NOTE ON LINEAR SYSTEM SOLVER

CHANGJIAN XIE

In general, we can model a PDE problem and then discretize it into a linear/nonlinear system to be solved. Here, we will focus on the solver to the linear system of equations that has a sparse structure. If not sparse systems, say dense matrix. we can recall the BLAS, Lapack or ScaLapack.

1. Introduction

In the area of scientific computing and engineering, very often, one need to solve the linear system of equations

$$Ax = b, \tag{1}$$

say,

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n = b_2 \\ \cdots \quad \cdots \quad \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n = b_m \end{cases} \tag{2}$$

where the matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{3}$$

and vector $x = (x_1, x_2, \cdots, x_n)^T$, $b = (b_1, b_2, \cdots, b_m)^T$

The matrix involved in these problems is usually a matrix with only a few elements that are not zero, that is, a sparse matrix. In practical applications, the scale of matrix is very large. This linear system solving problem has been studied for several decades.

2. Useful linear solvers

2.1. **direct methods.** This sort of methods are a variant of the Gaussian elimination method, involved the entries by matrix computation, such as LU, QR, or Cholesky decomposition. This method can be achived very high level accuracy, but is slow when handle large scale sparse matrix. For example, we use $x = A\backslash b$ since it can select the appropriate solver in `matlab`.

Date: November 25, 2020.

2.2. **iterative methods.** We can solve the linear system of equations using iterative methods if A is a large sparse matrix in order to attain the balance between running time and accuracy of solutions to approximate the solutions. Usually, we proceed

$$M^{-1}Ax = M^{-1}b$$

since it can decrease the condition number of the matrix. M is preconditioner, $M^{-1}A$ has small condition number. This would be efficient, easy to construct, easy to solve. We can also accelerate the solving efficiency by usage of paralleling. Consider solving the $N \times N$ linear system $Ax = b$. Most iterative methods, however, have the following form, where $r_k = b - Ax_k$ is the residual at iteration k,

$$x_{k+1} = x_k + M^{-1}r_k. \tag{4}$$

Let $e_k = x - x_k$, be the error, and note that $r_k = Ae_k$. The error propagation for the iterative method is

$$e_{k+1} = (I - M^{-1}A)e_k. \tag{5}$$

Most of iterative methods for solving linear equations follow up a similar steps:

- Guess a solution vector x_0 (usually, set $x_0 = 0$),
- calculate the norm of residual error $res = norm(b - Ax_0)$,
- If $res < tol$, where tol denotes the tolerance, obtain approximated solution x_0, else
- update x_0 according to residual error and repeat steps until $res < tol$.

The useful iterative solvers are as below.

a) `lsqr` least squares method. This solver requires square matrix that corresponds to PCG of $(A'A)x = A'b$.
b) `pcg` preconditioned conjugate gradient method. The coefficient matrix must be symmetric positive definite matrix. It is the most efficient solver to the positive definite linear systems of equations since only store finite number of vectors
(c) `minres` minimal residual method. The coefficient matrix must be symmetric, but do not need to be positive definite. To minimize the residual of 2-norm. This method generally can not breakdown.
(d) `symmlq` symmetric LQ. The coefficient matrix must be symmetric, but do not need to be positive definite. This method generally can not breakdown.
(e) `bicg` bi-conjugate gradient. The matrix must be square, but do not need to be symmetric. The cost of `bicg` is low. But convergence can not be reliable.
(f) `bicgstab` bicg with stability. The matrix must be square, but do not be symmetric. Using bicg and GMRES alternatively to improve stability.
(g) `cgs` conjugate gradient square method. The matrix must be square, but do not be symmetric. This method need same number of iterations with bicg.
(h) `gmres` generalized minimal residual method. The matrix must be square, but do not be symmetric. The one of most reliable methods since minimize the residual each iteration.
(i) `qmr` quasi-minimal residual method. The matrix must be square, but do not be symmetric. The cost is slightly higher than `bicg`, but has good stability.

Besides, one can accelerate the convergence use some preconditioner, `ilu` and `ichol` for example.

We can illustrate the solver as in fig. 1. In general, one can use `gmres` for the square non-symmetric problems. In some cases, the efficiency of `bicg, bicgstab, cgs` etc is better than `gmres`. But the convergence behavior is not clear.

There are a lot of linear solver library, such as *hypre, petsc, trilinos, pardiso, mumps, slepc* so on and so forth.

We can classify the linear solvers as below.

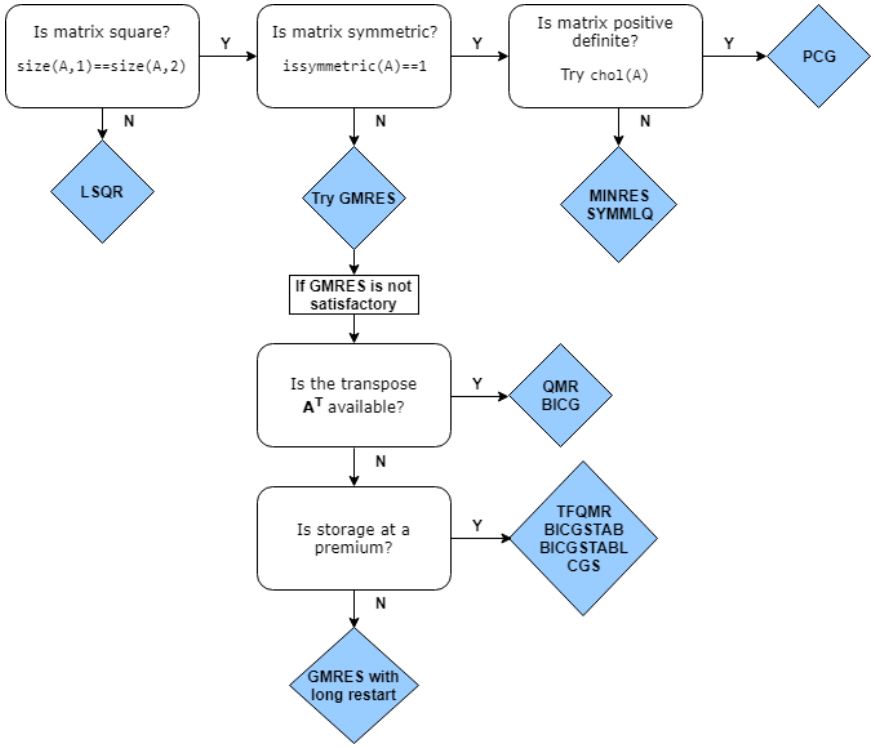

FIGURE 1. The schematic of solver

- Iterative linear solvers: Krylov (CG, GMRES, BICG, BICGSTAB and ORTHOMIN) and AMG (classical AMG, smoothed aggregation AMG, adptive AMG);
- Direct solvers: LLt, LDLt, LU
- Eigenvalue problems: LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) which is a simple, yet very efficient algorithm for computing smallest eigenpairs of the symmetrix generalized eigenvalue problem $Ax = \lambda Bx$ with large, possibly sparse, symmetric matrix A and symmetric positive definite matrix B. The matrix A is not assumed to be positive, which also allows one to use LOBPCG to compute the largest eigenpairs of $Ax = \lambda Bx$ simply by solving $-Ax = \mu Bx$ for the smallest eigenvalues $\mu = -\lambda$.

One thing about how to choose the best linear solver is that the condition number, scalability, memory, the structure of the matrix, etc. The availability of solving linear system of equation is close related to the condition number of coefficient matrix. Say, we can solve it easier when the condition number smaller.

The parallel linear solver packages are

- Direct solvers: Pardiso, PaStiX, MUMPS and SuperLU
- Iterative linear solvers: PETSc, Hypre, Trilinos
- Eigenvalue problems: SLEPc, ScaEigenPack

The url link of these package can be found in this table:

Here, the direct solver mainly use the LU decomposition or any other ways. The iterative solver contains Krylov subspace, Algebraic Multi-Grid. One can find the method of Jacobian,

TABLE 1. The downloaded link of packages

MUMPS	http://mumps.enseeiht.fr
SuperLU	https://portal.nersc.gov/project/sparse/superlu/
PETSc	https://www.mcs.anl.gov/petsc/
Hypre	https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-metho
Trilinos	https://trilinos.github.io
Pardiso	https://www.pardiso-project.org
PaStiX	http://pastix.gforge.inria.fr/files/README-txt.html
SLEPc	https://slepc.upv.es

Gauss-Seidel, SOR, or weighted SOR in the textbook. But these kind of methods can not be applied into the large scale problems.

We briefly introduce some packages as below.

I. **Pardiso** This package support sequential, shared memory and MPI, say, SMP and cluster (OpenMP and MPI). It contains iterative and direct solvers. The linear systems could be *sparse, symmetric, unsymmetric*. It has Fortran and C interface. The data type would be real, complex, single and double precision.

II. **PaStiX** This package support the Cluster, MPI and pthread. It's direct solver would be LU, LLt for symmetric matrix (Lt denotes transpose). The matrix would be *sparse, symmetric, and unsymmetric*. It has Fortran and C interface. The data type would be real, complex, single and double precision.

III. **MUMPS** This package was developed by France. We can proceed MPI and sequential process. It is direct solver that contains Multi frontal, LU, and LDLt. The matrix could be *sparse, SPD, symmetric and unsymmetric*. It has Fortran and C interface, and Matlab interface. The data type would be real, complex, single and double precision. It is more efficient than other direct methods.

IV. **SuperLU** This package developed by Xiaoye Li who graduated from Qinghua University that support sequential, shared memory (SMP), cluster and MPI. It is sparse direct solver using LU decomposition. It can handle general unsymmetric matrix. It has C iterface. The data type would be real, complex, single and double precision.

V. **PETSc** This package supports MPI with pthread, MPI with GPU. It provide Krylov subspace method, preconditioner, Newton's method (nolinear solver), trust region method, etc. It contains iterative linear and nonlinear solvers. It can handle ODE, PDE problems. It has C, C++, and Fortran interface. The PETSc numerical components are illustrated in fig. 2.

VI. **Hypre** This package called High Performance Preconditioners which developed by UC and CASC of LLNL is focused on the parallel compting and MPI. It contains preconditioner, Krylov solvers and AMG (BoomerAMG). It can be applied into the structural, and unstructural grid. It supports C (C++, Fortran) interface. The different solver for different grid can be shown in fig. 3.

3. LIBRARY HYPRE TUTORIAL

hypre provides several of the most commonly used Krylov-based iterative methods to be used in conjunction with its scalable preconditioners. This includes methods for nonsymmetric systems such as GMRES and methods for symmetric matrices such as Conjugate Gradient.

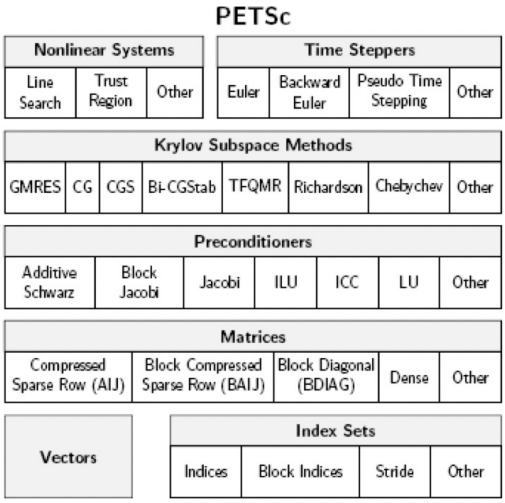

FIGURE 2. PETCs numerical components

Solvers	System Interfaces			
	Struct	SStruct	FEI	IJ
Jacobi	X	X		
SMG	X	X		
PFMG	X	X		
Split		X		
SysPFMG		X		
FAC		X		
Maxwell		X		
BoomerAMG		X	X	X
AMS		X	X	X
ADS		X	X	X
MLI		X	X	X
ParaSails		X	X	X
Euclid		X	X	X
PILUT		X	X	X
PCG	X	X	X	X
GMRES	X	X	X	X
FlexGMRES	X	X	X	X
LGMRES	X	X		X
BiCGSTAB	X	X	X	X
Hybrid	X	X	X	X
LOBPCG	X	X		X

FIGURE 3. Current solver availability via *hypre* conceptual interfaces. *X* denotes this sort of solver can support the system interfaces.

