example

sec:ex1

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 [**?**, **?**, **?**]; see §**??**. 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 Dirichlet boundary condition) on a simple quasi-uniform triangulation of the bounded domain Ω.

```
1  /*! \file poisson-amg.c
2   *  \brief The first test example for FASP: using AMG to solve
3   *         the discrete Poisson equation from P1 finite element.
4   *         C version.
5   */
6
7  #include "fasp.h"
8  #include "fasp_functs.h"
9
10 int main (int argc, const char * argv[])
11 {
12     input_param     inparam;  // parameters from input files
13     AMG_param       amgparam; // parameters for AMG
14
15     printf("\n=========================================");
16     printf("\n||   FASP: AMG example -- C version    ||");
17     printf("\n=========================================\n\n");
18
```

```
19      // Step 0. Set parameters
20      // Read input and AMG parameters from a disk file
21      // In this example, we read everything from a disk file:
22      //          "./ini/amg.dat"
23      // See the reference manual for details of the parameters.
24      fasp_param_init("ini/amg.dat",&inparam,NULL,&amgparam,NULL,NULL);
25
26      // Set local parameters using the input values
27      const int print_level = inparam.print_level;
28
29      // Step 1. Get stiffness matrix and right-hand side
30      // Read A and b -- P1 FE discretization for Poisson.
31      // The location of the data files are given in "amg.dat".
32      dCSRmat A;
33      dvector b, x;
34      char filename1[512], *datafile1;
35      char filename2[512], *datafile2;
36
37      // Read the stiffness matrix from matFE.dat
38      strncpy(filename1,inparam.workdir,128);
39      datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);
40
41      // Read the RHS from rhsFE.dat
42      strncpy(filename2,inparam.workdir,128);
43      datafile2="rhs_FE.dat"; strcat(filename2,datafile2);
44
45      fasp_dcsrvec2_read(filename1,filename2,&A,&b);
46
47      // Step 2. Print problem size and AMG parameters
48      if (print_level>PRINT_NONE) {
49          printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
50          printf("b: n = %d\n", b.row);
51          fasp_param_amg_print(&amgparam);
52      }
53
54      // Step 3. Solve the system with AMG as an iterative solver
55      // Set the initial guess to be zero and then solve it
56      // with AMG method as an iterative procedure
57      fasp_dvec_alloc(A.row, &x); fasp_dvec_set(A.row,&x,0.0);
58      fasp_solver_amg(&A, &b, &x, &amgparam);
59
60      // Step 4. Clean up memory
61      fasp_dcsr_free(&A);
62      fasp_dvec_free(&b);
63      fasp_dvec_free(&x);
64
65      return SUCCESS;
66  }
```

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 2–4 tells the Doxygen what is this function for.
- Line 7–8 includes the main FASP header file “fasp.h” and FASP function decoration header “fasp_functs.h”. These two headers shall be included in all the examples in this section. Please be noted that the function decorations in “fasp_functs.h” is automatically generated from the source files.
- Line 24 reads solver parameters from “tutorial/ini/amg.dat”; see more discussions in §**??**. In this files, 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.

- Line 32 defines a sparse matrix A in the compressed sparse row (CSR) format. Line 33 defines two vectors: the right-hand side b and the numerical solution x. We refer to §**??** for definitions of vectors and sparse matrices.
- Line 45 reads the matrix and the right-hand side from two disk files. Line 38–43 defines the filenames of them.
- Line 48–52 prints the information of the matrix and solver parameters.
- Line 57 allocates memory for the solution vector x and set its initial value to be all zero.
- Line 58 solves $Ax = b$ using the AMG method. Type the AMG method and other parameters have been given in "amgparam" at Line 24; see §**??**.
- Line 61–63 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:                     15
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
-----------------------------------------------------


