ANL-95/11 Rev 3.14

PETSc Users Manual

Revision 3.14

Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Lemont, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's **SciTech Connect** (http://www.osti.gov/scitech/)

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandra, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: **orders@ntis.gov**

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: **reports@osti.gov**

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

PETSc Users Manual

Release 3.14.2

The PETSc Development Team

Dec 03, 2020

CONTENTS

1 **Introduction to PETSc** **3**
- 1.1 About This Manual 3
- 1.2 Getting Started 4
  - 1.2.1 Suggested Reading 4
  - 1.2.2 Running PETSc Programs 6
  - 1.2.3 Writing PETSc Programs 7
  - 1.2.4 Simple PETSc Examples 8
  - 1.2.5 Profiling Programs 19
  - 1.2.6 Writing Application Codes with PETSc 21
  - 1.2.7 Citing PETSc 22
  - 1.2.8 Directory Structure 23

2 **Programming with PETSc** **25**
- 2.1 Vectors and Parallel Data 25
  - 2.1.1 Creating and Assembling Vectors 25
  - 2.1.2 Basic Vector Operations 28
  - 2.1.3 Indexing and Ordering 29
  - 2.1.4 Structured Grids Using Distributed Arrays 32
  - 2.1.5 Vectors Related to Unstructured Grids 38
- 2.2 Matrices 41
  - 2.2.1 Creating and Assembling Matrices 42
  - 2.2.2 Basic Matrix Operations 51
  - 2.2.3 Matrix-Free Matrices 52
  - 2.2.4 Other Matrix Operations 53
  - 2.2.5 Partitioning 55
- 2.3 KSP: Linear System Solvers 56
  - 2.3.1 Using KSP 57
  - 2.3.2 Solving Successive Linear Systems 58
  - 2.3.3 Krylov Methods 58
  - 2.3.4 Preconditioners 64
  - 2.3.5 Solving Block Matrices 76
  - 2.3.6 Solving Singular Systems 79
  - 2.3.7 Using External Linear Solvers 79
- 2.4 SNES: Nonlinear Solvers 81
  - 2.4.1 Basic SNES Usage 82
  - 2.4.2 The Nonlinear Solvers 90
  - 2.4.3 General Options 94
  - 2.4.4 Inexact Newton-like Methods 96
  - 2.4.5 Matrix-Free Methods 97
  - 2.4.6 Finite Difference Jacobian Approximations 112

2.4.7 Variational Inequalities . . . 114
2.4.8 Nonlinear Preconditioning . . . 114
2.5 TS: Scalable ODE and DAE Solvers . . . 115
2.5.1 Basic TS Options . . . 117
2.5.2 DAE Formulations . . . 119
2.5.3 Using Implicit-Explicit (IMEX) Methods . . . 120
2.5.4 GLEE methods . . . 122
2.5.5 Using fully implicit methods . . . 123
2.5.6 Using the Explicit Runge-Kutta timestepper with variable timesteps . . . 124
2.5.7 Special Cases . . . 124
2.5.8 Monitoring and visualizing solutions . . . 125
2.5.9 Error control via variable time-stepping . . . 125
2.5.10 Handling of discontinuities . . . 126
2.5.11 Using TChem from PETSc . . . 127
2.5.12 Using Sundials from PETSc . . . 127
2.6 Performing sensitivity analysis . . . 128
2.6.1 Using the discrete adjoint methods . . . 128
2.6.2 Checkpointing . . . 130
2.7 Solving Steady-State Problems with Pseudo-Timestepping . . . 131
2.8 High Level Support for Multigrid with KSPSetDM() and SNESSetDM() . . . 132
2.8.1 Adaptive Interpolation . . . 133
2.9 DMPlex: Unstructured Grids in PETSc . . . 135
2.9.1 Representing Unstructured Grids . . . 135
2.9.2 Data on Unstructured Grids . . . 137
2.9.3 Evaluating Residuals . . . 139
2.9.4 Networks . . . 140

3 Additional Information 145
3.1 PETSc for Fortran Users . . . 145
3.1.1 C vs. Fortran Interfaces . . . 145
3.1.2 Sample Fortran Programs . . . 148
3.2 Using MATLAB with PETSc . . . 161
3.2.1 Dumping Data for MATLAB . . . 161
3.2.2 Sending Data to an Interactive MATLAB Session . . . 162
3.2.3 Using the MATLAB Compute Engine . . . 163
3.3 Profiling . . . 164
3.3.1 Basic Profiling Information . . . 164
3.3.2 Profiling Application Codes . . . 169
3.3.3 Profiling Multiple Sections of Code . . . 170
3.3.4 Restricting Event Logging . . . 171
3.3.5 Interpreting `-log_info` Output: Informative Messages . . . 171
3.3.6 Time . . . 172
3.3.7 Saving Output to a File . . . 172
3.3.8 Accurate Profiling and Paging Overheads . . . 172
3.4 Hints for Performance Tuning . . . 173
3.4.1 Maximizing Memory Bandwidth . . . 173
3.4.2 Performance Pitfalls and Advice . . . 179
3.5 Other PETSc Features . . . 184
3.5.1 PETSc on a process subset . . . 184
3.5.2 Runtime Options . . . 184
3.5.3 Viewers: Looking at PETSc Objects . . . 186
3.5.4 Using SAWs with PETSc . . . 188
3.5.5 Debugging . . . 189
3.5.6 Error Handling . . . 189

3.5.7 Numbers . . . 191
3.5.8 Parallel Communication . . . 191
3.5.9 Graphics . . . 191
3.5.10 Emacs Users . . . 196
3.5.11 VS Code Users . . . 197
3.5.12 Vi and Vim Users . . . 197
3.5.13 Eclipse Users . . . 197
3.5.14 Qt Creator Users . . . 198
3.5.15 Visual Studio Users . . . 200
3.5.16 XCode Users (The Apple GUI Development System) . . . 200
3.6 Unimportant and Advanced Features of Matrices and Solvers . . . 201
3.6.1 Extracting Submatrices . . . 201
3.6.2 Matrix Factorization . . . 201
3.6.3 Unimportant Details of KSP . . . 203
3.6.4 Unimportant Details of PC . . . 204
3.7 Running PETSc Tests . . . 205
3.7.1 Quick start with the tests . . . 205
3.7.2 Understanding test output and more information . . . 205
3.8 Acknowledgments . . . 207

Bibliography **211**

Argonne National Laboratory

Mathematics and Computer Science Division

Prepared by

S. Balay [1], **S. Abhyankar** [2], **M. Adams** [3], **J. Brown** [1], **P. Brune** [1], **K. Buschelman** [1], **L. Dalcin** [4], **A. Dener** [1], **V. Eijkhout** [6], **W. Gropp** [1], **D. Karpeyev** [1], **D. Kaushik** [1], **M. Knepley** [1], **D. May** [7], **L. Curfman McInnes** [1], **R. Mills** [1], **T. Munson** [1], **K. Rupp** [1], **P. Sanan** [8], **B. Smith** [1], **S. Zampini** [4], **H. Zhang** [5], and **H. Zhang** [1]

[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]Electricity Infrastructure and Buildings Division, Pacific Northwest National Laboratory
[3]Computational Research, Lawrence Berkeley National Laboratory
[4]Extreme Computing Research Center, King Abdullah University of Science and Technology
[5]Computer Science Department, Illinois Institute of Technology
[6]Texas Advanced Computing Center, University of Texas at Austin
[7]Department of Earth Sciences, University of Oxford
[8]Institute of Geophysics, ETH Zurich

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

CHAPTER

ONE

INTRODUCTION TO PETSC

1.1 About This Manual

This manual describes the use of the Portable, Extensible Toolkit for Scientific Computation (PETSc) for the numerical solution of partial differential equations and related problems on high-performance computers. PETSc is a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on parallel (and serial) computers. PETSc uses the MPI standard for all distributed memory communication.

PETSc includes a large suite of parallel linear solvers, nonlinear solvers, and time integrators that may be used in application codes written in Fortran, C, C++, and Python (via petsc4py; see *Getting Started*). PETSc provides many of the mechanisms needed within parallel application codes, such as parallel matrix and vector assembly routines. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem. By using techniques of object-oriented programming, PETSc provides enormous flexibility for users.