- The iterative method contains Krylov solver (CG, GMRES, TFQMR, BiCGSTAB); BoomerAMG (a parallel AMG solver)
- preconditioner:
  - Diagonal: block Jacobi preconditioner;

- PILUT: parallel incomplete LU decomposition (PILU) with threshold;
- Euclid: Extension of parallel ILU preconditioner
- SMG: semi-coarsening multigrid preconditioner; smoother would be line relaxation and plane relaxation in 2D and 3D case
- PFMG: semi-coarsening multigrid preconditioner, smoother would be simple point relaxation
- BoomerAMG: parallel AMG preconditioner
- ParaSails: parallel sparse approximated inverse preconditioner

These include stencil-based structured/semi-structured interfaces most appropriate for finite-difference applications; a finite-element based unstructured interface; and a linear-algebra based interface. Each conceptual interface provides access to several solvers without the need to write new interface code.

3.1. **How to install hypre.** One can access here https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/software to download it. Then, go to the Download directory, type `./configure` and type `make` to compile in the top-level source directory and then type `make install`.

The example codes demonstrate the following general structure of the application calls to hypre:

- **Build any necessary auxiliary structures for your chosen conceptual interface.** This includes, e.g., the grid and stencil structures if you are using the structured-grid interface.
- **Build the matrix, solution vector, and right-hand-side vector through your chosen conceptual interface.** Each conceptual interface provides a series of calls for entering information about your problem into hypre.
- **Build solvers and preconditioners and set solver parameters (optional).**
- **Call the solve function for the solver.**
- **Retrieve desired information from solver.** Depending on your application, there may be different things you may want to do with the solution vector. Also, performance information such as number of iterations is typically available, though it may differ from solver to solver.

3.2. **How to use hypre.** In general, a solver (let's call it `SOLVER`) is set up and run using the following routines, where A is the matrix, b the right hand side and x the solution vector of the linear system to be solved:

```
/* Create Solver */
int HYPRE_SOLVERCreate(MPI_COMM_WORLD, &solver);

/* set certain parameters if desired */
HYPRE_SOLVERSetTol(solver, 1.e-8);

/* Set up Solver */
HYPRE_SOLVERSetup(solver, A, b, x);

/* Solve the system */
HYPRE_SOLVERSolve(solver, A, b, x);

```

```
/* Destroy the solver */
HYPRE_SOLVERDestroy(solver);
```

3.2.1. *Structured-Grid System Interface (Struct).* The interface uses a finite difference or finite volume style, and currently supports only scalar PDEs (i.e., one unknown per gridpoint). There are four basic steps involved in setting up the linear system to be solved:

- set up the grid,
- set up the stencil,
- set up the matrix,
- set up the rhs vector.

The detailed calling function in each part could be found in the tutorial.

3.2.2. *Semi-Structured-Grid System Interface (SStruct).* The `SStruct` interface is appropriate with grids that are mostly (but not entirely) structured, e.g., block-structured grids, composite grids in structured adaptive mesh refinement (AMR) applications. In addition, it supports more general PDEs than the `Struct` interface by allowing multiple variables (system PDEs) and multiple variable types. There are five basic steps involved in setting up the linear system to be solved:

- set up the grid,
- set up the stencils (if needed),
- set up the graph,
- set up the matrix,
- set up the rhs vector.

3.2.3. *Finite element interface (FE).* Many applications codes uses unstructured finite element meshes that shown in fig. 4.

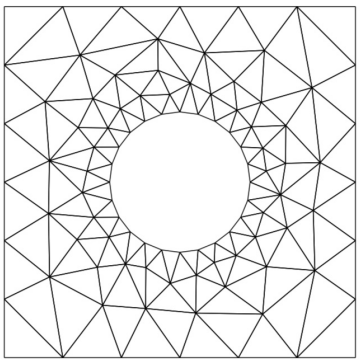

FIGURE 4. Example of an unstructured mesh.

We ignore the description since it is more complicated than previous two interfaces. Typically, FE interface codes contain data structures storing element connectivities, element stiffness, element stiffness matrices, element loads, boundary conditions, nodal coordinates, etc.

3.2.4. *Linear-Algebraic System Interface (IJ).* The `IJ` interface is the lowest common denominator for specifying linear systems in `hypre`. This interface provides access to general sparse-matrix solvers in `hypre`, not to the specialized solvers that requires more problem info.

As with the other interfaces in `hypre`, the `IJ` interface expects to get data in distributed form since this is the only scalable approach for assembling matrices on thousands of processes. Matrices are assumed to be distributed by blocks of rows as follows:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{P-1} \end{pmatrix} \tag{6}$$

here, the martix is distributed across the P processes, $0, 1, \cdots, P-1$ by blocks of rows. Each submatrix A_p is "owned" by a single process and its first and last row numbers are given by the global indices `ilower` and `iupper` in the `Create()` call below.

We give the example code to use the `IJ` interface:

- First building matrices

```
MPI_Comm            comm;
HYPRE_IJMatrix      ij_matrix;
HYPRE_ParCSRMatrix  parcsr_matrix;
int                 ilower, iupper;
int                 jlower, jupper;
int                 nrows;

int                 *ncols
int                 *rows
int                 *cols
double              *values

HYPRE_IJMatrixCreate(comm, ilower, iupper, jlower, jupper, &ij_matrix);
HYPRE_IJMatrixSetObjectType(ij_matrix, HYPRE_PARCSR);
HYPRE_IJMatrixInitialize(ij_matrix);

/* set matrix coefficients */
HYPRE_IJMatrixSetValues(ij_matrix, nrows, ncols, rows, cols, values);

/* add-to matrix cofficients, if desired */
HYPRE_IJMatrixAddToValues(ij_matrix, nrows, ncols, rows, cols, values);

HYPRE_IJMatrixAssemble(ij_matrix);
HYPRE_IJMatrixGetObject(ij_matrix, (void **) &parcsr_matrix);
```

- Then building rhs vector

```
MPI_Comm          comm;
HYPRE_IJVector    ij_vector;
HYPRE_ParVector   par_vector;
int               jlower, jupper;
int               nvalues;
```

```
int              *indices;
double           *values;

HYPRE_IJVectorCreate(comm, jlower, jupper, &ij_vector);
HYPRE_IJVectorSetObjectType(ij_vector, HYPRE_PARCSR);
HYPRE_IJVectorInitialize(ij_vector);

/* set vector values */
HYPRE_IJVectorSetValues(ij_vector, nvalues, indices, values);

HYPRE_IJVectorAssemble(ij_vector);
HYPRE_IJVectorGetObject(ij_vector, (void **) &par_vector);
```

3.3. **Data-types of hypre.** To begin with hypre, we should create the matrix by

`call HYPRE_IJMatrixCreate(mpi_comm, ilower, iupper,ilower,iupper,A,ierr)`,

we then initialize before setting coefficients by `call HYPRE_IJMatrixInitialize(A,ierr)`.

To assemble the matrix A, one should specify the row and column index and values of nonzero entries. Once we have index `cols` and `values`, we can set the values for row i of matrix A with command `call HYPRE_IJMatrixSetValues(A,1,nnz-1,i,cols,values, ierr)` with nnz the number of nonzero entries. We then assemble the matrix by

`call HYPRE_IJMatrixAssemble(A,ierr)`.

Other things can be done step by step.

3.4. **Data-types of MUMPS.** `MUMPS` ("MUltifrontal massively Parallel Solver") is a package for solving systems of linear equations of the form $Ax = b$, where A is a square matrix that can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed memory computers. `MUMPS` implements a direct method based on a multifrontal approach which performs a Gaussian factorization

$$A = LU, \tag{7}$$

where L is a lower triangular matrix and U an upper triangular matrix. If the matrix is symmetric then factorization

$$A = LDL^T, \tag{8}$$

where D is block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed.

We give an example of solving linear systems of equations as below.

Example 3.1 (assembled `DOUBLE PRECISION` problems)**.** *Consider $Ax = b$ with*

$$A = \begin{pmatrix} 2 & 3 & 4 & 0 & 0 \\ 3 & 0 & -3 & 0 & 6 \\ 0 & -1 & 1 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 20 \\ 24 \\ 9 \\ 6 \\ 13 \end{pmatrix} \tag{9}$$

Two files should be included in the program: `mpif.h` for MPI and `mumps_struc.h` for MUMPS. The file `mumps_root.h` shold be available since it is included in `mumps_struc.h` The initialization and termination of MPI are performed in the user program via the calls to `MPI_INIT` and `MPI_FINALIZE`.

Specifically, This package is initialized by calling `MUMPS` *with* `JOB=-1`*, the problem is read in by the host, (in the components* N*,* NNZ*,* IRN*,* JCN*,* A *and* RHS*), and the solution is computed in* `RHS` *with a call on all processors to* `MUMPS` *with* `JOB=6`*. A call finally with* `JOB=-2` *is performed to dellocate the data structures used by the instances of the package.*

We should input an format file as below.

```
5            :N
12           :NNZ
1 2 3.0
2 3 -3.0
4 3 2.0
5 5 1.0
2 1 3.0
1 1 2.0
5 2 4.0
3 4 2.0
2 5 6.0
3 2 -1.0
1 3 4.0
3 3 1.0      :A
20.0
24.0
9.0
6.0
13.0         :RHS
```

We then give the code as follows,

```
    PROGRAM MUMPS_TEST
    IMPLICIT NONE
    INCLUDE 'mpif.h'
    INCLUDE 'dmumps_struc.h'
    TYPE (DMUMPS_STRUC) mumps_par
    INTEGER IERR, I
    INTEGER(8) I8
    CALL MPI_INIT(IERR)
C Define a communicator for the package.
    mumps_par%COMM = MPI_COMM_WORLD
C Initialize an instance of the package
C for L U factorization (sym = 0, with working host)
    mumps_par%JOB = -1
    mumps_par%SYM = 0
    mumps_par%PAR = 1
    CALL DMUMPS(mumps_par)
    IF (mumps_par%INFOG(1).LT.0) THEN
     WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
     &  " mumps_par%INFOG(1)= ", mumps_par%INFOG(1),
     &  " mumps_par%INFOG(2)= ", mumps_par%INFOG(2)
     GOTO 500
```

```
22      END IF
23
24  C Define problem on the host (processor 0)
25      IF ( mumps_par%MYID .eq. 0 ) THEN
26       READ(5,*) mumps_par%N
27       READ(5,*) mumps_par%NNZ
28       ALLOCATE( mumps_par%IRN ( mumps_par%NNZ ) )
29       ALLOCATE( mumps_par%JCN ( mumps_par%NNZ ) )
30       ALLOCATE( mumps_par%A( mumps_par%NNZ ) )
31       ALLOCATE( mumps_par%RHS ( mumps_par%N ) )
32       DO I8 = 1, mumps_par%NNZ
33        READ(5,*) mumps_par%IRN(I8),mumps_par%JCN(I8),mumps_par%A(I8)
34       END DO
35       DO I = 1, mumps_par%N
36        READ(5,*) mumps_par%RHS(I)
37       END DO
38      END IF
39  C Call package for solution
40  mumps_par%JOB = 6
41      CALL DMUMPS(mumps_par)
42      IF (mumps_par%INFOG(1).LT.0) THEN
43       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
44       & " mumps_par%INFOG(1)= ", mumps_par%INFOG(1),
45       & " mumps_par%INFOG(2)= ", mumps_par%INFOG(2)
46       GOTO 500
47      END IF
48  C Solution has been assembled on the host
49      IF ( mumps_par%MYID .eq. 0 ) THEN
50       WRITE( 6, * ) ' Solution is ',(mumps_par%RHS(I),I=1,mumps_par%N)
51      END IF
52  C Deallocate user data
53      IF ( mumps_par%MYID .eq. 0 )THEN
54       DEALLOCATE( mumps_par%IRN )
55       DEALLOCATE( mumps_par%JCN )
56       DEALLOCATE( mumps_par%A )
57       DEALLOCATE( mumps_par%RHS )
58      END IF
59  C Destroy the instance (deallocate internal data structures)
60      mumps_par%JOB = -2
61      CALL DMUMPS(mumps_par)
62      IF (mumps_par%INFOG(1).LT.0) THEN
63       WRITE(6,'(A,A,I6,A,I9)') " ERROR RETURN: ",
64       & " mumps_par%INFOG(1)= ", mumps_par%INFOG(1),
65       & " mumps_par%INFOG(2)= ", mumps_par%INFOG(2)
66       GOTO 500
67      END IF
68  500 CALL MPI_FINALIZE(IERR)
```

We can run it by the following codes

```
* Supposing the MUMPS libraries with appropriate arithmetic have been
  ↪  generated, you may compile the example drivers by typing either
    make (which defaults to make d)
    make s
    make d
    make c
    make z
    make multi
or make all

* For the small Fortran driver, see comments in simpletest.F and try for
  ↪  example
    "mpirun -np 2 ./ssimpletest < input_simpletest_real"
    "mpirun -np 2 ./dsimpletest < input_simpletest_real"
    "mpirun -np 2 ./csimpletest < input_simpletest_cmplx"
    "mpirun -np 2 ./zsimpletest < input_simpletest_cmplx"
  if you are using the parallel version of MUMPS, or
    "./ssimpletest < input_simpletest_real"
    "./dsimpletest < input_simpletest_real"
    "./csimpletest < input_simpletest_cmplx"
    "./zsimpletest < input_simpletest_cmplx"

  if you are using the sequential version.
  The solution should be (1,2,3,4,5)
* For the small C driver, only an example using double arithmetic is available.
  Try for example
   "mpirun -np 3 ./c_example" (parallel version),or
   "./c_example" (sequential version).
  The solution should be (1,2)
* For multiple instances using different arithmetics, a small example is
  ↪  available in multiple_arithmetics_example.F.
Supposing the MUMPS libraries with all arithmetic have been generated,
you may compile the example driver by typing :
  make multi
  Then try for example:
    "mpirun -np 3 ./multiple_arithmetics_example" (parallel version),or
    "./multiple_arithmetics_example" (sequential version).
```

3.5. **An example of hypre.** In the tutorial of hypre, there are a lot of example codes like diffusion equation, Maxwell equation and Poisson equation, etc.