-----------------------------------------------------
  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
Ruge-Stuben AMG setup costs 0.0087 seconds.
-------------------------------------------------------------------
It Num |    ||r||/||b||    |     ||r||      |  Conv. Factor
-------------------------------------------------------------------
     1 |    5.160015e-04   |  3.877420e-03  |     0.0000
     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 0.0080 seconds.
AMG totally costs 0.0173 seconds.
```

2.2 The second example

sec:ex2

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
6  #include "fasp.h"
7  #include "fasp_functs.h"
8
9  int main (int argc, const char * argv[])
10 {
11     input_param          inparam;  // parameters from input files
12     itsolver_param       itparam;  // parameters for itsolver
13
14     printf("\n=========================================");
15     printf("\n||   FASP: ITS example -- C version   ||");
16     printf("\n=========================================\n\n");
17
18     // Step 0. Set parameters
19     // Read input and AMG parameters from a disk file
20     // In this example, we read everything from a disk file:
21     //          "./ini/its.dat"
22     // See the reference manual for details of the parameters.
23     fasp_param_init("ini/its.dat",&inparam,&itparam,NULL,NULL,NULL);
24
25     // Set local parameters
26     const int print_level = inparam.print_level;
27
28     // Step 1. Get stiffness matrix and right-hand side
29     // Read A and b -- P1 FE discretization for Poisson.
30     // The location of the data files are given in "its.dat".
31     dCSRmat A;
32     dvector b, x;
33     char filename1[512], *datafile1;
34     char filename2[512], *datafile2;
35
36     // Read the stiffness matrix from matFE.dat
37     strncpy(filename1,inparam.workdir,128);
```

```
38      datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);
39
40      // Read the RHS from rhsFE.dat
41      strncpy(filename2,inparam.workdir,128);
42      datafile2="rhs_FE.dat"; strcat(filename2,datafile2);
43
44      fasp_dcsrvec2_read(filename1,filename2,&A,&b);
45
46      // Step 2. Print problem size and ITS parameters
47      if (print_level>PRINT_NONE) {
48          printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
49          printf("b: n = %d\n", b.row);
50          fasp_param_solver_print(&itparam);
51      }
52
53      // Step 3. Solve the system with ITS as an iterative solver
54      // Set the initial guess to be zero and then solve it using standard
55      // iterative methods, without applying any preconditioners
56      fasp_dvec_alloc(A.row, &x); fasp_dvec_set(A.row,&x,0.0);
57      fasp_solver_dcsr_itsolver(&A, &b, &x, NULL, &itparam);
58
59      // Step 4. Clean up memory
60      fasp_dcsr_free(&A);
61      fasp_dvec_free(&b);
62      fasp_dvec_free(&x);
63
64      return SUCCESS;
65  }
```

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

- Line 23 is slightly different than Line 24 in the previous example. This is because we are calling general interface of Krylov subspace methods [**?**] and the parameters are recorded in "itparam".
- Line 56 allocates memory for the solution vector x and set its initial value to be all zero.
- Line 57 solves $Ax = b$ using the general interface for Krylov subspace methods. Type the iterative method and other parameters have been given in "itparam"; see §**??** 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

sec:ex3

This example is slightly longer and is a modification of the previous one. In this example, we want to demonstrate how to setup a preconditioner for the conjugate gradient (CG) 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
7   #include "fasp.h"
```