PETSc is a sophisticated set of software tools; as such, for some users it initially has a much steeper learning curve than packages such as MATLAB or a simple subroutine library. In particular, for individuals without some computer science background, experience programming in C, C++, python, or Fortran and experience using a debugger such as `gdb` or `lldb`, it may require a significant amount of time to take full advantage of the features that enable efficient software use. However, the power of the PETSc design and the algorithms it incorporates may make the efficient implementation of many application codes simpler than "rolling them" yourself.

- For many tasks a package such as MATLAB is often the best tool; PETSc is not intended for the classes of problems for which effective MATLAB code can be written.
- There are several packages (listed on https://www.mcs.anl.gov/petsc), built on PETSc, that may satisfy your needs without requiring directly using PETSc. We recommend reviewing these packages functionality before starting to code directly with PETSc.
- PETSc should *not* be used to attempt to provide a "parallel linear solver" in an otherwise sequential code. Certainly all parts of a previously sequential code need not be parallelized but the matrix generation portion must be parallelized to expect any kind of reasonable performance. Do not expect to generate your matrix sequentially and then "use PETSc" to solve the linear system in parallel.

Since PETSc is under continued development, small changes in usage and calling sequences of routines will occur. PETSc has been supported for twenty-five years; see https://www.mcs.anl.gov/petsc/miscellaneous/mailing-lists.html for information on contacting support.

We welcome any reports of corrections for this document at `petsc-maint@mcs.anl.gov`.

Manual pages and example usage : https://www.mcs.anl.gov/petsc/documentation/

Installing PETSc : https://www.mcs.anl.gov/petsc/documentation/installation.html

Tutorials : https://www.mcs.anl.gov/petsc/documentation/tutorials/index.html

1.2 Getting Started

PETSc consists of a collection of classes, which are discussed in detail in later parts of the manual (*Programming with PETSc* and *Additional Information*). The important PETSc classes include

- index sets (`IS`), including permutations, for indexing into vectors, renumbering, etc;
- vectors (`Vec`);
- matrices (`Mat`) (generally sparse);
- over thirty Krylov subspace methods (`KSP`);
- dozens of preconditioners, including multigrid, block solvers, and sparse direct solvers (`PC`);
- nonlinear solvers (`SNES`);
- timesteppers for solving time-dependent (nonlinear) PDEs, including support for differential algebraic equations, and the computation of adjoints (sensitivities/gradients of the solutions) (`TS`);
- managing interactions between mesh data structures and vectors, matrices, and solvers (`DM`);
- scalable optimization algorithms (`Tao`).

Each class consist of an abstract interface (simply a set of calling sequences; an abstract base class in C++) and an implementation for each algorithm and data structure. Thus, PETSc provides clean and effective codes for the various phases of solving PDEs, with a uniform approach for each type of problem. This design enables easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping.

The classes enable easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility, and also separates the issues of parallelism from the choice of algorithms. The PETSc infrastructure creates a foundation for building large-scale applications.

It is useful to consider the interrelationships among different pieces of PETSc. *Numerical Libraries in PETSc* is a diagram of some of these pieces. The figure illustrates the library's hierarchical organization, which enables users to employ the solvers that are most appropriate for a particular problem.

1.2.1 Suggested Reading

The manual is divided into three parts:

- *Introduction to PETSc*
- *Programming with PETSc*
- *Additional Information*

Introduction to PETSc describes the basic procedure for using the PETSc library and presents two simple examples of solving linear systems with PETSc. This section conveys the typical style used throughout the library and enables the application programmer to begin using the software immediately.

Programming with PETSc explains in detail the use of the various PETSc libraries, such as vectors, matrices, index sets, linear and nonlinear solvers, and graphics. *Additional Information* describes a variety of useful information, including profiling, the options database, viewers, error handling, and some details of PETSc design.

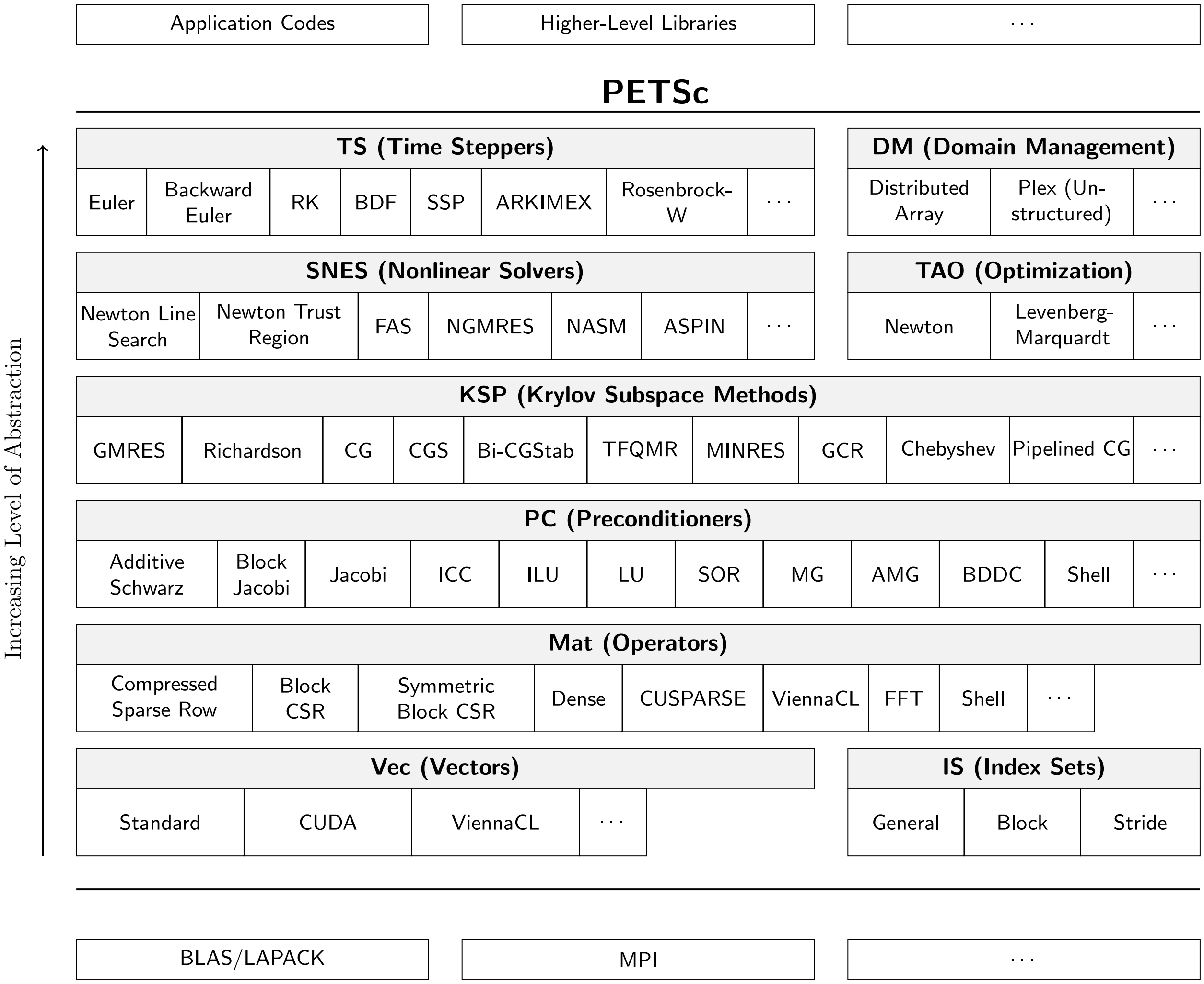

Application Codes
Higher-Level Libraries
· · ·
PETSc
TS (Time Steppers)
Euler
Backward Euler
RK
BDF
SSP
ARKIMEX
Rosenbrock-W
· · ·
DM (Domain Management)
Distributed Array
Plex (Unstructured)
· · ·
SNES (Nonlinear Solvers)
Newton Line Search
Newton Trust Region
FAS
NGMRES
NASM
ASPIN
· · ·
TAO (Optimization)
Newton
Levenberg-Marquardt
· · ·
KSP (Krylov Subspace Methods)
GMRES
Richardson
CG
CGS
Bi-CGStab
TFQMR
MINRES
GCR
Chebyshev
Pipelined CG
· · ·
PC (Preconditioners)
Additive Schwarz
Block Jacobi
Jacobi
ICC
ILU
LU
SOR
MG
AMG
BDDC
Shell
· · ·
Mat (Operators)
Compressed Sparse Row
Block CSR
Symmetric Block CSR
Dense
CUSPARSE
ViennaCL
FFT
Shell
· · ·
Vec (Vectors)
Standard
CUDA
ViennaCL
· · ·
IS (Index Sets)
General
Block
Stride
Increasing Level of Abstraction
BLAS/LAPACK
MPI
· · ·

Fig. 1.1: Numerical Libraries in PETSc

PETSc has evolved to become quite a comprehensive package, and therefore this manual can be rather intimidating for new users. Bear in mind that PETSc can be used efficiently before one understands all of the material presented here. Furthermore, the definitive reference for any PETSc function is always the online manual page. Manual pages for all PETSc functions can be accessed at www.mcs.anl.gov/petsc/documentation. The manual pages provide hyperlinked indices (organized by both concept and routine name) to the tutorial examples and enable easy movement among related topics.