We can take 2D Laplacian equation with zero boundary conditions as an example.

```
!c
!c   Example 5
!c
!c   Interface:    Linear-Algebraic (IJ), Fortran (95) version
!c
```

```
6  !c   Compile with: make ex5f
7  !c
8  !c   Sample run:   mpirun -np 4 ex5f
9  !c
10 !c   Description:  This example solves the 2-D
11 !c                 Laplacian problem with zero boundary conditions
12 !c                 on an nxn grid.  The number of unknowns is N=n^2.
13 !c                 The standard 5-point stencil is used, and we solve
14 !c                 for the interior nodes only.
15 !c
16 !c                 This example solves the same problem as Example 3.
17 !c                 Available solvers are AMG, PCG, and PCG with AMG,
18 !c                 and PCG with ParaSails
19 !c
20 !c
21 !c                 Notes: for PCG, GMRES and BiCGStab, precond_id means:
22 !c                        0 - do not set up a preconditioner
23 !c                        1 - set up a ds preconditioner
24 !c                        2 - set up an amg preconditioner
25 !c                        3 - set up a pilut preconditioner
26 !c                        4 - set up a ParaSails preconditioner
27
28       program ex5f
29
30
31       implicit none
32
33       include 'mpif.h'
34
35
36
37       integer    MAX_LOCAL_SIZE
38       integer    HYPRE_PARCSR
39
40       parameter  (MAX_LOCAL_SIZE=123000)
41
42 !c     the following is from HYPRE.c
43       parameter  (HYPRE_PARCSR=5555)
44
45       integer    ierr
46       integer    num_procs, myid
47       integer    local_size, extra
48       integer    n, solver_id, print_solution, ng
49       integer    nnz, ilower, iupper, i
50       integer    precond_id;
51       double precision h, h2
52       double precision rhs_values(MAX_LOCAL_SIZE)
53       double precision x_values(MAX_LOCAL_SIZE)
54       integer    rows(MAX_LOCAL_SIZE)
```

```
55       integer    cols(5)
56       double precision values(5)
57       integer    num_iterations
58       double precision final_res_norm, tol
59
60       integer    mpi_comm
61
62       integer*8  parcsr_A
63       integer*8  A
64       integer*8  b
65       integer*8  x
66       integer*8  par_b
67       integer*8  par_x
68       integer*8  solver
69       integer*8  precond
70
71 !c-----------------------------------------------------------------------
72 !c     Initialize MPI
73 !c-----------------------------------------------------------------------
74
75       call MPI_INIT(ierr)
76       call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
77       call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
78       mpi_comm = MPI_COMM_WORLD
79 !c   Default problem parameters
80       n = 33
81       solver_id = 0
82       print_solution  = 1
83       tol = 1.0d-7
84
85 !c   The input section not implemented yet.
86
87 !c   Preliminaries: want at least one processor per row
88       if ( n*n .lt. num_procs ) then
89          n = int(sqrt(real(num_procs))) + 1
90       endif
91 !c     ng = global no. rows, h = mesh size
92       ng = n*n
93       h = 1.0d0/(n+1)
94       h2 = h*h
95
96 !c     Each processor knows only of its own rows - the range is denoted by
   ↪  ilower
97 !c     and upper.  Here we partition the rows. We account for the fact that
98 !c     N may not divide evenly by the number of processors.
99       local_size = ng/num_procs
100      extra = ng - local_size*num_procs
101
102      ilower = local_size*myid
```

```
      ilower = ilower + min(myid, extra)

      iupper = local_size*(myid+1)
      iupper = iupper + min(myid+1, extra)
      iupper = iupper - 1

!c     How many rows do I have?
      local_size = iupper - ilower + 1

!c     Create the matrix.
!c     Note that this is a square matrix, so we indicate the row partition
!c     size twice (since number of rows = number of cols)
      call HYPRE_IJMatrixCreate(mpi_comm, ilower,&
     iupper, ilower, iupper, A, ierr)


!c     Choose a parallel csr format storage (see the User's Manual)
      call HYPRE_IJMatrixSetObjectType(A, HYPRE_PARCSR, ierr)

!c     Initialize before setting coefficients
      call HYPRE_IJMatrixInitialize(A, ierr)


!c     Now go through my local rows and set the matrix entries.
!c     Each row has at most 5 entries. For example, if n=3:
!c
!c      A = [M -I 0; -I M -I; 0 -I M]
!c      M = [4 -1 0; -1 4 -1; 0 -1 4]
!c
!c     Note that here we are setting one row at a time, though
!c     one could set all the rows together (see the User's Manual).

      do i = ilower, iupper
         nnz = 1


!c        The left identity block:position i-n
         if ( (i-n) .ge. 0 ) then
            cols(nnz) = i-n
            values(nnz) = -1.0d0
            nnz = nnz + 1
         endif

!c        The left -1: position i-1
         if ( mod(i,n).ne.0 ) then
            cols(nnz) = i-1
            values(nnz) = -1.0d0
            nnz = nnz + 1
         endif
```

```
152
153 !c         Set the diagonal: position i
154           cols(nnz) = i
155           values(nnz) = 4.0d0
156           nnz = nnz + 1
157
158 !c         The right -1: position i+1
159           if ( mod((i+1),n) .ne. 0 ) then
160              cols(nnz) = i+1
161              values(nnz) = -1.0d0
162              nnz = nnz + 1
163           endif
164
165 !c         The right identity block:position i+n
166           if ( (i+n) .lt. ng ) then
167              cols(nnz) = i+n
168              values(nnz) = -1.0d0
169              nnz = nnz + 1
170           endif
171
172 !c         Set the values for row i
173           call HYPRE_IJMatrixSetValues(A, 1, nnz-1, i, cols, values, ierr)
174
175        enddo
176
177
178 !c      Assemble after setting the coefficients
179        call HYPRE_IJMatrixAssemble(A, ierr)
180
181 !c      Get parcsr matrix object
182        call HYPRE_IJMatrixGetObject(A, parcsr_A, ierr)
183
184
185 !c      Create the rhs and solution
186        call HYPRE_IJVectorCreate(mpi_comm,ilower, iupper, b, ierr)
187        call HYPRE_IJVectorSetObjectType(b, HYPRE_PARCSR, ierr)
188        call HYPRE_IJVectorInitialize(b, ierr)
189
190        call HYPRE_IJVectorCreate(mpi_comm, ilower, iupper, x, ierr)
191        call HYPRE_IJVectorSetObjectType(x, HYPRE_PARCSR, ierr)
192        call HYPRE_IJVectorInitialize(x, ierr)
193
194
195 !c      Set the rhs values to h^2 and the solution to zero
196        do i = 1, local_size
197           rhs_values(i) = h2
198           x_values(i) = 0.0
199           rows(i) = ilower + i -1
200        enddo
```

```
201       call HYPRE_IJVectorSetValues(b, local_size, rows, rhs_values, ierr)
202       call HYPRE_IJVectorSetValues(x, local_size, rows, x_values, ierr)
203
204
205       call HYPRE_IJVectorAssemble(b, ierr)
206       call HYPRE_IJVectorAssemble(x, ierr)
207
208 !c get the x and b objects
209
210       call HYPRE_IJVectorGetObject(b, par_b, ierr)
211       call HYPRE_IJVectorGetObject(x, par_x, ierr)
212
213 !c     Choose a solver and solve the system
214
215 !c     AMG
216       if ( solver_id .eq. 0 ) then
217
218 !c        Create solver
219          call HYPRE_BoomerAMGCreate(solver, ierr)
220
221
222 !c        Set some parameters (See Reference Manual for more parameters)
223
224 !c        print solve info + parameters
225          call HYPRE_BoomerAMGSetPrintLevel(solver, 3, ierr)
226 !c        old defaults, Falgout coarsening, mod. class. interpolation
227          call HYPRE_BoomerAMGSetOldDefault(solver, ierr)
228 !c        G-S/Jacobi hybrid relaxation
229          call HYPRE_BoomerAMGSetRelaxType(solver, 3, ierr)
230 !c        C/F relaxation
231          call HYPRE_BoomerAMGSetRelaxOrder(solver, 1, ierr)
232 !c        Sweeeps on each level
233          call HYPRE_BoomerAMGSetNumSweeps(solver, 1, ierr)
234 !c          maximum number of levels
235          call HYPRE_BoomerAMGSetMaxLevels(solver, 20, ierr)
236 !c        conv. tolerance
237          call HYPRE_BoomerAMGSetTol(solver, 1.0d-7, ierr)
238
239 !c        Now setup and solve!
240          call HYPRE_BoomerAMGSetup(solver, parcsr_A, par_b, par_x, ierr)
241          call HYPRE_BoomerAMGSolve(solver, parcsr_A, par_b, par_x, ierr)
242
243
244 !c        Run info - needed logging turned on
245          call HYPRE_BoomerAMGGetNumIterations(solver, num_iterations,ierr)
246          call HYPRE_BoomerAMGGetFinalReltvRes(solver, final_res_norm,ierr)
247
248
249          if ( myid .eq. 0 ) then
```

```
250             print *
251             print '(A,I2)', " Iterations = ", num_iterations
252             print '(A,ES16.8)', &
253                  " Final Relative Residual Norm = ", final_res_norm
254             print *
255           endif
256
257 !c         Destroy solver
258           call HYPRE_BoomerAMGDestroy(solver, ierr)
259 !c      PCG (with DS)
260         elseif ( solver_id .eq. 50 ) then
261
262
263 !c         Create solver
264           call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
265
266 !c         Set some parameters (See Reference Manual for more parameters)
267           call HYPRE_ParCSRPCGSetMaxIter(solver, 1000, ierr)
268           call HYPRE_ParCSRPCGSetTol(solver, 1.0d-7, ierr)
269           call HYPRE_ParCSRPCGSetTwoNorm(solver, 1, ierr)
270           call HYPRE_ParCSRPCGSetPrintLevel(solver, 2, ierr)
271           call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
272
273 !c         set ds (diagonal scaling) as the pcg preconditioner
274           precond_id = 1
275           call HYPRE_ParCSRPCGSetPrecond(solver, precond_id, &
276               precond, ierr)
277
278
279
280 !c         Now setup and solve!
281           call HYPRE_ParCSRPCGSetup(solver, parcsr_A, par_b,par_x, ierr)
282           call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b,par_x, ierr)
283
284
285 !c         Run info - needed logging turned on
286
287          call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
288          call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm,ierr)
289
290         if ( myid .eq. 0 ) then
291             print *
292             print *, "Iterations = ", num_iterations
293             print *, "Final Relative Residual Norm = ", final_res_norm
294             print *
295           endif
296
297 !c        Destroy solver
298          call HYPRE_ParCSRPCGDestroy(solver, ierr)
```

```
299
300
301 !c     PCG with AMG preconditioner
302       elseif ( solver_id == 1 ) then
303
304 !c        Create solver
305          call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
306
307 !c        Set some parameters (See Reference Manual for more parameters)
308          call HYPRE_ParCSRPCGSetMaxIter(solver, 1000, ierr)
309          call HYPRE_ParCSRPCGSetTol(solver, 1.0d-7, ierr)
310          call HYPRE_ParCSRPCGSetTwoNorm(solver, 1, ierr)
311          call HYPRE_ParCSRPCGSetPrintLevel(solver, 2, ierr)
312          call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
313
314 !c        Now set up the AMG preconditioner and specify any parameters
315
316          call HYPRE_BoomerAMGCreate(precond, ierr)
317
318
319 !c        Set some parameters (See Reference Manual for more parameters)
320
321 !c        print less solver info since a preconditioner
322          call HYPRE_BoomerAMGSetPrintLevel(precond, 1, ierr);
323 !c        Falgout coarsening
324          call HYPRE_BoomerAMGSetCoarsenType(precond, 6, ierr)
325 !c        old defaults
326          call HYPRE_BoomerAMGSetOldDefault(precond, ierr)
327 !c        SYMMETRIC G-S/Jacobi hybrid relaxation
328          call HYPRE_BoomerAMGSetRelaxType(precond, 6, ierr)
329 !c        Sweeeps on each level
330          call HYPRE_BoomerAMGSetNumSweeps(precond, 1, ierr)
331 !c        conv. tolerance
332          call HYPRE_BoomerAMGSetTol(precond, 0.0d0, ierr)
333 !c        do only one iteration!
334          call HYPRE_BoomerAMGSetMaxIter(precond, 1, ierr)
335
336 !c        set amg as the pcg preconditioner
337          precond_id = 2
338          call HYPRE_ParCSRPCGSetPrecond(solver, precond_id,&
339              precond, ierr)
340
341
342 !c        Now setup and solve!
343          call HYPRE_ParCSRPCGSetup(solver, parcsr_A, par_b,&
344                                    par_x, ierr)
345          call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b,&
346                                    par_x, ierr)
347
```

```
348
349 !c         Run info - needed logging turned on
350
351          call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
352          call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm,ierr)
353
354         if ( myid .eq. 0 ) then
355              print *
356              print *, "Iterations = ", num_iterations
357              print *, "Final Relative Residual Norm = ", final_res_norm
358              print *
359           endif
360
361 !c        Destroy precond and solver
362
363          call HYPRE_BoomerAMGDestroy(precond, ierr)
364          call HYPRE_ParCSRPCGDestroy(solver, ierr)
365
366 !c      PCG with ParaSails
367        elseif ( solver_id .eq. 8 ) then
368
369 !c         Create solver
370           call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
371
372 !c         Set some parameters (See Reference Manual for more parameters)
373           call HYPRE_ParCSRPCGSetMaxIter(solver, 1000, ierr)
374           call HYPRE_ParCSRPCGSetTol(solver, 1.0d-7, ierr)
375           call HYPRE_ParCSRPCGSetTwoNorm(solver, 1, ierr)
376           call HYPRE_ParCSRPCGSetPrintLevel(solver, 2, ierr)
377           call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
378
379 !c         Now set up the Parasails preconditioner and specify any parameters
380           call HYPRE_ParaSailsCreate(MPI_COMM_WORLD, precond,ierr)
381           call HYPRE_ParaSailsSetParams(precond, 0.1d0, 1, ierr)
382           call HYPRE_ParaSailsSetFilter(precond, 0.05d0, ierr)
383           call HYPRE_ParaSailsSetSym(precond, 1)
384           call HYPRE_ParaSailsSetLogging(precond, 3, ierr)
385
386 !c         set parsails as the pcg preconditioner
387           precond_id = 4
388           call HYPRE_ParCSRPCGSetPrecond(solver, precond_id,precond, ierr)
389
390
391 !c         Now setup and solve!
392           call HYPRE_ParCSRPCGSetup(solver, parcsr_A, par_b,par_x, ierr)
393           call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b,par_x, ierr)
394
395
396 !c         Run info - needed logging turned on
```

```
397
398         call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
399         call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm, ierr)
400
401        if ( myid .eq. 0 ) then
402             print *
403             print *, "Iterations = ", num_iterations
404             print *, "Final Relative Residual Norm = ", final_res_norm
405             print *
406          endif
407
408 !c       Destroy precond and solver
409
410         call HYPRE_ParaSailsDestroy(precond, ierr)
411         call HYPRE_ParCSRPCGDestroy(solver, ierr)
412
413       else
414          if ( myid .eq. 0 ) then
415            print *,'Invalid solver id specified'
416            stop
417          endif
418       endif
419
420
421
422 !c     Print the solution
423       if ( print_solution .ne. 0 ) then
424          call HYPRE_IJVectorPrint(x, "ij.out.x", ierr)
425       endif
426
427 !c     Clean up
428
429       call HYPRE_IJMatrixDestroy(A, ierr)
430       call HYPRE_IJVectorDestroy(b, ierr)
431       call HYPRE_IJVectorDestroy(x, ierr)
432
433
434 !c     Finalize MPI
435       call MPI_Finalize(ierr)
436
437       stop
438       end
```