```
8  #include "fasp_functs.h"
9
10 int main (int argc, const char * argv[])
11 {
12     input_param          inparam;  // parameters from input files
13     itsolver_param       itparam;  // parameters for itsolver
14     AMG_param            amgparam; // parameters for AMG
15     ILU_param            iluparam; // parameters for ILU
16
17     printf("\n=========================================");
18     printf("\n||   FASP: PCG example -- C version   ||");
19     printf("\n=========================================\n\n");
20
21     // Step 0. Set parameters
22     // Read input and precondition parameters from a disk file
23     // In this example, we read everything from a disk file:
24     //          "./ini/pcg.dat"
25     // See the reference manual for details of the parameters.
26     fasp_param_init("ini/pcg.dat",&inparam,&itparam,&amgparam,&iluparam,NULL);
27
28     // Set local parameters
29     const SHORT print_level = itparam.print_level;
30     const SHORT pc_type     = itparam.precond_type;
31     const SHORT stop_type   = itparam.stop_type;
32     const INT   maxit       = itparam.maxit;
33     const REAL  tol         = itparam.tol;
34
35     // Step 1. Get stiffness matrix and right-hand side
36     // Read A and b -- P1 FE discretization for Poisson.
37     // The location of the data files are given in "pcg.dat".
38     dCSRmat A;
39     dvector b, x;
40     char filename1[512], *datafile1;
41     char filename2[512], *datafile2;
42
43     // Read the stiffness matrix from matFE.dat
44     strncpy(filename1,inparam.workdir,128);
45     datafile1="csrmat_FE.dat"; strcat(filename1,datafile1);
46
47     // Read the RHS from rhsFE.dat
48     strncpy(filename2,inparam.workdir,128);
49     datafile2="rhs_FE.dat"; strcat(filename2,datafile2);
50
51     fasp_dcsrvec2_read(filename1,filename2,&A,&b);
52
53     // Step 2. Print problem size and PCG parameters
54     if (print_level>PRINT_NONE) {
55         printf("A: m = %d, n = %d, nnz = %d\n", A.row, A.col, A.nnz);
56         printf("b: n = %d\n", b.row);
57         fasp_param_solver_print(&itparam);
58     }
59
60     // Setp 3. Setup preconditioner
61     // Preconditioiner type is determined by pc_type
62     precond *pc = fasp_precond_setup(pc_type, &amgparam, &iluparam, &A);
63
64     // Step 4. Solve the system with PCG as an iterative solver
65     // Set the initial guess to be zero and then solve it using pcg solver
66     // Note that we call PCG interface directly. There is another way which
67     // calls the abstract iterative method interface; see possion-its.c for
68     // more details.
69     fasp_dvec_alloc(A.row, &x); fasp_dvec_set(A.row, &x, 0.0);
70     fasp_solver_dcsr_pcg(&A, &b, &x, pc, tol, maxit, stop_type, print_level);
71
```

```
72      // Step 5. Clean up memory
73      if (pc_type!=PREC_NULL) fasp_mem_free(pc->data);
74      fasp_dcsr_free(&A);
75      fasp_dvec_free(&b);
76      fasp_dvec_free(&x);
77
78      return SUCCESS;
79  }
```

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

- Line 26 reads parameters from "tutorial/ini/pcg.dat". In this example, we need parameters for iterative methods, AMG preconditioner, and ILU method. The type of the preconditioner is set in "pcg.dat" and recorded in "pc_type"; see Line 30.
- Line 62 sets up the desired preconditioner and prepare it for the preconditioned iterative methods.
- Line 70 calls PCG to solve $Ax = b$. One can also call the general iterative method interface in the previous example.

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

c:parameters

In the previous examples, we have seen how to read input parameters from disk files. Now we take a brief look inside those files and 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 stores the matrices and right-hand side vectors.
- 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 (§**??**).
- Line 56–59 gives the parameters for the setup phase of the aggregation-base AMG methods (§**??**).

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:                     15
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:      1
-----------------------------------------------------


-----------------------------------------------------
  Level     Num of rows      Num of nonzeros
-----------------------------------------------------
    0            3969             27281
    1             541              5121
    2              41               391
-----------------------------------------------------
AMG grid complexity     = 1.147
AMG operator complexity = 1.202
Smoothed Aggregation AMG setup costs 0.0038 seconds.
-------------------------------------------------------------------------
It Num |    ||r||/||b||    |     ||r||       |  Conv. Factor
-------------------------------------------------------------------------
     1 |    4.274024e-02   |  3.211655e-01   |     0.0000
     2 |    7.681829e-03   |  5.772402e-02   |     0.1797
     3 |    3.769909e-03   |  2.832845e-02   |     0.4908
     4 |    1.908833e-03   |  1.434365e-02   |     0.5063
     5 |    9.544638e-04   |  7.172183e-03   |     0.5000
     6 |    4.811769e-04   |  3.615735e-03   |     0.5041
     7 |    2.454538e-04   |  1.844428e-03   |     0.5101
     8 |    1.264489e-04   |  9.501825e-04   |     0.5152
     9 |    6.560382e-05   |  4.929706e-04   |     0.5188
    10 |    3.419782e-05   |  2.569747e-04   |     0.5213
    11 |    1.788119e-05   |  1.343657e-04   |     0.5229
    12 |    9.367771e-06   |  7.039279e-05   |     0.5239
    13 |    4.913626e-06   |  3.692274e-05   |     0.5245
    14 |    2.579247e-06   |  1.938138e-05   |     0.5249
    15 |    1.354509e-06   |  1.017827e-05   |     0.5252
    16 |    7.115253e-07   |  5.346656e-06   |     0.5253
    17 |    3.738252e-07   |  2.809056e-06   |     0.5254
    18 |    1.964204e-07   |  1.475973e-06   |     0.5254
    19 |    1.032110e-07   |  7.755642e-07   |     0.5255
    20 |    5.423451e-08   |  4.075375e-07   |     0.5255
    21 |    2.849901e-08   |  2.141518e-07   |     0.5255
    22 |    1.497562e-08   |  1.125322e-07   |     0.5255
```

```
    23 |    7.869358e-09    |  5.913318e-08  |      0.5255
Number of iterations = 23 with relative residual 7.869358e-09.
AMG solve costs 0.0140 seconds.
AMG totally costs 0.0184 seconds.
```