Visual Studio Code , Eclipse, Emacs, and Vim users may find their development environment's options for searching in the source code (for example, `etags ctags` for Emacs and Vim) are extremely useful for exploring the PETSc source code. Details of these feature are provided in *Emacs Users*.

The complete PETSc distribution, manual pages, and additional information are available via the PETSc home page. The PETSc home page also contains details regarding installation, new features and changes in recent versions of PETSc, machines that we currently support, and a frequently asked questions (FAQ) list.

Note to Fortran Programmers: In most of the manual, the examples and calling sequences are given for the C/C++ family of programming languages. However, Fortran programmers can use all of the functionality of PETSc from Fortran, with only minor differences in the user interface. *PETSc for Fortran Users* provides a discussion of the differences between using PETSc from Fortran and C, as well as several complete Fortran examples.

Note to Python Programmers: To program with PETSc in Python you need to install the PETSc4py package developed by Lisandro Dalcin. This can be done by configuring PETSc with the option `--download-petsc4py`. See the PETSc installation guide for more details.

1.2.2 Running PETSc Programs

Before using PETSc, the user must first set the environmental variable `PETSC_DIR`, indicating the full path of the PETSc home directory. For example, under the UNIX bash shell a command of the form

```
export PETSC_DIR=$HOME/petsc
```

can be placed in the user's `.bashrc` or other startup file. In addition, the user may need to set the environment variable `PETSC_ARCH` to specify a particular configuration of the PETSc libraries. Note that `PETSC_ARCH` is just a name selected by the installer to refer to the libraries compiled for a particular set of compiler options and machine type. Using different values of `PETSC_ARCH` allows one to switch between several different sets (say debug and optimized) of libraries easily. To determine if you need to set `PETSC_ARCH`, look in the directory indicated by `PETSC_DIR`, if there are subdirectories beginning with `arch` then those subdirectories give the possible values for `PETSC_ARCH`.

All PETSc programs use the MPI (Message Passing Interface) standard for message-passing communication [For94]. Thus, to execute PETSc programs, users must know the procedure for beginning MPI jobs on their selected computer system(s). For instance, when using the MPICH implementation of MPI and many others, the following command initiates a program that uses eight processors:

```
mpiexec -n 8 ./petsc_program_name petsc_options
```

PETSc also comes with a script that automatically uses the correct `mpiexec` for your configuration.

```
${PETSC_DIR}/lib/petsc/bin/petscmpiexec -n 8 ./petsc_program_name petsc_options
```

All PETSc-compliant programs support the use of the `-help` option as well as the `-version` option.

Certain options are supported by all PETSc programs. We list a few particularly useful ones below; a complete list can be obtained by running any PETSc program with the option `-help`.

- `-log_view` - summarize the program's performance (see *Profiling*)

- `-fp_trap` - stop on floating-point exceptions; for example divide by zero
- `-malloc_dump` - enable memory tracing; dump list of unfreed memory at conclusion of the run, see *Detecting Memory Allocation Problems*,
- `-malloc_debug` - enable memory debugging (by default this is activated for the debugging version of PETSc), see *Detecting Memory Allocation Problems*,
- `-start_in_debugger [noxterm,gdb,lldb] [-display name]` - start all processes in debugger. See *Debugging*, for more information on debugging PETSc programs.
- `-on_error_attach_debugger [noxterm,gdb,lldb] [-display name]` - start debugger only on encountering an error
- `-info` - print a great deal of information about what the program is doing as it runs

1.2.3 Writing PETSc Programs

Most PETSc programs begin with a call to

```
ierr = PetscInitialize(int *argc,char ***argv,char *file,char *help);if (ierr) return␣
↪ierr;
```

which initializes PETSc and MPI. The arguments `argc` and `argv` are the command line arguments delivered in all C and C++ programs. The argument `file` optionally indicates an alternative name for the PETSc options file, `.petscrc`, which resides by default in the user's home directory. *Runtime Options* provides details regarding this file and the PETSc options database, which can be used for runtime customization. The final argument, `help`, is an optional character string that will be printed if the program is run with the `-help` option. In Fortran the initialization command has the form

```
call PetscInitialize(character(*) file,integer ierr)
```

`PetscInitialize()` automatically calls `MPI_Init()` if MPI has not been not previously initialized. In certain circumstances in which MPI needs to be initialized directly (or is initialized by some other library), the user can first call `MPI_Init()` (or have the other library do it), and then call `PetscInitialize()`. By default, `PetscInitialize()` sets the PETSc "world" communicator `PETSC_COMM_WORLD` to `MPI_COMM_WORLD`.

For those not familiar with MPI, a *communicator* is a way of indicating a collection of processes that will be involved together in a calculation or communication. Communicators have the variable type `MPI_Comm`. In most cases users can employ the communicator `PETSC_COMM_WORLD` to indicate all processes in a given run and `PETSC_COMM_SELF` to indicate a single process.

MPI provides routines for generating new communicators consisting of subsets of processors, though most users rarely need to use these. The book *Using MPI*, by Lusk, Gropp, and Skjellum [GLS94] provides an excellent introduction to the concepts in MPI. See also the MPI homepage. Note that PETSc users need not program much message passing directly with MPI, but they must be familiar with the basic concepts of message passing and distributed memory computing.

All PETSc routines return a `PetscErrorCode`, which is an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For the C/C++ interface, the error variable is the routine's return value, while for the Fortran version, each PETSc routine has as its final argument an integer error variable.

All PETSc programs should call `PetscFinalize()` as their final (or nearly final) statement, as given below in the C/C++ and Fortran formats, respectively:

```
ierr = PetscFinalize();
return ierr;
```

```
call PetscFinalize(ierr)
```

This routine handles options to be called at the conclusion of the program, and calls `MPI_Finalize()` if `PetscInitialize()` began MPI. If MPI was initiated externally from PETSc (by either the user or another software package), the user is responsible for calling `MPI_Finalize()`.

1.2.4 Simple PETSc Examples

To help the user start using PETSc immediately, we begin with a simple uniprocessor example that solves the one-dimensional Laplacian problem with finite differences. This sequential code, which can be found in `$PETSC_DIR/src/ksp/ksp/tutorials/ex1.c`, illustrates the solution of a linear system with `KSP`, the interface to the preconditioners, Krylov subspace methods, and direct linear solvers of PETSc. Following the code we highlight a few of the most important parts of this example.

Listing: `src/ksp/ksp/tutorials/ex1.c`

```

static char help[] = "Solves a tridiagonal linear system with KSP.\n\n";

/*T
   Concepts: KSP^solving a system of linear equations
   Processors: 1
T*/

/*
  Include "petscksp.h" so that we can use KSP solvers.  Note that this file
  automatically includes:
     petscsys.h    - base PETSc routines   petscvec.h - vectors
     petscmat.h    - matrices              petscpc.h  - preconditioners
     petscis.h     - index sets
     petscviewer.h - viewers

  Note:  The corresponding parallel example is ex23.c
*/
#include <petscksp.h>

int main(int argc,char **args)
{
  Vec            x, b, u;      /* approx solution, RHS, exact solution */
  Mat            A;            /* linear system matrix */
  KSP            ksp;          /* linear solver context */
  PC             pc;           /* preconditioner context */
  PetscReal      norm;         /* norm of solution error */
  PetscErrorCode ierr;
  PetscInt       i,n = 10,col[3],its;
  PetscMPIInt    size;
  PetscScalar    value[3];

  ierr = PetscInitialize(&argc,&args,(char*)0,help);if (ierr) return ierr;
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size != 1) SETERRQ(PETSC_COMM_WORLD,PETSC_ERR_WRONG_MPI_SIZE,"This is a␣
↪uniprocessor example only!");
```

(continues on next page)