4. Application to LLG equation

If we consider Landau-Lifshitz-Gilbert equation without the damping term as a example as below,

$$m_t = -m \times \Delta m. \tag{10}$$

We use BDF2 scheme and obtain that

$$\frac{\frac{3}{2}m_h^{n+2} - 2m_h^{n+1} + \frac{1}{2}m_h^n}{\Delta t} = -(2m_h^{n+1} - m_h^n) \times \Delta_h m_h^{n+2} \tag{11}$$

We list the code as below in 3D case.

```
! A fortran95 program for G95
! By XCJ
program main
! A test problem for Backward Differentiation formula Method (BDF_damping_3Dim)
! solve the Laudau-Lifshitz eqn
!
! BDF with damping alpha=0.d0
!
! author : Changjian
! date: 2018/10/19, version 1
!
! Variable declarations
! some remiders: 1) Wunused 2) the small and capital character is the same
! here we need o Ax = b
! for block tridiagnol matrix A (The matirx is the sum of a diagonal matrix D and a antisymmetric matrix E1)
! big picture:
! A = [ B1 C  O ... O  O  O
!       C  D  C ... O  O  O
!       O  C  D ... O  O  O
!       ...     ...     ...
!       O  O  O ... C  D  C
!       O  O  O ... O  C  B2 ]
! note that : A is non-singular i.e. determinent(A) .NE. 0; i.e. rank(A)=n (full rank)

! B1 = [ E    Cy    O   ... O  O  O
!        Cy  E-Cy  Cy   ... O  O  O
!        O    Cy   E-Cy ... O  O  O
!       ...     ...     ...     ...
!       O  O  O ... Cy    E-Cy    Cy
!       O  O  O ... O      Cy     E ] ! here E w.r.t z-direction (1 component)
! the above matrix Cy is antisymmetric

! B2 = [ E    Cy    O   ... O  O  O
!        Cy  E-Cy  Cy   ... O  O  O
!        O    Cy   E-Cy ... O  O  O
!       ...     ...     ...     ...
!       O  O  O ... Cy    E-Cy    Cy
!       O  O  O ... O      Cy     E ] ! here E w.r.t z-direction (Nz component)

! C = [ Cz O   O ... O  O  O
!       O  Cz  O ... O  O  O
```

```
!       0  0  Cz ... 0  0  0
!       ...     ...     ...
!       0  0  0 ...  0  Cz  0
!       0  0  0 ...  0  0  Cz ]
! the above matrix Cz is antisymmetric

! D = [ E-Cz    Cy    0    ... 0  0  0
!        Cy  E-Cy-Cz  Cy   ... 0  0  0
!        0    Cy   E-Cy-Cz ... 0  0  0
!       ...     ...     ...     ...
!       0  0  0 ... Cy    E-Cy-Cz    Cy
!       0  0  0 ... 0      Cy     E-Cz ] ! here E w.r.t z-direction (k=2:Nz-1
↪  component)
  implicit none

include 'mpif.h'

      integer    MAX_LOCAL_SIZE
      integer    HYPRE_PARCSR

      parameter  (MAX_LOCAL_SIZE=8000000)

!c     the following is from HYPRE.c
      parameter  (HYPRE_PARCSR=5555)

      integer    ierr
      integer    num_procs, myid
      integer    local_size, extra
      integer    solver_id, print_solution, ng
      integer    nnz, ilower, iupper, i
      integer    precond_id
      double precision h, h2
      double precision rhs_values(MAX_LOCAL_SIZE)
      double precision x_values(MAX_LOCAL_SIZE)
      integer    rows(MAX_LOCAL_SIZE)
      integer, parameter :: Nx=40,Ny=40,Nz=40,Nt=64
      integer, parameter :: nnl=3*Nx*Ny*Nz

      integer    cols(nnl,15)
      integer   mat_counter(nnl,1)
      double precision values(nnl,15)

      integer    num_iterations
      double precision final_res_norm, tol

      integer    mpi_comm

      integer*8  parcsr_A
      integer*8  AA        ! means :  HYPRE_TYPE
```

```
      integer*8  bb         ! means :  HYPRE_TYPE
      integer*8  x_hat    ! means :  HYPRE_TYPE
      integer*8  par_b      ! means :  HYPRE_TYPE
      integer*8  par_x      ! means :  HYPRE_TYPE
      integer*8  solver     ! means :  HYPRE_TYPE
      integer*8  precond    ! means :  HYPRE_TYPE

!!integer, parameter :: nnl=2304, nnl=192
integer :: nonzero_row_index(15,1),nonzero_col_index(15,1),i_index(nnl,1)
double precision :: nonzero_value(15,1)
double precision :: rhs(nnl,1),g(nnl,1)

!*************************** the semi-implict for BDF***********

double precision, parameter :: T=1.d0,T0=0.d0,EPS = 0.d0
double precision, parameter :: alpha = 0.d0 ! the damping parameter
double precision :: dt=0.d0,dx=0.d0,dy=0.d0,dz=0.d0,e_rr=0.d0
double precision :: xleft = 0.d0,xright = 1.d0
double precision :: yleft = 0.d0,yright = 1.d0
double precision :: zleft = 0.d0,zright = 1.d0

!double precision A(3*Nx*Ny*Nz,3*Nx*Ny*Nz)
double precision Ax(Nx,Nx),Iy(Nx,Nx),Iz(Nx,Nx),one(Nx,Nx)
double precision b(3*Nx,Ny,Nz)
double precision b1(3*Nx*Ny*Nz,1)
!double precision B_1(3*Nx*Ny,3*Nx*Ny)
!double precision B_2(3*Nx*Ny,3*Nx*Ny)
!double precision D(3*Nx*Ny,3*Nx*Ny)
!double precision C(3*Nx*Ny,3*Nx*Ny)
!double precision Cy(3*Nx,3*Nx,Ny,Nz),Cz(3*Nx,3*Nx,Ny,Nz),EE(3*Nx,3*Nx,Ny,Nz)

double precision Cy_1(Nx,Nx)

double precision x(Nx,1),XX(Nx,1),X_prime(Nx,1),X_prime_2(Nx,1)
double precision y(Ny,1),YY(Ny,1),Y_prime(Ny,1),Y_prime_2(Ny,1)
double precision z(Nz,1),ZZ(Ny,1),Z_prime(Nz,1),Z_prime_2(Nz,1)
double precision phi(Nx,1),psi(Nx,1)

double precision  m0(3*Nx,Ny,Nz),m0_prime(3*Nx,Ny,Nz)
double precision  m1(3*Nx,Ny,Nz),m1_prime(3*Nx,Ny,Nz)
double precision  mt(3*Nx,Ny,Nz),mt_p(3*Nx,Ny,Nz),mt_exact(3*Nx,Ny,Nz)
double precision  m(3*Nx*Ny*Nz,1),mt_exact_row(3*Nx*Ny*Nz,1)

double precision norm_mt(Nx,1)

double precision e(Nx,1)

! for the loop
integer :: ii=1,kk=1,jj=1,ll=1,nsteps=2,counter,counter_sum=0
```

```
139 double precision :: t1=0.d0,t2=0.d0,tt=0.d0
140 !***************new idea ***********
141 double precision :: Cy(1:3*Nx,1:3*Nx), Cz(1:3*Nx,1:3*Nx), EE(1:3*Nx,1:3*Nx)
142 !*************************************************************
143
144 !initialization for multi-arrays
145 !A = 0.d0
146 Ax = 0.d0
147 Iy = 0.d0
148 Iz = 0.d0
149 one = 0.d0
150 b = 0.d0
151 b1 = 0.d0
152 !B_1 = 0.d0
153 !B_2 = 0.d0
154 !C = 0.d0
155 !D = 0.d0
156 Cy = 0.d0
157 Cz = 0.d0
158 EE = 0.d0
159 Cy_1 = 0.d0
160
161 x = 0.d0
162 XX = 0.d0
163 X_prime = 0.d0
164 X_prime_2 = 0.d0
165
166 y = 0.d0
167 YY = 0.d0
168 Y_prime = 0.d0
169 Y_prime_2 = 0.d0
170
171 z = 0.d0
172 ZZ = 0.d0
173 Z_prime = 0.d0
174 Z_prime_2 = 0.d0
175
176 phi = 0.d0
177 psi = 0.d0
178
179 m0 = 0.d0
180 m0_prime = 0.d0
181
182 m1 = 0.d0
183 m1_prime = 0.d0
184
185 mt = 0.d0
186 mt_p = 0.d0
187 m = 0.d0
```

```
mt_exact = 0.d0
mt_exact_row = 0.d0

norm_mt = 0.d0

e = 1.d0
! new
nonzero_row_index = 0
nonzero_col_index = 0
nonzero_value = 0.d0
! time step
t1=second()

dt = T/Nt
! 3d grid partition
dx = (xright-xleft)/Nx
dy = (yright-yleft)/Ny
dz = (zright-zleft)/Nz
x(1,1)=xleft+dx/2
y(1,1)=yleft+dy/2
z(1,1)=zleft+dz/2
do ii=1,Nx-1
  x(ii+1,1)=x(ii,1)+dx
enddo

do ii=1,Ny-1
  y(ii+1,1)=y(ii,1)+dy
enddo

do ii=1,Nz-1
  z(ii+1,1)=z(ii,1)+dz
enddo

XX = x**2*(e-x)**2        ! change variable
X_prime = 4*x**3-6*x**2+2*x        ! the derivative of X
X_prime_2 = 12*x**2-12*x+2*e ! the derivative of X_prime

YY = y**2*(e-y)**2        ! change variable
Y_prime = 4*y**3-6*y**2+2*y        ! the derivative of Y
Y_prime_2 = 12*y**2-12*y+2*e ! the derivative of Y_prime

ZZ = z**2*(e-z)**2        ! change variable
Z_prime = 4*z**3-6*z**2+2*z        ! the derivative of Z
Z_prime_2 = 12*z**2-12*z+2*e ! the derivative of Z_prime

! initial magnetization
tt = T0
do kk=1,Nz
    do jj=1,Ny
```

```
237           m0(1:Nx,jj,kk)=dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
238           m0(1+Nx:2*Nx,jj,kk)=dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
239           m0(1+2*Nx:3*Nx,jj,kk)=e(:,1)*dcos(tt)
240       enddo
241   enddo
242
243   m0_prime = m0
244   tt = T0+dt
245   do kk=1,Nz
246       do jj=1,Ny
247           m1(1:Nx,jj,kk)=dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
248           m1(1+Nx:2*Nx,jj,kk)=dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
249           m1(1+2*Nx:3*Nx,jj,kk)=e(:,1)*dcos(tt)
250       enddo
251   enddo
252
253   m1_prime = m1
254   m = reshape(m1,(/3*Nx*Ny*Nz,1/))
255
256   do ii=2,Nx
257   Ax(ii,ii)=-2.d0
258   Ax(ii,ii-1)=1.d0
259   Ax(ii-1,ii)=1.d0
260   enddo
261   Ax(1,1)=-1.d0
262   Ax(Nx,Nx)=-1.d0
263
264   Ax = dt*Ax/dx**2
265
266   do ii=1,Nx
267    one(ii,ii)=1.d0
268   enddo
269   Iy=dt/dy**2*one
270   Iz=dt/dz**2*one
271   ! construct the hypre and mpi environment
272
273   !c-----------------------------------------------------------------------
274   !c     Initialize MPI
275   !c-----------------------------------------------------------------------
276
277         call MPI_INIT(ierr)
278         call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
279         call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
280         mpi_comm = MPI_COMM_WORLD
281
282         !c   Default problem parameters
283   !      n = 7 ! here is a Question**
284         solver_id =11 ! here we use GMRES solver
285         print_solution  = 1
```

```
      tol = 1.0d-7
      ng = 3*Nx*Ny*Nz
!      write(*,*) 'precond_id is'
!      write(*,*) precond_id
      local_size = ng/num_procs
      extra = ng - local_size*num_procs

      ilower = local_size*myid
      ilower = ilower + min(myid, extra)

      iupper = local_size*(myid+1)
      iupper = iupper + min(myid+1, extra)
      iupper = iupper - 1
!c     How many rows do I have?
      local_size = iupper - ilower + 1
!      write(*,*) 'local_size is '
!      write(*,*) local_size
!counter = 0

do nsteps=2,Nt
!nsteps = 2

!counter = 0

!nsteps = 3
tt =  T0 + nsteps*dt

!open(10,file='BDF_out')
!kk=1
!jj=1
!t2 = second()
!write(*,*) 'first is:'
!write(*,*) t2-t1
!t1 = second()

do kk=1,Nz
    do jj=1,Ny
       !****** c_11
        e(:,1)=(2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk))**2+&
             (2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk))**2
        call diag(Nx,e,Cy_1)