You can compare this with the sample results in §**??**.

NOTE: The input parameters allowed in FASP are not limited to the ones used in this example. For more parameters and their ranges, we refer to the Reference Manual.

Chapter 3

Basic Usage

ch:basic

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

3.1 Vectors and sparse matrices

sec:blas

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 [**?**].

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

ex:sparse **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 **??** 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 some storage and has no 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 [**?**]. 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 **??** 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
$$val = \| \ 1. \ | \ 1.5 \ | \ 12. \ \| \ 1. \ | \ 7. \ | \ 6. \ | \ 1. \ \| \ 3. \ | \ 6. \ \| \ 2. \ | \ 5. \ | \ 1. \ \|$$

NOTE: The indexes in JA and entries of val need not 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 another "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 [**?**] 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;
```

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

} dCSRLmat; /**< Sparse matrix of REAL type in CSRL format */
```

(iv) Block sparse matrix formats

For PDE applications, we often need to solve systems of partial differential equations. Many iterative methods and preconditioners could take advantages of fact and improve efficiency. So we often need to use block matrices 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: BSR (or Block Compressed Sparse Row) and CSR Block (Block of CSR matrices). The definitions of these types of block matrices are given as follows:

```
/**
 * \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
```

```
32      //! in row-major order and the size is (NNZ*nb*nb).
33      REAL *val;
34
35      //! integer array of row pointers, the size is ROW+1
36      INT *IA;
37
38      //! Element i of the integer array columns is the number of the column in the
39      //! block matrix that contains the i-th non-zero block. The size is NNZ.
40      INT *JA;
41
42  } dBSRmat; /**< Matrix of REAL type in BSR format */
```

```
1   /**
2    * \struct block_dCSRmat
3    * \brief Block of REAL CSR matrix structure.
4    *
5    * CSR Block Format in REAL
6    *
7    * \note The starting index of A is 0.
8    */
9   typedef struct block_dCSRmat{
10
11      //! row number of blocks in A, m
12      INT brow;
13      //! column number of blocks A, n
14      INT bcol;
15      //! blocks of dCSRmat, point to blocks[brow][bcol]
16      dCSRmat **blocks;
17
18  } block_dCSRmat; /**< Matrix of REAL type in Block BSR format */
```

NOTE: For more details as well as other specialized block matrices, we refer to "core/include/-fasp_block.h".