(continued from previous page)

```
ierr = PetscOptionsGetInt(NULL,NULL,"-n",&n,NULL);CHKERRQ(ierr);


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       Compute the matrix and right-hand-side vector that define
       the linear system, Ax = b.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

/*
   Create vectors.  Note that we form 1 vector from scratch and
   then duplicate as needed.
*/
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

/*
   Create matrix.  When using MatCreate(), the matrix format can
   be specified at runtime.

   Performance tuning note:  For problems of substantial size,
   preallocation of matrix memory is crucial for attaining good
   performance. See the matrix chapter of the users manual for details.
*/
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
ierr = MatSetUp(A);CHKERRQ(ierr);

/*
   Assemble matrix
*/
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=1; i<n-1; i++) {
  col[0] = i-1; col[1] = i; col[2] = i+1;
  ierr   = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
}
i    = n - 1; col[0] = n - 2; col[1] = n - 1;
ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
i    = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

/*
   Set exact solution; then compute right-hand-side vector.
*/
ierr = VecSet(u,1.0);CHKERRQ(ierr);
ierr = MatMult(A,u,b);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
              Create the linear solver and set various options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

(continues on next page)

(continued from previous page)

```
  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

  /*
     Set operators. Here the matrix that defines the linear system
     also serves as the matrix that defines the preconditioner.
  */
  ierr = KSPSetOperators(ksp,A,A);CHKERRQ(ierr);

  /*
     Set linear solver defaults for this problem (optional).
     - By extracting the KSP and PC contexts from the KSP context,
       we can then directly call any KSP and PC routines to set
       various options.
     - The following four statements are optional; all of these
       parameters could alternatively be specified at runtime via
       KSPSetFromOptions();
  */
  ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
  ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
  ierr = KSPSetTolerances(ksp,1.e-5,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);
↪ CHKERRQ(ierr);

  /*
    Set runtime options, e.g.,
        -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
    These options will override those specified above as long as
    KSPSetFromOptions() is called _after_ any other customization
    routines.
  */
  ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      Solve the linear system
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

  /*
     View solver info; we could instead use the option -ksp_view to
     print this info to the screen at the conclusion of KSPSolve().
  */
  ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      Check the solution and clean up
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  ierr = VecAXPY(x,-1.0,u);CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %g, Iterations %D\n",
↪ (double)norm,its);CHKERRQ(ierr);

  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  ierr = VecDestroy(&x);CHKERRQ(ierr); ierr = VecDestroy(&u);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  ierr = VecDestroy(&b);CHKERRQ(ierr); ierr = MatDestroy(&A);CHKERRQ(ierr);
  ierr = KSPDestroy(&ksp);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.  This routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if certain runtime
         options are chosen (e.g., -log_view).
  */
  ierr = PetscFinalize();
  return ierr;
}
```

Include Files

The C/C++ include files for PETSc should be used via statements such as

```
#include <petscksp.h>
```

where `petscksp.h` is the include file for the linear solver library. Each PETSc program must specify an include file that corresponds to the highest level PETSc objects needed within the program; all of the required lower level include files are automatically included within the higher level files. For example, `petscksp.h` includes `petscmat.h` (matrices), `petscvec.h` (vectors), and `petscsys.h` (base PETSc file). The PETSc include files are located in the directory `${PETSC_DIR}/include`. See *Fortran Include Files* for a discussion of PETSc include files in Fortran programs.

The Options Database

As shown in *Simple PETSc Examples*, the user can input control data at run time using the options database. In this example the command `PetscOptionsGetInt(NULL,NULL,"-n",&n,&flg);` checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged. A complete description of the options database may be found in *Runtime Options*.

Vectors

One creates a new parallel or sequential vector, `x`, of global dimension `M` with the commands

```
VecCreate(MPI_Comm comm,Vec *x);
VecSetSizes(Vec x, PetscInt m, PetscInt M);
```

where `comm` denotes the MPI communicator and `m` is the optional local size which may be `PETSC_DECIDE`. The type of storage for the vector may be set with either calls to `VecSetType()` or `VecSetFromOptions()`. Additional vectors of the same type can be formed with

```
VecDuplicate(Vec old,Vec *new);
```

The commands

```
VecSet(Vec x,PetscScalar value);
VecSetValues(Vec x,PetscInt n,PetscInt *indices,PetscScalar *values,INSERT_VALUES);
```

respectively set all the components of a vector to a particular scalar value and assign a different value to each component. More detailed information about PETSc vectors, including their basic operations, scattering/gathering, index sets, and distributed arrays, is discussed in Chapter *Vectors and Parallel Data*.

Note the use of the PETSc variable type `PetscScalar` in this example. The `PetscScalar` is simply defined to be `double` in C/C++ (or correspondingly `double precision` in Fortran) for versions of PETSc that have *not* been compiled for use with complex numbers. The `PetscScalar` data type enables identical code to be used when the PETSc libraries have been compiled for use with complex numbers. *Numbers* discusses the use of complex numbers in PETSc programs.

Matrices

Usage of PETSc matrices and vectors is similar. The user can create a new parallel or sequential matrix, `A`, which has `M` global rows and `N` global columns, with the routines

```
MatCreate(MPI_Comm comm,Mat *A);
MatSetSizes(Mat A,PETSC_DECIDE,PETSC_DECIDE,PetscInt M,PetscInt N);
```

where the matrix format can be specified at runtime via the options database. The user could alternatively specify each processes' number of local rows and columns using `m` and `n`.

```
MatSetSizes(Mat A,PetscInt m,PetscInt n,PETSC_DETERMINE,PETSC_DETERMINE);
```

Generally one then sets the "type" of the matrix, with, for example,

```
MatSetType(A,MATAIJ);
```

This causes the matrix `A` to used the compressed sparse row storage format to store the matrix entries. See `MatType` for a list of all matrix types. Values can then be set with the command

```
MatSetValues(Mat A,PetscInt m,PetscInt *im,PetscInt n,PetscInt *in,PetscScalar␣
↪*values,INSERT_VALUES);
```

After all elements have been inserted into the matrix, it must be processed with the pair of commands

```
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Matrices discusses various matrix formats as well as the details of some basic matrix manipulation routines.

Linear Solvers

After creating the matrix and vectors that define a linear system, `Ax` = `b`, the user can then use `KSP` to solve the system with the following sequence of commands:

```
KSPCreate(MPI_Comm comm,KSP *ksp);
KSPSetOperators(KSP ksp,Mat Amat,Mat Pmat);
KSPSetFromOptions(KSP ksp);
KSPSolve(KSP ksp,Vec b,Vec x);
KSPDestroy(KSP ksp);
```

The user first creates the `KSP` context and sets the operators associated with the system (matrix that defines the linear system, `Amat` and matrix from which the preconditioner is constructed, `Pmat`). The user then sets various options for customized solution, solves the linear system, and finally destroys the `KSP` context. We emphasize the command `KSPSetFromOptions()`, which enables the user to customize the linear solution method at runtime by using the options database, which is discussed in *Runtime Options*. Through this database, the user not only can select an iterative method and preconditioner, but also can prescribe the convergence tolerance, set various monitoring routines, etc. (see, e.g., *Profiling Programs*).

KSP: Linear System Solvers describes in detail the `KSP` package, including the `PC` and `KSP` packages for preconditioners and Krylov subspace methods.

Nonlinear Solvers

Most PDE problems of interest are inherently nonlinear. PETSc provides an interface to tackle the nonlinear problems directly called `SNES`. *SNES: Nonlinear Solvers* describes the nonlinear solvers in detail. We recommend most PETSc users work directly with `SNES`, rather than using PETSc for the linear problem within a nonlinear solver.