        Cy(1:Nx,1:Nx) = -alpha*matmul(Cy_1,Iy)
        Cz(1:Nx,1:Nx) = -alpha*matmul(Cy_1,Iz)
        EE(1:Nx,1:Nx) = 3.d0/2.d0*one-alpha*matmul(Cy_1,Ax-Iy-Iz)
        !***** the c_12
!        write(*,*) '1 is'
!        write(*,11) Cy_1*Iy
!        write(*,*) '2 is'
!        write(*,11) matmul(Cy_1,Iy)
```

```
335         e(:,1)=2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk) &
336            -alpha*(2*m1(Nx+1:2*Nx,jj,kk) &
337              -m0(Nx+1:2*Nx,jj,kk))*(2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk))
338 !        write(*,*) e
339
340         call diag(Nx,e,Cy_1)
341 !        write(*,11) Cy_1
342         11 format(1x,4F6.2)
343         Cy(1:Nx,Nx+1:2*Nx)=-matmul(Cy_1,Iy) ! here Cy is antisymmeric
344         Cz(1:Nx,Nx+1:2*Nx)=-matmul(Cy_1,Iz) ! here Cz is antisymmeric
345         ! here EE is a matrix which is : D+E1; where D is diagonal and E1 is
            ↪  antisymmeric
346         EE(1:Nx,Nx+1:2*Nx)=-matmul(Cy_1,Ax-Iy-Iz) ! here is matrix multiply
347 !        write(*,11) EE(1:Nx,Nx+1:2*Nx)
348         !****** the c_13
349         e(:,1)=2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk) &
350           +alpha*(2*m1(2*Nx+1:3*Nx,jj,kk)- &
351           m0(2*Nx+1:3*Nx,jj,kk))*(2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk))
352         call diag(Nx,e,Cy_1)
353         Cy(1:Nx,2*Nx+1:3*Nx)=matmul(Cy_1,Iy)
354         Cz(1:Nx,2*Nx+1:3*Nx)=matmul(Cy_1,Iz)
355         EE(1:Nx,2*Nx+1:3*Nx)=matmul(Cy_1,Ax-Iy-Iz)
356         !******* the c_21
357         e(:,1)=2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk) &
358            +alpha*(2*m1(1:Nx,jj,kk)- &
359           m0(1:Nx,jj,kk))*(2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk))
360         call diag(Nx,e,Cy_1)
361         Cy(Nx+1:2*Nx,1:Nx)=matmul(Cy_1,Iy)
362         Cz(Nx+1:2*Nx,1:Nx)=matmul(Cy_1,Iz)
363         EE(Nx+1:2*Nx,1:Nx)=matmul(Cy_1,Ax-Iy-Iz)
364         !****** the c_22
365         e(:,1)=(2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk))**2 &
366           +(2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk))**2
367         call diag(Nx,e,Cy_1)
368         Cy(Nx+1:2*Nx,Nx+1:2*Nx)=-alpha*matmul(Cy_1,Iy)
369         Cz(Nx+1:2*Nx,Nx+1:2*Nx)=-alpha*matmul(Cy_1,Iz)
370         EE(Nx+1:2*Nx,Nx+1:2*Nx)=3.d0/2.d0*one-alpha*matmul(Cy_1,Ax-Iy-Iz)
371         !******* the c_23
372         e(:,1)=2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk)-alpha*(2*m1(2*Nx+1:3*Nx,jj,kk) &
373             -m0(2*Nx+1:3*Nx,jj,kk))*(2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk))
374         call diag(Nx,e,Cy_1)
375         Cy(Nx+1:2*Nx,2*Nx+1:3*Nx)=-matmul(Cy_1,Iy)
376         Cz(Nx+1:2*Nx,2*Nx+1:3*Nx)=-matmul(Cy_1,Iz)
377         EE(Nx+1:2*Nx,2*Nx+1:3*Nx)=-matmul(Cy_1,Ax-Iy-Iz)
378         !******* the c_31
379         e(:,1)=2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk) &
380            -alpha*(2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk)) &
381             *(2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk))
382         call diag(Nx,e,Cy_1)
```

```
383         Cy(2*Nx+1:3*Nx,1:Nx)=-matmul(Cy_1,Iy)
384         Cz(2*Nx+1:3*Nx,1:Nx)=-matmul(Cy_1,Iz)
385         EE(2*Nx+1:3*Nx,1:Nx)=-matmul(Cy_1,Ax-Iy-Iz)
386         !****** the c_32
387         e(:,1)=2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk)+alpha*(2*m1(Nx+1:2*Nx,jj,kk)- &
388           m0(Nx+1:2*Nx,jj,kk))*(2*m1(2*Nx+1:3*Nx,jj,kk)-m0(2*Nx+1:3*Nx,jj,kk))
389         call diag(Nx,e,Cy_1)
390         Cy(2*Nx+1:3*Nx,Nx+1:2*Nx)=matmul(Cy_1,Iy)
391         Cz(2*Nx+1:3*Nx,Nx+1:2*Nx)=matmul(Cy_1,Iz)
392         EE(2*Nx+1:3*Nx,Nx+1:2*Nx)=matmul(Cy_1,Ax-Iy-Iz)
393         !****** the c_33
394         e(:,1)=(2*m1(1:Nx,jj,kk)-m0(1:Nx,jj,kk))**2 &
395           +(2*m1(Nx+1:2*Nx,jj,kk)-m0(Nx+1:2*Nx,jj,kk))**2
396         call diag(Nx,e,Cy_1)
397         Cy(2*Nx+1:3*Nx,2*Nx+1:3*Nx)=-alpha*matmul(Cy_1,Iy)
398         Cz(2*Nx+1:3*Nx,2*Nx+1:3*Nx)=-alpha*matmul(Cy_1,Iz)
399         EE(2*Nx+1:3*Nx,2*Nx+1:3*Nx)=3.d0/2.d0*one-alpha*matmul(Cy_1,Ax-Iy-Iz)
400
401
402         do ii=1,3*Nx
403             counter = 0
404             do ll=1,3*Nx
405                 if ( dabs(Cy(ii,ll)) .GT. EPS ) then
406                     ! consider the matrix Cy
407                     if (jj.EQ.1) then
408                         nonzero_row_index(counter+1,1) = (kk-1)*3*Nx*Ny+ii
409                         nonzero_col_index(counter+1,1) =
                            ↪  (kk-1)*3*Nx*Ny+3*Nx+ll-1
410                         nonzero_value(counter+1,1) = Cy(ii,ll)
411 !                        write(10,*)
    ↪  (kk-1)*3*Nx*Ny+ii,(kk-1)*3*Nx*Ny+3*Nx+ll,Cy(ii,ll)
412                         counter = counter+1
413                         counter_sum = counter_sum+1
414                     elseif (jj.EQ.Ny) then
415                         nonzero_row_index(counter+1,1) =
    ↪  (kk-1)*3*Nx*Ny+3*Nx*(Ny-1)+ii
416                         nonzero_col_index(counter+1,1) =
                            ↪  (kk-1)*3*Nx*Ny+3*Nx*(Ny-2)+ll-1
417                         nonzero_value(counter+1,1) = Cy(ii,ll)
418 !                        write(10,*)
    ↪  (kk-1)*3*Nx*Ny+3*Nx*(Ny-1)+ii,(kk-1)*3*Nx*Ny+3*Nx*(Ny-2)+ll,Cy(ii,ll)
419                         counter = counter+1
420                         counter_sum = counter_sum+1
421                     else
422                         nonzero_row_index(counter+1,1) =
    ↪  (kk-1)*3*Nx*Ny+3*Nx*(jj-1)+ii
423                         nonzero_col_index(counter+1,1) =
                            ↪  (kk-1)*3*Nx*Ny+3*Nx*(jj-2)+ll-1
```

```
424                         nonzero_value(counter+1,1) = Cy(ii,ll)
425                         counter = counter+1
426                         counter_sum = counter_sum+1
427
428                         nonzero_row_index(counter+1,1) =
                            ↪  (kk-1)*3*Nx*Ny+3*Nx*(jj-1)+ii
429                         nonzero_col_index(counter+1,1) =
                            ↪  (kk-1)*3*Nx*Ny+3*Nx*jj+ll-1
430                         nonzero_value(counter+1,1) = Cy(ii,ll)
431                         counter = counter+1
432                         counter_sum = counter_sum+1
433 !                        write(10,*)
    ↪  (kk-1)*3*Nx*Ny+3*Nx*(jj-1)+ii,(kk-1)*3*Nx*Ny+3*Nx*(jj-2)+ll,Cy(ii,ll)
434 !                        write(10,*)
    ↪  (kk-1)*3*Nx*Ny+3*Nx*(jj-1)+ii,(kk-1)*3*Nx*Ny+3*Nx*jj+ll,Cy(ii,ll)
435 !!                        counter = counter+2
436                     endif
437                 endif
438
439                 if ( dabs(EE(ii,ll)) .GT. EPS ) then
440                     ! consider the matrix E in B1
441                     if ( (kk.EQ.1).AND.(jj.EQ.1) ) then
442                         nonzero_row_index(counter+1,1) = ii
443                         nonzero_col_index(counter+1,1) = ll-1
444                         nonzero_value(counter+1,1) = EE(ii,ll)
445 !                        write(10,*) ii,ll,EE(ii,ll)
446                         counter = counter+1
447                         counter_sum = counter_sum+1
448                     endif
449
450                     if ( (kk.EQ.1).AND.(jj.EQ.Ny) ) then
451                         nonzero_row_index(counter+1,1) = 3*Nx*(Ny-1)+ii
452                         nonzero_col_index(counter+1,1) = 3*Nx*(Ny-1)+ll-1
453                         nonzero_value(counter+1,1) = EE(ii,ll)
454 !                        write(10,*) 3*Nx*(Ny-1)+ii,3*Nx*(Ny-1)+ll,EE(ii,ll)
455                         counter = counter+1
456                         counter_sum = counter_sum+1
457                     endif
458                      ! consider the matrix E in B2
459                     if ( (kk.EQ.Nz).AND.(jj.EQ.1) ) then
460                         nonzero_row_index(counter+1,1) = 3*Nx*Ny*(Nz-1)+ii
461                         nonzero_col_index(counter+1,1) = 3*Nx*Ny*(Nz-1)+ll-1
462                         nonzero_value(counter+1,1) = EE(ii,ll)
463 !                        write(10,*)
    ↪  3*Nx*Ny*(Nz-1)+ii,3*Nx*Ny*(Nz-1)+ll,EE(ii,ll)
464                         counter = counter+1
465                         counter_sum = counter_sum+1
466                     endif
467
```

```
468                     if ( (kk.EQ.Nz).AND.(jj.EQ.Ny) ) then
469                         nonzero_row_index(counter+1,1) =
    ↪ 3*Nx*Ny*(Nz-1)+3*Nx*(Ny-1)+ii
470                         nonzero_col_index(counter+1,1) =
                            ↪ 3*Nx*Ny*(Nz-1)+3*Nx*(Ny-1)+ll-1
471                         nonzero_value(counter+1,1) = EE(ii,ll)
472 !                        write(10,*)
    ↪ 3*Nx*Ny*(Nz-1)+3*Nx*(Ny-1)+ii,3*Nx*Ny*(Nz-1)+3*Nx*(Ny-1)+ll,EE(ii,ll)
473                         counter = counter+1
474                         counter_sum = counter_sum+1
475                     endif
476                 endif
477
478                 if ( dabs(EE(ii,ll)-Cy(ii,ll)) .GT. EPS ) then
479                     ! consider the k element of B1 (jj=2,Ny-1)
480                     if ( (jj.NE. 1).AND. (jj.NE.Ny).AND.(kk.EQ.1) ) then
481                         nonzero_row_index(counter+1,1) = 3*(jj-1)*Nx+ii
482                         nonzero_col_index(counter+1,1) = 3*(jj-1)*Nx+ll-1
483                         nonzero_value(counter+1,1) = EE(ii,ll)-Cy(ii,ll)
484 !                        write(10,*) 3*(jj-1)*Nx+ii,3*(jj-1)*Nx+ll,
    ↪ EE(ii,ll)-Cy(ii,ll)
485                         counter = counter+1
486                         counter_sum = counter_sum+1
487                     endif