Sparse BLAS

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

```
1   /**
2    * \fn void fasp_blas_dcsr_mxv (dCSRmat *A, REAL *x, REAL *y)
3    *
4    * \brief Matrix-vector multiplication y = A*x
5    *
6    * \param A   Pointer to dCSRmat matrix A
7    * \param x   Pointer to array x
8    * \param y   Pointer to array y
9    *
10   * \author Chensong Zhang
11   * \date   07/01/2009
12   */
13  void fasp_blas_dcsr_mxv (dCSRmat *A,
14                           REAL *x,
15                           REAL *y)
16  {
17      const INT    m  = A->row;
18      const INT   *ia = A->IA, *ja = A->JA;
19      const REAL *aj = A->val;
20
```

```
21      INT i, k, beg, end;
22      register REAL tmp;
23
24      for ( i=0; i<m; ++i ) {
25          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.

3.2 Iterative methods

sec:iter

In FASP, there are a couple of standard preconditioned iterative methods [**?**] 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 "core/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 §**??**. 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 */
```

3.3 Geometric multigrid

sec:gmg

To be added.

3.4 Algebraic multigrid

sec:amg

The classical algebraic multigrid method [**?**] 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] have emerged in recent years.

The following code segment is part of "core/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, 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;
```

```
36          break;
37
38      case NL_AMLI_CYCLE: // call Nonlinear AMLI-cycle
39          if ( (status=fasp_amg_solve_nl_amli(mgl, param)) < 0 ) goto FINISHED;
40          break;
41
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 §**??**.
- 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
```

```
AMG_relaxation           = 1.1    % relaxation parameter for SOR smoother
AMG_polynomial_degree    = 3      % degree of the polynomial smoother
AMG_presmooth_iter       = 1      % number of presmoothing sweeps
AMG_postsmooth_iter      = 1      % number of postsmoothing sweeps

%-------------------------------------------------------%
% parameters for classical AMG SETUP                    %
%-------------------------------------------------------%

AMG_coarsening_type      = 1      % 1 Modified RS
                                  % 3 Compatible Relaxation
                                  % 4 Aggressive
AMG_interpolation_type   = 1      % 1 Direct | 2 Standard | 3 Energy-min
AMG_strong_threshold     = 0.25   % Strong threshold
AMG_truncation_threshold = 0.4    % Truncation threshold
AMG_max_row_sum          = 0.9    % Max row sum

%-------------------------------------------------------%
% parameters for aggregation-type AMG SETUP             %
%-------------------------------------------------------%

AMG_strong_coupled       = 0.08   % Strong coupled threshold
AMG_max_aggregation      = 20     % Max size of aggregations
AMG_tentative_smooth     = 0.67   % Smoothing factor for tentative prolongation
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

ch:advanced

In this chapter, we discuss a few topics for more advanced users.

4.1 Block preconditioners

sec:block

To be added.

4.2 An OpenMP example

sec:mop

To be added.

4.3 A CUDA example

sec:cuda

To be added.

4.4 The debug environment

sec:debug

To be added.

Appendix

ch:append

List of Data Structures

List of Functions

Bibliography

rmick.1982uq [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.

tanov.2007zr [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.

leary.2000ly [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.

lgout.2003ve [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 (electronic), 2003.

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

evski.2001cr [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 (electronic), 2001. Copper Mountain Conference (2000).

Livne.2004kl [7] O. E. Livne. Coarsening by compatible relaxation. *Numer. Linear Algebra Appl.*, 11(2-3):205–227, 2004.

-crummey2004 [8] 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.

Notay.2008tg [9] A. Muresan and Y. Notay. Analysis of aggregation-based multigrid. *SIAM Journal on Scientific Computing*, 30(2):1082–1103, 2008.

etzky.1984hc [10] S. Pissanetzky. *Sparse matrix technology*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1984.

tuben.1985ij [11] 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.

tuben.1987bs [12] J. Ruge and K. Stüben. Algebraic multigrid. *Multigrid methods*, 3:73–130, 1987.

.Saad.2003fv [13] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

andel.1996kl [14] 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.

nChan.2000qa [15] 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.

Xu.Xu.2010ff [16] J. Xu. Fast Poisson-Based Solvers for Linear and Nonlinear PDEs Jinchao Xu. In *PROCEEDINGS OF THE INTERNATIONAL CONGRESS OF MATHEMATICIANS 2010*, number 2000, pages 2886–2912, 2010.

tanov.2004pi [17] J. Xu and L. Zikatanov. On an energy minimizing basis for algebraic multigrid methods. *Computing and Visualization in Science*, 7(3):121–127, 2004.