Error Checking

All PETSc routines return an integer indicating whether an error has occurred during the call. The PETSc macro `CHKERRQ(ierr)` checks the value of `ierr` and calls the PETSc error handler upon error detection. `CHKERRQ(ierr)` should be used in all subroutines to enable a complete error traceback. Below, we indicate a traceback generated by error detection within a sample PETSc program. The error occurred on line 3618 of the file `${PETSC_DIR}/src/mat/impls/aij/seq/aij.c` and was caused by trying to allocate too large an array in memory. The routine was called in the program `ex3.c` on line 66. See *Error Checking* for details regarding error checking when using the PETSc Fortran interface.

```
$ cd $PETSC_DIR/src/ksp/ksp/tutorials
$ make ex3
$ mpiexec -n 1 ./ex3 -m 100000
[0]PETSC ERROR: --------------------- Error Message --------------------------------
[0]PETSC ERROR: Out of memory. This could be due to allocating
[0]PETSC ERROR: too large an object or bleeding by not properly
[0]PETSC ERROR: destroying unneeded objects.
[0]PETSC ERROR: Memory allocated 11282182704 Memory used by process 7075897344
[0]PETSC ERROR: Try running with -malloc_dump or -malloc_view for info.
[0]PETSC ERROR: Memory requested 18446744072169447424
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html for trouble␣
↪shooting.
[0]PETSC ERROR: Petsc Development GIT revision: v3.7.1-224-g9c9a9c5  GIT Date: 2016-
↪05-18 22:43:00 -0500
[0]PETSC ERROR: ./ex3 on a arch-darwin-double-debug named Patricks-MacBook-Pro-2.
↪local by patrick Mon Jun 27 18:04:03 2016
[0]PETSC ERROR: Configure options PETSC_DIR=/Users/patrick/petsc PETSC_ARCH=arch-
↪darwin-double-debug --download-mpich --download-f2cblaslapack --with-cc=clang --
↪with-cxx=clang++ --with-fc=gfortran --with-debugging=1 --with-precision=double --
↪with-scalar-type=real --with-viennacl=0 --download-c2html -download-sowing
[0]PETSC ERROR: #1 MatSeqAIJSetPreallocation_SeqAIJ() line 3618 in /Users/patrick/
↪petsc/src/mat/impls/aij/seq/aij.c
[0]PETSC ERROR: #2 PetscTrMallocDefault() line 188 in /Users/patrick/petsc/src/sys/
↪memory/mtr.c
[0]PETSC ERROR: #3 MatSeqAIJSetPreallocation_SeqAIJ() line 3618 in /Users/patrick/
↪petsc/src/mat/impls/aij/seq/aij.c
[0]PETSC ERROR: #4 MatSeqAIJSetPreallocation() line 3562 in /Users/patrick/petsc/src/
↪mat/impls/aij/seq/aij.c
```

(continues on next page)

(continued from previous page)

```
[0]PETSC ERROR: #5 main() line 66 in /Users/patrick/petsc/src/ksp/ksp/tutorials/ex3.c
[0]PETSC ERROR: PETSc Option Table entries:
[0]PETSC ERROR: -m 100000
[0]PETSC ERROR: ----------------End of Error Message ------- send entire error␣
↪message to petsc-maint@mcs.anl.gov----------
```

When running the debug version of the PETSc libraries, it does a great deal of checking for memory corruption (writing outside of array bounds etc). The macro `CHKMEMQ` can be called anywhere in the code to check the current status of the memory for corruption. By putting several (or many) of these macros into your code you can usually easily track down in what small segment of your code the corruption has occured. One can also use Valgrind to track down memory errors; see the FAQ.

Parallel Programming

Since PETSc uses the message-passing model for parallel programming and employs MPI for all interprocessor communication, the user is free to employ MPI routines as needed throughout an application code. However, by default the user is shielded from many of the details of message passing within PETSc, since these are hidden within parallel objects, such as vectors, matrices, and solvers. In addition, PETSc provides tools such as generalized vector scatters/gathers to assist in the management of parallel data.

Recall that the user must specify a communicator upon creation of any PETSc object (such as a vector, matrix, or solver) to indicate the processors over which the object is to be distributed. For example, as mentioned above, some commands for matrix, vector, and linear solver creation are:

```
MatCreate(MPI_Comm comm,Mat *A);
VecCreate(MPI_Comm comm,Vec *x);
KSPCreate(MPI_Comm comm,KSP *ksp);
```

The creation routines are collective over all processors in the communicator; thus, all processors in the communicator *must* call the creation routine. In addition, if a sequence of collective routines is being used, they *must* be called in the same order on each processor.

The next example, given below, illustrates the solution of a linear system in parallel. This code, corresponding to *KSP Tutorial ex2 <https://www.mcs.anl.gov/petsc/petsc-current/src/ksp/ksp/tutorials/ex2.c.html>*, handles the two-dimensional Laplacian discretized with finite differences, where the linear system is again solved with KSP. The code performs the same tasks as the sequential version within *Simple PETSc Examples*. Note that the user interface for initiating the program, creating vectors and matrices, and solving the linear system is *exactly* the same for the uniprocessor and multiprocessor examples. The primary difference between the examples in *Simple PETSc Examples* and here is that each processor forms only its local part of the matrix and vectors in the parallel case.

Listing: **`src/ksp/ksp/tutorials/ex2.c`**

```

static char help[] = "Solves a linear system in parallel with KSP.\n\
Input parameters include:\n\
  -view_exact_sol   : write exact solution vector to stdout\n\
  -m <mesh_x>       : number of mesh points in x-direction\n\
  -n <mesh_y>       : number of mesh points in y-direction\n\n";

/*T
   Concepts: KSP^basic parallel example;
   Concepts: KSP^Laplacian, 2d
```

(continues on next page)

(continued from previous page)

```
   Concepts: Laplacian, 2d
   Processors: n
T*/

/*
  Include "petscksp.h" so that we can use KSP solvers.
*/
#include <petscksp.h>

int main(int argc,char **args)
{
  Vec            x,b,u;    /* approx solution, RHS, exact solution */
  Mat            A;        /* linear system matrix */
  KSP            ksp;      /* linear solver context */
  PetscReal      norm;     /* norm of solution error */
  PetscInt       i,j,Ii,J,Istart,Iend,m = 8,n = 7,its;
  PetscErrorCode ierr;
  PetscBool      flg;
  PetscScalar    v;

  ierr = PetscInitialize(&argc,&args,(char*)0,help);if (ierr) return ierr;
  ierr = PetscOptionsGetInt(NULL,NULL,"-m",&m,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-n",&n,NULL);CHKERRQ(ierr);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         Compute the matrix and right-hand-side vector that define
         the linear system, Ax = b.
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Create parallel matrix, specifying only its global dimensions.
     When using MatCreate(), the matrix format can be specified at
     runtime. Also, the parallel partitioning of the matrix is
     determined by PETSc at runtime.

     Performance tuning note:  For problems of substantial size,
     preallocation of matrix memory is crucial for attaining good
     performance. See the matrix chapter of the users manual for details.
  */
  ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
  ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n);CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);
  ierr = MatMPIAIJSetPreallocation(A,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatSeqAIJSetPreallocation(A,5,NULL);CHKERRQ(ierr);
  ierr = MatSeqSBAIJSetPreallocation(A,1,5,NULL);CHKERRQ(ierr);
  ierr = MatMPISBAIJSetPreallocation(A,1,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatMPISELLSetPreallocation(A,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatSeqSELLSetPreallocation(A,5,NULL);CHKERRQ(ierr);

  /*
     Currently, all PETSc parallel matrix formats are partitioned by
     contiguous chunks of rows across the processors.  Determine which
     rows of the matrix are locally owned.
  */
  ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);

  /*
     Set matrix elements for the 2-D, five-point stencil in parallel.
```

(continues on next page)