488                     ! consider the k element of B2 (jj=2,Ny-1)
489                     if ( (jj.NE. 1).AND. (jj.NE.Ny).AND.(kk.EQ.Nz) ) then
490                         nonzero_row_index(counter+1,1) =
    ↪ 3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
491                         nonzero_col_index(counter+1,1) =
                            ↪ 3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll-1
492                         nonzero_value(counter+1,1) = EE(ii,ll)-Cy(ii,ll)
493 !                        write(10,*)
    ↪ 3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll,
    ↪ EE(ii,ll)-Cy(ii,ll)
494                         counter = counter+1
495                         counter_sum = counter_sum+1
496                     endif
497                 endif
498                     ! for D
499                 if ( dabs(EE(ii,ll)-Cz(ii,ll)).GT.EPS ) then
500                     if ( (kk.NE.1).AND.(kk.NE.Nz).AND.(jj.EQ.1) ) then
501                         nonzero_row_index(counter+1,1) = 3*(kk-1)*Nx*Ny+ii
502                         nonzero_col_index(counter+1,1) = 3*(kk-1)*Nx*Ny+ll-1
503                         nonzero_value(counter+1,1) = EE(ii,ll)-Cz(ii,ll)
504 !                        write(10,*)
    ↪ 3*(kk-1)*Nx*Ny+ii,3*(kk-1)*Nx*Ny+ll,EE(ii,ll)-Cz(ii,ll)
505                         counter = counter+1
506                         counter_sum = counter_sum+1
```

```
507                     endif
508
509                     if ( (kk.NE.1).AND.(kk.NE.Nz).AND.(jj.EQ.Ny) ) then
510                         nonzero_row_index(counter+1,1) =
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
511                         nonzero_col_index(counter+1,1) =
                            ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll-1
512                         nonzero_value(counter+1,1) = EE(ii,ll)-Cz(ii,ll)
513 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll,EE(ii,ll)-Cz(ii,ll)
514                         counter = counter+1
515                         counter_sum = counter_sum+1
516                     endif
517                 endif
518
519                 if ( dabs(EE(ii,ll)-Cy(ii,ll)-Cz(ii,ll)).GT.EPS ) then
520                     if ( (kk.NE.1).AND.(kk.NE.Nz).AND.(jj.NE.1).AND.(jj.NE.Ny)
    ↪  ) then
521                         nonzero_row_index(counter+1,1) =
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
522                         nonzero_col_index(counter+1,1) =
                            ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll-1
523                         nonzero_value(counter+1,1) =
                            ↪  EE(ii,ll)-Cy(ii,ll)-Cz(ii,ll)
524 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ll, &
525 ! EE(ii,ll)-Cy(ii,ll)-Cz(ii,ll)
526                         counter = counter+1
527                         counter_sum = counter_sum+1
528                     endif
529                 endif
530                 ! for off-diagnal matrix Cz
531                 if ( dabs(Cz(ii,ll)).GT.EPS ) then
532                     ! for the 1st and last rows
533                     if ( (kk.EQ.1) ) then
534                         nonzero_row_index(counter+1,1) =
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
535                         nonzero_col_index(counter+1,1) =
                            ↪  3*kk*Nx*Ny+3*(jj-1)*Nx+ll-1
536                         nonzero_value(counter+1,1) = Cz(ii,ll)
537 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*kk*Nx*Ny+3*(jj-1)*Nx+ll,Cz(ii,ll)
538                         counter = counter+1
539                         counter_sum = counter_sum+1
540                     endif
541
542                     if ( (kk.EQ.Nz) ) then
```

```
543                         nonzero_row_index(counter+1,1) =
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
544                         nonzero_col_index(counter+1,1) =
                            ↪  3*(kk-2)*Nx*Ny+3*(jj-1)*Nx+ll-1
545                         nonzero_value(counter+1,1) = Cz(ii,ll)
546 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*(kk-2)*Nx*Ny+3*(jj-1)*Nx+ll,Cz(ii,ll)
547                         counter = counter+1
548                         counter_sum = counter_sum+1
549                     endif
550                     ! for k rows; (k=2,Nz)
551                     if ( (kk.NE.1).AND.(kk.NE.Nz) ) then
552                         nonzero_row_index(counter+1,1) =
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
553                         nonzero_col_index(counter+1,1) =
                            ↪  3*kk*Nx*Ny+3*(jj-1)*Nx+ll-1
554                         nonzero_value(counter+1,1) = Cz(ii,ll)
555                         counter = counter+1
556                         counter_sum = counter_sum+1
557
558                         nonzero_row_index(counter+1,1) =
                            ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii
559                         nonzero_col_index(counter+1,1) =
                            ↪  3*(kk-2)*Nx*Ny+3*(jj-1)*Nx+ll-1
560                         nonzero_value(counter+1,1) = Cz(ii,ll)
561                         counter = counter+1
562                         counter_sum = counter_sum+1
563 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*kk*Nx*Ny+3*(jj-1)*Nx+ll,Cz(ii,ll)
564 !                        write(10,*)
    ↪  3*(kk-1)*Nx*Ny+3*(jj-1)*Nx+ii,3*(kk-2)*Nx*Ny+3*(jj-1)*Nx+ll,Cz(ii,ll)
565 !!                        counter = counter+2
566                     endif
567                 endif
568
569             enddo
570             mat_counter((kk-1)*3*Nx*Ny+(jj-1)*3*Nx+ii,1) = counter
571             cols((kk-1)*3*Nx*Ny+(jj-1)*3*Nx+ii,1:counter) =
                ↪  nonzero_col_index(1:counter,1)
572             cols((kk-1)*3*Nx*Ny+(jj-1)*3*Nx+ii,counter+1:) = 0 ! the rest to
                ↪  avoid out of memory
573             values((kk-1)*3*Nx*Ny+(jj-1)*3*Nx+ii,1:counter) =
                ↪  nonzero_value(1:counter,1)
574         enddo
575
576
577     enddo
578 enddo
```

```
!t2 = second()
!write(*,*) 'circle is:'
!write(*,*) t2-t1
!t1 = second()

!write(*,*) cols
!write(*,*) 'nonzero_row_index'
!write(*,*) nonzero_row_index
!!!
!close(10)
!!!!write(*,*) 'precond_id is'
!!!!write(*,*) precond_id
!write(*,*) 'counter_sum is'
!write(*,*) counter_sum
e = 1.d0
! the rhs term
    do kk=1,Nz
        do jj=1,Ny
            phi(:,1) = (YY(jj,1)*ZZ(kk,1))**2*(X_prime(:,1))**2+&
               (XX(:,1)*ZZ(kk,1))**2*(Y_prime(jj,1))**2 &
            +(XX(:,1)*YY(jj,1))**2*(Z_prime(kk,1))**2

            psi(:,1) = YY(jj,1)*ZZ(kk,1)*X_prime_2(:,1)+XX(:,1)*ZZ(kk,1)*&
              Y_prime_2(jj,1)+XX(:,1)*YY(jj,1)*Z_prime_2(kk,1)

            b(1:Nx,jj,kk) =
            ↪  dcos(tt)*dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))+dsin(tt)*dcos(tt)* &
           (dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*phi(:,1)- &
           dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*psi(:,1)) &
            +alpha*(dcos(tt)*dcos(tt)*dsin(tt)*dcos(XX(:,1)*&
            YY(jj,1)*ZZ(kk,1))*phi(:,1)+&
            dsin(tt)*dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*psi(:,1))

            b(Nx+1:2*Nx,jj,kk)=dcos(tt)*dsin(XX(:,1)*&
              YY(jj,1)*ZZ(kk,1))-dsin(tt)*dcos(tt)* &
            (dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*phi(:,1)+dsin(XX(:,1)*&
              YY(jj,1)*ZZ(kk,1))*psi(:,1))+alpha*(dcos(tt)*dcos(tt)* &
              dsin(tt)*dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*&
              phi(:,1)-dsin(tt)*dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*psi(:,1))

            b(2*Nx+1:3*Nx,jj,kk)=-dsin(tt)*e(:,1)+dsin(tt)*&
              dsin(tt)*psi(:,1)-alpha*dsin(tt)*&
              dsin(tt)*dcos(tt)*phi(:,1)
        enddo
    enddo

b1 = 2.d0*reshape(m1_prime,(/3*Nx*Ny*Nz,1/))-1.d0/2.d0*&
   reshape(m0_prime,(/3*Nx*Ny*Nz,1/))+dt*reshape(b,(/3*Nx*Ny*Nz,1/));
!write(*,*) 'b1 is'
```

```
!write(*,*) b1(1:3,1)

rhs = b1

!t2 = second()
!write(*,*) 'b is:'
!write(*,*) t2-t1
!t1 = second()

! here, the subroutine always happen the bug w.r.t. Invalid communicator
!*****************************************need to check
call GMRESsolver(MAX_LOCAL_SIZE,HYPRE_PARCSR,local_size, extra,&
    ilower,iupper,mpi_comm,nnl,cols,values,mat_counter,rhs,print_solution,&
            solver_id,myid,g)

!t2 = second()
!write(*,*) 'AMGsolver is:'
!write(*,*) t2-t1
!t1 = second()
!call AMGsolver(MAX_LOCAL_SIZE,HYPRE_PARCSR,local_size, extra,&
!             ilower,iupper,mpi_comm,nonzero_row_index,nonzero_col_index,&
!             nonzero_value,counter,rhs,print_solution,solver_id,myid,g,nn,nnl)
!call AMGsolver(MAX_LOCAL_SIZE,HYPRE_PARCSR,local_size,&
!             ilower,iupper,mpi_comm,nonzero_row_index,nonzero_col_index,&
!             nonzero_value,counter,rhs,print_solution,solver_id,myid,g)
!*****************************************************
!write(*,*) 'g is'
!write(*,*) g(1:3,1)
mt = reshape(g,(/3*Nx,Ny,Nz/))
!write(*,*) 'mt is'
!write(*,*) mt
!write(*,*) 'counter is'
!write(*,*) counter
mt_p = mt

do kk=1,Nz
    do jj=1,Ny
    norm_mt(:,1)
     ↪  =dsqrt(mt(1:Nx,jj,kk)**2+mt(Nx+1:2*Nx,jj,kk)**2+mt(2*Nx+1:3*Nx,jj,kk)**2)
    mt_p(1:Nx,jj,kk) = mt(1:Nx,jj,kk)/norm_mt(:,1)
    mt_p(Nx+1:2*Nx,jj,kk) = mt(Nx+1:2*Nx,jj,kk)/norm_mt(:,1)
    mt_p(2*Nx+1:3*Nx,jj,kk) = mt(2*Nx+1:3*Nx,jj,kk)/norm_mt(:,1)
    enddo
enddo

m0_prime = m1_prime
m1_prime = mt ! without projection term
m0 = m1
m1 = mt_p