(continued from previous page)

```
      - Each processor needs to insert only elements that it owns
        locally (but any non-local elements will be sent to the
        appropriate processor during matrix assembly).
      - Always specify global rows and columns of matrix entries.

     Note: this uses the less common natural ordering that orders first
     all the unknowns for x = h then for x = 2h etc; Hence you see J = Ii +- n
     instead of J = I +- m as you might expect. The more standard ordering
     would first do all variables for y = h, then y = 2h etc.

   */
  for (Ii=Istart; Ii<Iend; Ii++) {
    v = -1.0; i = Ii/n; j = Ii - i*n;
    if (i>0)   {J = Ii - n; ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
↪CHKERRQ(ierr);}
    if (i<m-1) {J = Ii + n; ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
↪CHKERRQ(ierr);}
    if (j>0)   {J = Ii - 1; ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
↪CHKERRQ(ierr);}
    if (j<n-1) {J = Ii + 1; ierr = MatSetValues(A,1,&Ii,1,&J,&v,ADD_VALUES);
↪CHKERRQ(ierr);}
    v = 4.0; ierr = MatSetValues(A,1,&Ii,1,&Ii,&v,ADD_VALUES);CHKERRQ(ierr);
  }

  /*
     Assemble matrix, using the 2-step process:
       MatAssemblyBegin(), MatAssemblyEnd()
     Computations can be done while messages are in transition
     by placing code between these two statements.
  */
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  /* A is symmetric. Set symmetric flag to enable ICC/Cholesky preconditioner */
  ierr = MatSetOption(A,MAT_SYMMETRIC,PETSC_TRUE);CHKERRQ(ierr);

  /*
     Create parallel vectors.
      - We form 1 vector from scratch and then duplicate as needed.
      - When using VecCreate(), VecSetSizes and VecSetFromOptions()
        in this example, we specify only the
        vector's global dimension; the parallel partitioning is determined
        at runtime.
      - When solving a linear system, the vectors and matrices MUST
        be partitioned accordingly.  PETSc automatically generates
        appropriately partitioned matrices and vectors when MatCreate()
        and VecCreate() are used with the same communicator.
      - The user can alternatively specify the local vector and matrix
        dimensions when more sophisticated partitioning is needed
        (replacing the PETSC_DECIDE argument in the VecSetSizes() statement
        below).
  */
  ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
  ierr = VecSetSizes(u,PETSC_DECIDE,m*n);CHKERRQ(ierr);
  ierr = VecSetFromOptions(u);CHKERRQ(ierr);
  ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  ierr = VecDuplicate(b,&x);CHKERRQ(ierr);

  /*
     Set exact solution; then compute right-hand-side vector.
     By default we use an exact solution of a vector with all
     elements of 1.0;
  */
  ierr = VecSet(u,1.0);CHKERRQ(ierr);
  ierr = MatMult(A,u,b);CHKERRQ(ierr);

  /*
     View the exact solution vector if desired
  */
  flg  = PETSC_FALSE;
  ierr = PetscOptionsGetBool(NULL,NULL,"-view_exact_sol",&flg,NULL);CHKERRQ(ierr);
  if (flg) {ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);}

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                Create the linear solver and set various options
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

  /*
     Set operators. Here the matrix that defines the linear system
     also serves as the preconditioning matrix.
  */
  ierr = KSPSetOperators(ksp,A,A);CHKERRQ(ierr);

  /*
     Set linear solver defaults for this problem (optional).
     - By extracting the KSP and PC contexts from the KSP context,
       we can then directly call any KSP and PC routines to set
       various options.
     - The following two statements are optional; all of these
       parameters could alternatively be specified at runtime via
       KSPSetFromOptions().  All of these defaults can be
       overridden at runtime, as indicated below.
  */
  ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n+1)),1.e-50,PETSC_DEFAULT,PETSC_DEFAULT);
↪CHKERRQ(ierr);

  /*
    Set runtime options, e.g.,
        -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
    These options will override those specified above as long as
    KSPSetFromOptions() is called _after_ any other customization
    routines.
  */
  ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      Solve the linear system
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      Check the solution and clean up
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  ierr = VecAXPY(x,-1.0,u);CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);

  /*
     Print convergence information.  PetscPrintf() produces a single
     print statement from all processes that share a communicator.
     An alternative is PetscFPrintf(), which prints to a file.
  */
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %g iterations %D\n",(double)norm,
↪its);CHKERRQ(ierr);

  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  ierr = KSPDestroy(&ksp);CHKERRQ(ierr);
  ierr = VecDestroy(&u);CHKERRQ(ierr);  ierr = VecDestroy(&x);CHKERRQ(ierr);
  ierr = VecDestroy(&b);CHKERRQ(ierr);  ierr = MatDestroy(&A);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.  This routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if certain runtime
         options are chosen (e.g., -log_view).
  */
  ierr = PetscFinalize();
  return ierr;
}
```

Compiling and Running Programs

The output below illustrates compiling and running a PETSc program using MPICH on an OS X laptop. Note that different machines will have compilation commands as determined by the configuration process. See *Writing Application Codes with PETSc* for a discussion about how to compile your PETSc programs. Users who are experiencing difficulties linking PETSc programs should refer to the FAQ on the PETSc website https://www.mcs.anl.gov/petsc or given in the file `$PETSC_DIR/docs/faq.html`.

```
$ cd $PETSC_DIR/src/ksp/ksp/tutorials
$ make ex2
/Users/patrick/petsc/arch-darwin-double-debug/bin/mpicc -o ex2.o -c -Wall -Wwrite-
↪strings -Wno-strict-aliasing -Wno-unknown-pragmas -Qunused-arguments -
↪fvisibility=hidden -g3   -I/Users/patrick/petsc/include -I/Users/patrick/petsc/arch-
↪darwin-double-debug/include -I/opt/X11/include -I/opt/local/include    `pwd`/ex2.c
/Users/patrick/petsc/arch-darwin-double-debug/bin/mpicc -Wl,-multiply_defined,
↪suppress -Wl,-multiply_defined -Wl,suppress -Wl,-commons,use_dylibs -Wl,-search_
↪paths_first -Wl,-multiply_defined,suppress -Wl,-multiply_defined -Wl,suppress -Wl,-
↪commons,use_dylibs -Wl,-search_paths_first    -Wall -Wwrite-strings -Wno-strict-
↪aliasing -Wno-unknown-pragmas -Qunused-arguments -fvisibility=hidden -g3  -o ex2␣
↪ex2.o  -Wl,-rpath,/Users/patrick/petsc/arch-darwin-double-debug/lib -L/Users/
↪patrick/petsc/arch-darwin-double-debug/lib  -lpetsc -Wl,-rpath,/Users/patrick/petsc/
↪arch-darwin-double-debug/lib -lf2clapack -lf2cblas -Wl,-rpath,/opt/X11/lib -L/opt/
↪X11/lib -lX11 -lssl -lcrypto -Wl,-rpath,/Applications/Xcode.app/Contents/Developer/
↪Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/7.0.2/lib/darwin -L/Applications/
↪Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/7.0.
↪2/lib/darwin -lmpifort -lgfortran -Wl,-rpath,/opt/local/lib/gcc5/gcc/x86_64-apple-
↪darwin14/5.3.0 -L/opt/local/lib/gcc5/gcc/x86_64-apple-darwin14/5.3.0 -Wl,-rpath,/
↪opt/local/lib/gcc5 -L/opt/local/lib/gcc5 -lgfortran -lgcc_ext.10.5 -lquadmath -lm -
↪lclang_rt.osx -lmpicxx -lc++ -Wl,-rpath,/Applications/Xcode.app/Contents/Developer/
↪Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/7.0.2/lib/darwin -L/
↪Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪bin/../lib/clang/7.0.2/lib/darwin -lclang_rt.osx -Wl,-rpath,/Users/patrick/petsc/
↪arch-darwin-double-debug/lib -L/Users/patrick/petsc/arch-darwin-double-debug/lib -
↪ldl -lmpi -lpmpi -lSystem -Wl,-rpath,/Applications/Xcode.app/Contents/Developer/
↪Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/7.0.2/lib/darwin -L/
↪Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪bin/../lib/clang/7.0.2/lib/darwin -lclang_rt.osx -ldl
/bin/rm -f ex2.o
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 1 ./ex2
Norm of error 0.000156044 iterations 6
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 2 ./ex2
Norm of error 0.000411674 iterations 7
```

1.2.5 Profiling Programs

The option `-log_view` activates printing of a performance summary, including times, floating point operation (flop) rates, and message-passing activity. *Profiling* provides details about profiling, including interpretation of the output data below. This particular example involves the solution of a linear system on one processor using GMRES and ILU. The low floating point operation (flop) rates in this example are due to the fact that the code solved a tiny system. We include this example merely to demonstrate the ease of extracting performance information.

```
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 1 ./ex1 -n 1000 -pc_type ilu -ksp_type␣
↪gmres -ksp_rtol 1.e-7 -log_view
...
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------------
↪----------------------------------
Event                Count      Time (sec)     Flops                             ---␣
↪Global ---  --- Stage ----  Total
                  Max Ratio   Max     Ratio   Max  Ratio  Mess   AvgLen  Reduct  %T
↪%F %M %L %R  %T %F %M %L %R Mflop/s
--------------------------------------------------------------------------------------
↪----------------------------------

--- Event Stage 0: Main Stage

VecMDot                1 1.0 3.2830e-06 1.0 2.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪5  0  0  0   0  5  0  0  0   609
VecNorm                3 1.0 4.4550e-06 1.0 6.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0␣
↪14  0  0  0   0 14  0  0  0  1346
VecScale               2 1.0 4.0110e-06 1.0 2.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪5  0  0  0   0  5  0  0  0   499
VecCopy                1 1.0 3.2280e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
VecSet                11 1.0 2.5537e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪0  0  0  0   2  0  0  0  0     0
VecAXPY                2 1.0 2.0930e-06 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0␣
↪10  0  0  0   0 10  0  0  0  1911
VecMAXPY               2 1.0 1.1280e-06 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0␣
↪10  0  0  0   0 10  0  0  0  3546
VecNormalize           2 1.0 9.3970e-06 1.0 6.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  1␣
↪14  0  0  0   1 14  0  0  0   638
MatMult                2 1.0 1.1177e-05 1.0 9.99e+03 1.0 0.0e+00 0.0e+00 0.0e+00  1␣
↪24  0  0  0   1 24  0  0  0   894
MatSolve               2 1.0 1.9933e-05 1.0 9.99e+03 1.0 0.0e+00 0.0e+00 0.0e+00  1␣
↪24  0  0  0   1 24  0  0  0   501
MatLUFactorNum         1 1.0 3.5081e-05 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  2␣
↪10  0  0  0   2 10  0  0  0   114
MatILUFactorSym        1 1.0 4.4259e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪0  0  0  0   3  0  0  0  0     0
MatAssemblyBegin       1 1.0 8.2015e-08 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd         1 1.0 3.3536e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪0  0  0  0   2  0  0  0  0     0
MatGetRowIJ            1 1.0 1.5960e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
MatGetOrdering         1 1.0 3.9791e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪0  0  0  0   3  0  0  0  0     0
MatView                2 1.0 6.7909e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  5 ␣
↪0  0  0  0   5  0  0  0  0     0
KSPGMRESOrthog         1 1.0 7.5970e-06 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  1␣
↪10  0  0  0   1 10  0  0  0   526
KSPSetUp               1 1.0 3.4424e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪0  0  0  0   2  0  0  0  0     0
KSPSolve               1 1.0 2.7264e-04 1.0 3.30e+04 1.0 0.0e+00 0.0e+00 0.0e+00 19␣
↪79  0  0  0  19 79  0  0  0   121
PCSetUp                1 1.0 1.5234e-04 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00 11␣
↪10  0  0  0  11 10  0  0  0    26
PCApply                2 1.0 2.1022e-05 1.0 9.99e+03 1.0 0.0e+00 0.0e+00 0.0e+00  1␣
↪24  0  0  0   1 24  0  0  0   475
--------------------------------------------------------------------------------------
↪----------------------------------
```

(continues on next page)

(continued from previous page)

```

Memory usage is given in bytes:

Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.

--- Event Stage 0: Main Stage

              Vector     8              8        76224     0.
              Matrix     2              2       134212     0.
       Krylov Solver     1              1        18400     0.
      Preconditioner     1              1         1032     0.
           Index Set     3              3        10328     0.
              Viewer     1              0            0     0.
====================================================================================================
...
```

1.2.6 Writing Application Codes with PETSc

The examples throughout the library demonstrate the software usage and can serve as templates for developing custom applications. We suggest that new PETSc users examine programs in the directories `${PETSC_DIR}/src/<library>/tutorials` where `<library>` denotes any of the PETSc libraries (listed in the following section), such as `SNES` or `KSP` or `TS`. The manual pages located at `${PETSC_DIR}/docs/index.htm` or https://www.mcs.anl.gov/petsc/documentation/ provide links (organized by both routine names and concepts) to the tutorial examples.

To write a new application program using PETSc, we suggest the following procedure:

1. Install and test PETSc according to the instructions at the PETSc web site.
2. Make a working directory for your source code: for example, `mkdir $HOME/application`
3. Change to that working directory; for example,`cd $HOME/application`
4. Copy one of the examples in the directory that corresponds to the class of problem of interest into your working directory, for example, `cp $PETSC_DIR/src/snes/tutorials/ex19.c ex19.c`
5. Copy $PETSC_DIR/share/petsc/Makefile.user to your working directory, for example, `cp $PETSC_DIR/share/petsc/Makefile.user Makefile`
6. Compile and run the example program, for example, `make ex19; ./ex19`
7. Use the example program as a starting point for developing a custom code.

We highly recommend against the following since it requires changing your makefile for each new configuration/computing system but if you do not wish to include any PETSc utilities in your makefile, you can use the following commands in the PETSc root directory to get the information needed by your makefile:

```
make getlinklibs getincludedirs getcflags getcxxflags getfortranflags getccompiler getfortrancompiler getcxxcompiler
```

All the libraries listed need to be linked into your executable and the include directories and flags need to be passed to the compiler. Usually this is done with `CFLAGS=<list of -I and other flags>` and `FFLAGS=<list of -I and other flags>` in your makefile.

1.2.7 Citing PETSc

If you use the TS component of PETSc please cite the following:

```
@article{abhyankar2018petsc,
  title={PETSc/TS: A Modern Scalable ODE/DAE Solver Library},
  author={Abhyankar, Shrirang and Brown, Jed and Constantinescu, Emil M and Ghosh,␣
↪Debojyoti and Smith, Barry F and Zhang, Hong},
  journal={arXiv preprint arXiv:1806.01437},
  year={2018}
}
```

When citing PETSc in a publication please cite the following:

```
@Misc{petsc-web-page,
   Author = "Satish Balay and Shrirang Abhyankar and Mark~F. Adams and Jed Brown
   and Peter Brune and Kris Buschelman and Lisandro Dalcin and Alp Dener and Victor␣
↪Eijkhout
   and William~D. Gropp and Dinesh Kaushik and Matthew~G. Knepley and Dave~A. May
   and Lois Curfman McInnes and Richard Tran Mills and Todd Munson and Karl Rupp
   and Patrick Sanan and Barry~F. Smith and Stefano Zampini and Hong Zhang and Hong␣
↪Zhang",
   Title  = "{PETS}c {W}eb page",
   Note   = "https://www.mcs.anl.gov/petsc",
   Year   = "2020"}

@TechReport{petsc-user-ref,
   Author = "Satish Balay and Shrirang Abhyankar and Mark~F. Adams and Jed Brown
   and Peter Brune and Kris Buschelman and Lisandro Dalcin and Alp Dener and Victor␣
↪Eijkhout
   and William~D. Gropp and Dinesh Kaushik and Matthew~G. Knepley and Dave~A. May
   and Lois Curfman McInnes and Richard Tran Mills and Todd Munson and Karl Rupp
   and Patrick Sanan and Barry~F. Smith and Stefano Zampini and Hong Zhang and Hong␣
↪Zhang",
   Title       = "{PETS}c Users Manual",
   Number      = "ANL-95/11 - Revision 3.13",
   Institution = "Argonne National Laboratory",
   Year        = "2020"}

@InProceedings{petsc-efficient,
   Author    = "Satish Balay and William D. Gropp and Lois C. McInnes and Barry F.␣
↪Smith",
   Title     = "Efficient Management of Parallelism in Object Oriented
                Numerical Software Libraries",
   Booktitle = "Modern Software Tools in Scientific Computing",
   Editor    = "E. Arge and A. M. Bruaset and H. P. Langtangen",
   Pages     = "163--202",
   Publisher = "Birkhauser Press",
   Year      = "1997"}
```

1.2.8 Directory Structure

We conclude this introduction with an overview of the organization of the PETSc software. The root directory of PETSc contains the following directories:

- `docs` (only in the tarball distribution of PETSc; not the git repository) - All documentation for PETSc. The files `manual.pdf` contains the hyperlinked users manual, suitable for printing or on-screen viewering. Includes the subdirectory - `manualpages` (on-line manual pages).
- `conf` - Base PETSc configuration files that define the standard make variables and rules used by PETSc
- `include` - All include files for PETSc that are visible to the user.
- `include/petsc/finclude` - PETSc include files for Fortran programmers using the .F suffix (recommended).
- `include/petsc/private` - Private PETSc include files that should *not* need to be used by application programmers.
- `share` - Some small test matrices in data files
- `src` - The source code for all PETSc libraries, which currently includes
  - `vec` - vectors,
    - `is` - index sets,
  - `mat` - matrices,
  - `ksp` - complete linear equations solvers,
    - `ksp` - Krylov subspace accelerators,
    - `pc` - preconditioners,
  - `snes` - nonlinear solvers
  - `ts` - ODE solvers and timestepping,
  - `dm` - data management between meshes and solvers, vectors, and matrices,
  - `sys` - general system-related routines,
    - `logging` - PETSc logging and profiling routines,
    - `classes` - low-level classes
      - `draw` - simple graphics,
      - `viewer` - mechanism for printing and visualizing PETSc objects,
      - `bag` - mechanism for saving and loading from disk user data stored in C structs.
      - `random` - random number generators.