```

```
675 !write(*,*) 'counter is'
676 !write(*,*) counter
677 write(*,*) 'the iteration times :',nsteps
678 enddo
679
680 !write(*,*) 'nonzero_col_index is'
681 !write(*,*) nonzero_col_index
682
683 !write(*,*) 'counter is'
684 !write(*,*) counter
685 m = reshape(mt_p,(/3*Nx*Ny*Nz,1/))
686 !write(*,*) 'm is'
687 !write(*,*) m(1:4,1)
688
689 tt = T0 + Nt*dt
690 !tt = T0 + 2*dt
691 mt_exact = 0.d0
692 e = 1.d0
693 !write(*,*) 'e is:'
694 !write(*,*) e(:,1)
695 do kk=1,Nz
696     do jj=1,Ny
697         mt_exact(1:Nx,jj,kk)=dcos(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
698         mt_exact(1+Nx:2*Nx,jj,kk)=dsin(XX(:,1)*YY(jj,1)*ZZ(kk,1))*dsin(tt)
699         mt_exact(1+2*Nx:3*Nx,jj,kk)=e(:,1)*dcos(tt)
700     enddo
701 enddo
702 !write(*,*) 'mt_exact(1+2*Nx:3*Nx,1,1) is:'
703 !write(*,*) mt_exact(1+2*Nx:3*Nx,1,1)
704 mt_exact_row = reshape(mt_exact,(/3*Nx*Ny*Nz,1/))
705 !write(*,*) 'mt_exact_row is:'
706 !write(*,*) mt_exact_row
707
708 !err = max(abs(m-mt_exact_row)) ! the infty norm
709 e_rr=maxval(dabs(m-mt_exact_row))
710
711 t2=second()
712 write(*,*) 'the accuracy is:'
713 write(*,*) e_rr
714 write(*,*) 'the cpu time is:'
715 write(*,*) t2-t1
716 !~ test counter
717 !do nsteps=2,3
718 !counter = 0
719 !counter = counter+1
720 !write(*,*) 'counter is'
721 !write(*,*) counter
722 !enddo
723
```

```
!write(*,*) 'ierr is'
!write(*,*) ierr
call MPI_Finalize(ierr)

end program main
```

```
subroutine GMRESsolver(MAX_LOCAL_SIZE,HYPRE_PARCSR,local_size,&
↪  extra,ilower,iupper,mpi_comm,nnl,cols,values,mat_counter,rhs,&
   print_solution,solver_id,myid,g)

implicit none

 include 'mpif.h'

      integer    MAX_LOCAL_SIZE
      integer    HYPRE_PARCSR

!      parameter  (MAX_LOCAL_SIZE=123000)

!c     the following is from HYPRE.c
!      parameter  (HYPRE_PARCSR=5555)

      integer    ierr
      integer    num_procs, myid
      integer    local_size, extra
      integer    n, solver_id, print_solution
      integer    nnz, ilower, iupper, i
      integer    precond_id;
!      double precision h, h2
      double precision rhs_values(MAX_LOCAL_SIZE)
      double precision x_values(MAX_LOCAL_SIZE)
      integer    rows(MAX_LOCAL_SIZE)
      integer :: nnl
      integer    cols(nnl,15)
      double precision values(nnl,15)
      integer    num_iterations,mat_counter(nnl,1), i_index(iupper+1,1)
      double precision final_res_norm, tol,t1,t2

      integer    mpi_comm

      integer*8  parcsr_A
      integer*8  AA        ! means :  HYPRE_TYPE
      integer*8  bb        ! means :  HYPRE_TYPE
      integer*8  x_hat        ! means :  HYPRE_TYPE
      integer*8  par_b    ! means :  HYPRE_TYPE
      integer*8  par_x    ! means :  HYPRE_TYPE
      integer*8  solver   ! means :  HYPRE_TYPE
      integer*8  precond  ! means :  HYPRE_TYPE
```

```
42
43 !      double precision :: A_values(273,1)
44 !      integer :: A_index(273,2)
45 !      integer :: nn,nnl
46 !      integer :: nonzero_row_index(nn,1), nonzero_col_index(nn,1)
47 !      double precision :: nonzero_value(nn,1)
48
49       integer :: ii,jj,counter
50
51       double precision :: g(nnl,1)
52       double precision :: rhs(nnl,1)
53 !!!c-------------------------------------------------------------------------
54 !!!c     Initialize MPI
55 !!!c-------------------------------------------------------------------------
56 !!
57 !!       call MPI_INIT(ierr)
58 !!       call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
59 !!       call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
60 !!       mpi_comm = MPI_COMM_WORLD
61       call HYPRE_IJMatrixCreate(mpi_comm, ilower,&
62      iupper, ilower, iupper, AA, ierr)
63
64
65 !c     Choose a parallel csr format storage (see the User's Manual)
66       call HYPRE_IJMatrixSetObjectType(AA, HYPRE_PARCSR, ierr)
67
68 !c     Initialize before setting coefficients
69       call HYPRE_IJMatrixInitialize(AA, ierr)
70
71 !write(*,*) 'A_index, A_values are :'
72 !write(*,*) A_index(:,2)
73
74 !write(*,*) 'ilower, iupper are :'
75 !write(*,*) ilower, iupper
76     call HYPRE_IJMatrixSetValues(AA, 1, mat_counter(i+1,1), i, cols(i+1,:),
       ↪  values(i+1,:), ierr)
77 !c     Assemble after setting the coefficients
78       call HYPRE_IJMatrixAssemble(AA, ierr)
79
80 !c     Get parcsr matrix object
81       call HYPRE_IJMatrixGetObject(AA, parcsr_A, ierr)
82
83
84 !c     Create the rhs and solution
85       call HYPRE_IJVectorCreate(mpi_comm,ilower, iupper, bb, ierr)
86       call HYPRE_IJVectorSetObjectType(bb, HYPRE_PARCSR, ierr)
87       call HYPRE_IJVectorInitialize(bb, ierr)
88
89       call HYPRE_IJVectorCreate(mpi_comm, ilower, iupper, x_hat, ierr)
```

```
90       call HYPRE_IJVectorSetObjectType(x_hat, HYPRE_PARCSR, ierr)
91       call HYPRE_IJVectorInitialize(x_hat, ierr)
92 !c     Set the rhs values to h^2 and the solution to zero
93 !write(*,*) 'h2 is'
94 !write(*,*) h2
95       do i = 1, local_size
96          rows(i) = ilower + i -1
97          rhs_values(i) = rhs(rows(i)+1,1)
98          x_values(i) = 0.0
99       enddo
100 !t2 = second()
101 !write(*,*) 'AMGsolver 1 is:'
102 !write(*,*) t2-t1
103 !t1 = second()
104       call HYPRE_IJVectorSetValues(bb, local_size, rows, rhs_values, ierr)
105       call HYPRE_IJVectorSetValues(x_hat, local_size, rows, x_values, ierr)
106
107
108       call HYPRE_IJVectorAssemble(bb, ierr)
109       call HYPRE_IJVectorAssemble(x_hat, ierr)
110
111 !c get the x and b objects
112
113       call HYPRE_IJVectorGetObject(bb, par_b, ierr)
114       call HYPRE_IJVectorGetObject(x_hat, par_x, ierr)
115 !*****************************************************************
116           ! the first method : BoomerAMG
117           ! solver_id = 0
118 !********************* begin the BOOMERAMG solver *****************
119
120 ! here on preconditioner, only the BoomerAMG as the solver
121 ! note that only BoomerAMG is not convergent when (n>7)
122
123       if ( solver_id .eq. 0 ) then
124
125 !c        Create solver
126          call HYPRE_BoomerAMGCreate(solver, ierr)
127
128
129 !c        Set some parameters (See Reference Manual for more parameters)
130
131 !c        print solve info + parameters
132          call HYPRE_BoomerAMGSetPrintLevel(solver, 0, ierr)
133 !c        old defaults, Falgout coarsening, mod. class. interpolation
134 !!         call HYPRE_BoomerAMGSetOldDefault(solver, ierr)
135
136          call HYPRE_BoomerAMGSetCoarsenType( solver, 6, ierr)
137          call HYPRE_BoomerAMGSetInterpType( solver, 0, ierr)
138          call HYPRE_BoomerAMGSetPMaxElmts( solver, 2, ierr)
```

```
139
140 !c         G-S/Jacobi hybrid relaxation
141          call HYPRE_BoomerAMGSetRelaxType(solver, 6, ierr)
142 !c         C/F relaxation
143          call HYPRE_BoomerAMGSetRelaxOrder(solver, 1, ierr)
144 !c         Sweeeps on each level
145
146          call HYPRE_BoomerAMGSetNumSweeps(solver, 2, ierr)
147 !c          maximum number of levels
148          call HYPRE_BoomerAMGSetMaxLevels(solver, 40, ierr)
149 !c         conv. tolerance
150          call HYPRE_BoomerAMGSetTol(solver, 1.0d-11, ierr)
151
152 !          maximum iterations:
153          call HYPRE_BoomerAMGSetMaxIter(solver, 200, ierr)
154
155 !c         Now setup and solve!
156          call HYPRE_BoomerAMGSetup(solver, parcsr_A, par_b, par_x, ierr)
157          call HYPRE_BoomerAMGSolve(solver, parcsr_A, par_b, par_x, ierr)
158
159
160 !c         Run info - needed logging turned on
161          call HYPRE_BoomerAMGGetNumIterations(solver, num_iterations,ierr)
162          call HYPRE_BoomerAMGGetFinalReltvRes(solver, final_res_norm,ierr)
163
164
165          if ( myid .eq. 0 ) then
166             print *
167             print '(A,I2)', " Iterations = ", num_iterations
168             print '(A,ES16.8)', &
169                  " Final Relative Residual Norm = ", final_res_norm
170             print *
171          endif
172
173 !c         Destroy solver
174          call HYPRE_BoomerAMGDestroy(solver, ierr)
175 !********************** end the BOOMERAMG solver **************
176
177
178          ! the second method : PCG (with DS)
179          ! solver_id = 50
180 !********************** begin the PCG solver (with DS preconditioner) ******
181
182
183 !c     PCG (with DS)
184       elseif ( solver_id .eq. 50 ) then
185
186
187 !c         Create solver
```

```
188           call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
189
190 !c         Set some parameters (See Reference Manual for more parameters)
191           call HYPRE_ParCSRPCGSetMaxIter(solver, 1000, ierr)
192           call HYPRE_ParCSRPCGSetTol(solver, 1.0d-11, ierr)
193           call HYPRE_ParCSRPCGSetTwoNorm(solver, 1, ierr)
194           call HYPRE_ParCSRPCGSetPrintLevel(solver, 2, ierr)
195           call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
196
197 !c         set ds (diagonal scaling) as the pcg preconditioner
198           precond_id = 1
199           call HYPRE_ParCSRPCGSetPrecond(solver, precond_id, &
200               precond, ierr)
201
202
203
204 !c         Now setup and solve!
205           call HYPRE_ParCSRPCGSetup(solver, parcsr_A, par_b,par_x, ierr)
206           call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b,par_x, ierr)
207
208
209 !c         Run info - needed logging turned on
210
211          call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
212          call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm,ierr)
213
214        if ( myid .eq. 0 ) then
215             print *
216             print *, "Iterations = ", num_iterations
217             print *, "Final Relative Residual Norm = ", final_res_norm
218             print *
219           endif
220
221 !c        Destroy solver
222          call HYPRE_ParCSRPCGDestroy(solver, ierr)
223 !********************** end the PCG solver (with DS preconditioner) ***********
224
225
226           ! the third method : PCG with AMG preconditioner
227           ! solver_id = 1
228 !********************** begin the PCG solver (with AMG preconditioner) ******
229
230 !c      PCG with AMG preconditioner
231       elseif ( solver_id .eq. 1 ) then
232
233 !c         Create solver
234           call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
235
236 !c         Set some parameters (See Reference Manual for more parameters)
```

```
237          call HYPRE_ParCSRPCGSetMaxIter(solver, 10**3, ierr) ! need to change
238          call HYPRE_ParCSRPCGSetTol(solver, 1.0d-11, ierr)
239          call HYPRE_ParCSRPCGSetTwoNorm(solver, 2, ierr)
240          call HYPRE_ParCSRPCGSetPrintLevel(solver, 0, ierr)
241          call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
242 !c         Now setup and solve!
243          call HYPRE_ParCSRPCGSetup(solver,parcsr_A,par_b,par_x,ierr)
244          call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b, par_x, ierr)
245         call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
246         call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm,ierr)
247
248        if ( myid .eq. 0 ) then
249             print *
250             print *, "Iterations = ", num_iterations
251             print *, "Final Relative Residual Norm = ", final_res_norm
252             print *
253          endif
254
255 !c       Destroy precond and solver
256
257 !!!!!         call HYPRE_BoomerAMGDestroy(precond, ierr)
258         call HYPRE_ParCSRPCGDestroy(solver, ierr)
259 !********************** end the PCG solver (with AMG preconditioner) ******
260
261         ! the fourth method : PCG with ParaSails preconditioner
262         ! solver_id = 8
263 !********************** begin the PCG solver (with ParaSails preconditioner)
    ↪  ******
264
265
266 !c     PCG with ParaSails
267       elseif ( solver_id .eq. 8 ) then
268
269 !c         Create solver
270          call HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, solver, ierr)
271
272 !c         Set some parameters (See Reference Manual for more parameters)
273          call HYPRE_ParCSRPCGSetMaxIter(solver, 1000, ierr)
274          call HYPRE_ParCSRPCGSetTol(solver, 1.0d-7, ierr)
275          call HYPRE_ParCSRPCGSetTwoNorm(solver, 1, ierr)
276          call HYPRE_ParCSRPCGSetPrintLevel(solver, 2, ierr)
277          call HYPRE_ParCSRPCGSetLogging(solver, 1, ierr)
278
279 !c         Now set up the Parasails preconditioner and specify any parameters
280          call HYPRE_ParaSailsCreate(MPI_COMM_WORLD, precond,ierr)
281          call HYPRE_ParaSailsSetParams(precond, 1.d0, 1, ierr) ! here need to
          ↪  adjust (0.1d0 will be non-positive definite)
282          call HYPRE_ParaSailsSetFilter(precond, 5.d0, ierr) ! original: 0.05d0
283          call HYPRE_ParaSailsSetSym(precond, 1)
```

```
284           call HYPRE_ParaSailsSetLogging(precond, 3, ierr)
285
286 !c         set parsails as the pcg preconditioner
287           precond_id = 4
288           call HYPRE_ParCSRPCGSetPrecond(solver, precond_id,precond, ierr)
289
290 !c         Now setup and solve!
291           call HYPRE_ParCSRPCGSetup(solver, parcsr_A, par_b,par_x, ierr)
292           call HYPRE_ParCSRPCGSolve(solver, parcsr_A, par_b,par_x, ierr)
293
294
295 !c         Run info - needed logging turned on
296
297          call HYPRE_ParCSRPCGGetNumIterations(solver, num_iterations,ierr)
298          call HYPRE_ParCSRPCGGetFinalRelative(solver, final_res_norm, ierr)
299
300         if ( myid .eq. 0 ) then
301              print *
302              print *, "Iterations = ", num_iterations
303              print *, "Final Relative Residual Norm = ", final_res_norm
304              print *
305           endif
306
307 !c        Destroy precond and solver
308
309          call HYPRE_ParaSailsDestroy(precond, ierr)
310          call HYPRE_ParCSRPCGDestroy(solver, ierr)
311 !********************* end the PCG solver (with ParaSails preconditioner)
    ↪  ******
312
313 ! start the ParCSR GMRES Solver
314 !c     GMRES
315       elseif ( solver_id .eq. 11 ) then
316       !c        Create solver
317           call HYPRE_ParCSRGMRESCreate(MPI_COMM_WORLD, solver, ierr)
318
319 !c         Set some parameters (See Reference Manual for more parameters)
320           call HYPRE_ParCSRGMRESSetMaxIter(solver, 10**3, ierr)
321           call HYPRE_ParCSRGMRESSetTol(solver, 1.0d-11, ierr)
322 !          call HYPRE_ParCSRGMRESSetTwoNorm(solver, 1, ierr)
323           call HYPRE_ParCSRGMRESSetPrintLevel(solver, 0, ierr)
324           call HYPRE_ParCSRGMRESSetLogging(solver, 1, ierr)
325 !c         Now setup and solve!
326           call HYPRE_ParCSRGMRESSetup(solver, parcsr_A, par_b,par_x, ierr)
327           call HYPRE_ParCSRGMRESSolve(solver, parcsr_A, par_b,par_x, ierr)
328 !c        Destroy solver
329
330          call HYPRE_ParCSRGMRESDestroy(solver, ierr)
331
```

```
332
333 ! start ParCSR FlexGMRES Solver
334 !c     FlexGMRES GMRES
335       elseif ( solver_id .eq. 12 ) then
336          call HYPRE_ParCSRFlexGMRESCreate(MPI_COMM_WORLD, solver, ierr)
337
338 !c         Set some parameters (See Reference Manual for more parameters)
339          call HYPRE_ParCSRFlexGMRESSetMaxIter(solver, 1000, ierr)
340          call HYPRE_ParCSRFlexGMRESSetTol(solver, 1.0d-7, ierr)
341 !         call HYPRE_ParCSRGMRESSetTwoNorm(solver, 1, ierr)
342          call HYPRE_ParCSRFlexGMRESSetPrintLevel(solver, 2, ierr)
343          call HYPRE_ParCSRFlexGMRESSetLogging(solver, 1, ierr)
344 !c         Now setup and solve!
345          call HYPRE_ParCSRFlexGMRESSetup(solver, parcsr_A, par_b,par_x, ierr)
346          call HYPRE_ParCSRFlexGMRESSolve(solver, parcsr_A, par_b,par_x, ierr)
347
348          call HYPRE_ParCSRFlexGMRESDestroy(solver, ierr)
349 ! start ParCSR LGMRES Solver
350 !  LGMRES
351         elseif ( solver_id .eq. 13 ) then
352             call HYPRE_ParCSRLGMRESCreate(MPI_COMM_WORLD, solver, ierr)
353
354 !c          Set some parameters (See Reference Manual for more parameters)
355             call HYPRE_ParCSRLGMRESSetMaxIter(solver, 1000, ierr)
356             call HYPRE_ParCSRLGMRESSetTol(solver, 1.0d-7, ierr)
357 !           call HYPRE_ParCSRGMRESSetTwoNorm(solver, 1, ierr)
358             call HYPRE_ParCSRLGMRESSetPrintLevel(solver, 2, ierr)
359             call HYPRE_ParCSRLGMRESSetLogging(solver, 1, ierr)
360 !c          Now setup and solve!
361             call HYPRE_ParCSRLGMRESSetup(solver, parcsr_A, par_b,par_x, ierr)
362             call HYPRE_ParCSRLGMRESSolve(solver, parcsr_A, par_b,par_x, ierr)
363
364             call HYPRE_ParCSRLGMRESDestroy(solver, ierr)
365 ! start ParCSR BiCGSTAB Solver
366 !   BiCGSTAB
367         elseif ( solver_id .eq. 14 ) then
368             call HYPRE_ParCSRBiCGSTABCreate(MPI_COMM_WORLD, solver, ierr)
369
370 !c          Set some parameters (See Reference Manual for more parameters)
371             call HYPRE_ParCSRBiCGSTABSetMaxIter(solver, 1000, ierr)
372             call HYPRE_ParCSRBiCGSTABSetTol(solver, 1.0d-11, ierr)
373 !           call HYPRE_ParCSRGMRESSetTwoNorm(solver, 1, ierr)
374 !            call HYPRE_ParCSRBiCGSTABSetPrintLevel(solver, 2, ierr)
375 !            call HYPRE_ParCSRBiCGSTABSetLogging(solver, 1, ierr)
376 !c          Now setup and solve!
377             call HYPRE_ParCSRBiCGSTABSetup(solver, parcsr_A, par_b,par_x, ierr)
378             call HYPRE_ParCSRBiCGSTABSolve(solver, parcsr_A, par_b,par_x, ierr)
379
380             call HYPRE_ParCSRBiCGSTABDestroy(solver, ierr)
```

```
! start ParCSR Hybrid Solver
 ! Hybrid
        elseif ( solver_id .eq. 15 ) then
            call HYPRE_ParCSRHybridCreate(solver, ierr)
!c          Set some parameters (See Reference Manual for more parameters)
!            call HYPRE_ParCSRHybridSetMaxIter(solver, 1000, ierr)
            call HYPRE_ParCSRHybridSetTol(solver, 1.0d-7, ierr)
           call HYPRE_ParCSRHybridSetTwoNorm(solver, 1, ierr)
            call HYPRE_ParCSRHybridSetPrintLevel(solver, 2, ierr)
            call HYPRE_ParCSRHybridSetLogging(solver, 1, ierr)
!c          Now setup and solve!
            call HYPRE_ParCSRHybridSetup(solver, parcsr_A, par_b,par_x, ierr)
            call HYPRE_ParCSRHybridSolve(solver, parcsr_A, par_b,par_x, ierr)

            call HYPRE_ParCSRHybridDestroy(solver, ierr)
      endif

!c     Print the solution
      if ( print_solution .ne. 0 ) then
!         call HYPRE_IJVectorPrint(x, "ij", ierr)
!call HYPRE_IJVectorSetValues(b, local_size, rows, rhs_values, ierr)
          call HYPRE_IJVectorGetValues(x_hat, local_size, rows, g, ierr)
!         call HYPRE_IJVectorPrint(x, g, ierr)
      endif

!c     Clean up

      call HYPRE_IJMatrixDestroy(AA, ierr)
      call HYPRE_IJVectorDestroy(bb, ierr)
      call HYPRE_IJVectorDestroy(x_hat, ierr)
      return


end subroutine GMRESsolver
```

```
subroutine diag(Nx,e,A)
implicit none
integer :: Nx,ii
double precision e(Nx,1)
double precision A(Nx,Nx)
do ii=1,Nx
A(ii,ii)=e(ii,1)
!write(*,*),A(ii,ii)
!99 format(1X,1F2.1)
enddo
return
end
```