Each PETSc source code library directory has the following subdirectories:

- **`tutorials` - Programs designed to teach users about PETSc.** These codes can serve as templates for the design of custom applications.
- **`tests` - Programs designed for thorough testing of PETSc. As** such, these codes are not intended for examination by users.
- `interface` - The calling sequences for the abstract interface to the component. Code here does not know about particular implementations.
- `impls` - Source code for one or more implementations.

- `utils` - Utility routines. Source here may know about the implementations, but ideally will not know about implementations for other components.

CHAPTER

TWO

PROGRAMMING WITH PETSC

2.1 Vectors and Parallel Data

The vector (denoted by `Vec`) is one of the simplest PETSc objects. Vectors are used to store discrete PDE solutions, right-hand sides for linear systems, etc. This chapter is organized as follows:

- (`Vec`) *Creating and Assembling Vectors* and *Basic Vector Operations* - basic usage of vectors
- Section *Indexing and Ordering* - management of the various numberings of degrees of freedom, vertices, cells, etc.
  - (`AO`) Mapping between different global numberings
  - (`ISLocalToGlobalMapping`) Mapping between local and global numberings
- (`DM`) *Structured Grids Using Distributed Arrays* - management of grids
- (`IS`, `VecScatter`) *Vectors Related to Unstructured Grids* - management of vectors related to unstructured grids

2.1.1 Creating and Assembling Vectors

PETSc currently provides two basic vector types: sequential and parallel (MPI-based). To create a sequential vector with `m` components, one can use the command

```
VecCreateSeq(PETSC_COMM_SELF,PetscInt m,Vec *x);
```

To create a parallel vector one can either specify the number of components that will be stored on each process or let PETSc decide. The command

```
VecCreateMPI(MPI_Comm comm,PetscInt m,PetscInt M,Vec *x);
```

creates a vector distributed over all processes in the communicator, `comm`, where `m` indicates the number of components to store on the local process, and `M` is the total number of vector components. Either the local or global dimension, but not both, can be set to `PETSC_DECIDE` or `PETSC_DETERMINE`, respectively, to indicate that PETSc should decide or determine it. More generally, one can use the routines

```
VecCreate(MPI_Comm comm,Vec *v);
VecSetSizes(Vec v, PetscInt m, PetscInt M);
VecSetFromOptions(Vec v);
```

which automatically generates the appropriate vector type (sequential or parallel) over all processes in `comm`. The option `-vec_type mpi` can be used in conjunction with `VecCreate()` and `VecSetFromOptions()` to specify the use of MPI vectors even for the uniprocessor case.

We emphasize that all processes in `comm` *must* call the vector creation routines, since these routines are collective over all processes in the communicator. If you are not familiar with MPI communicators, see the discussion in *Writing PETSc Programs* on page . In addition, if a sequence of `VecCreateXXX()` routines is used, they must be called in the same order on each process in the communicator.

One can assign a single value to all components of a vector with the command

```
VecSet(Vec x,PetscScalar value);
```

Assigning values to individual components of the vector is more complicated, in order to make it possible to write efficient parallel code. Assigning a set of components is a two-step process: one first calls

```
VecSetValues(Vec x,PetscInt n,PetscInt *indices,PetscScalar *values,INSERT_VALUES);
```

any number of times on any or all of the processes. The argument `n` gives the number of components being set in this insertion. The integer array `indices` contains the *global component indices*, and `values` is the array of values to be inserted. Any process can set any components of the vector; PETSc ensures that they are automatically stored in the correct location. Once all of the values have been inserted with `VecSetValues()`, one must call

```
VecAssemblyBegin(Vec x);
```

followed by

```
VecAssemblyEnd(Vec x);
```

to perform any needed message passing of nonlocal components. In order to allow the overlap of communication and calculation, the user's code can perform any series of other actions between these two calls while the messages are in transition.

Example usage of `VecSetValues()` may be found in `$PETSC_DIR/src/vec/vec/tutorials/ex2.c` or `ex2f.F`.

Often, rather than inserting elements in a vector, one may wish to add values. This process is also done with the command

```
VecSetValues(Vec x,PetscInt n,PetscInt *indices, PetscScalar *values,ADD_VALUES);
```

Again one must call the assembly routines `VecAssemblyBegin()` and `VecAssemblyEnd()` after all of the values have been added. Note that addition and insertion calls to `VecSetValues()` *cannot* be mixed. Instead, one must add and insert vector elements in phases, with intervening calls to the assembly routines. This phased assembly procedure overcomes the nondeterministic behavior that would occur if two different processes generated values for the same location, with one process adding while the other is inserting its value. (In this case the addition and insertion actions could be performed in either order, thus resulting in different values at the particular location. Since PETSc does not allow the simultaneous use of `INSERT_VALUES` and `ADD_VALUES` this nondeterministic behavior will not occur in PETSc.)

You can call `VecGetValues()` to pull local values from a vector (but not off-process values), an alternative method for extracting some components of a vector are the vector scatter routines. See *Scatters and Gathers* for details; see also below for `VecGetArray()`.

One can examine a vector with the command

```
VecView(Vec x,PetscViewer v);
```

To print the vector to the screen, one can use the viewer `PETSC_VIEWER_STDOUT_WORLD`, which ensures that parallel vectors are printed correctly to `stdout`. To display the vector in an X-window, one can use the default X-windows viewer `PETSC_VIEWER_DRAW_WORLD`, or one can create a viewer with the routine

`PetscViewerDrawOpenX()`. A variety of viewers are discussed further in *Viewers: Looking at PETSc Objects*.

To create a new vector of the same format as an existing vector, one uses the command

```
VecDuplicate(Vec old,Vec *new);
```

To create several new vectors of the same format as an existing vector, one uses the command

```
VecDuplicateVecs(Vec old,PetscInt n,Vec **new);
```

This routine creates an array of pointers to vectors. The two routines are very useful because they allow one to write library code that does not depend on the particular format of the vectors being used. Instead, the subroutines can automatically correctly create work vectors based on the specified existing vector. As discussed in *Duplicating Multiple Vectors*, the Fortran interface for `VecDuplicateVecs()` differs slightly.

When a vector is no longer needed, it should be destroyed with the command

```
VecDestroy(Vec *x);
```

To destroy an array of vectors, use the command

```
VecDestroyVecs(PetscInt n,Vec **vecs);
```

Note that the Fortran interface for `VecDestroyVecs()` differs slightly, as described in *Duplicating Multiple Vectors*.

It is also possible to create vectors that use an array provided by the user, rather than having PETSc internally allocate the array space. Such vectors can be created with the routines

```
VecCreateSeqWithArray(PETSC_COMM_SELF,PetscInt bs,PetscInt n,PetscScalar *array,Vec␣
↪*V);
```

and

```
VecCreateMPIWithArray(MPI_Comm comm,PetscInt bs,PetscInt n,PetscInt N,PetscScalar␣
↪*array,Vec *vv);
```

Note that here one must provide the value `n`; it cannot be `PETSC_DECIDE` and the user is responsible for providing enough space in the array; `n*sizeof(PetscScalar)`.

2.1.2 Basic Vector Operations

Table 2.1: PETSc Vector Operations

Function Name	**Operation**
`VecAXPY(Vec y,PetscScalar a,Vec x);`	$y = y + a * x$
`VecAYPX(Vec y,PetscScalar a,Vec x);`	$y = x + a * y$
`VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);`	$w = a * x + y$
`VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);`	$y = a * x + b * y$
`VecScale(Vec x, PetscScalar a);`	$x = a * x$
`VecDot(Vec x, Vec y, PetscScalar *r);`	$r = \bar{x}^T * y$
`VecTDot( Vec x, Vec y, PetscScalar *r);`	$r = x' * y$
`VecNorm(Vec x, NormType type, PetscReal *r);`	$r = \|\|x\|\|_{type}$
`VecSum(Vec x, PetscScalar *r);`	$r = \sum x_i$
`VecCopy(Vec x, Vec y);`	$y = x$
`VecSwap(Vec x, Vec y);`	$y = x$ while $x = y$
`VecPointwiseMult(Vec w,Vec x,Vec y);`	$w_i = x_i * y_i$
`VecPointwiseDivide(Vec w,Vec x,Vec y);`	$w_i = x_i / y_i$
`VecMDot(Vec x,PetscInt n,Vec y[],PetscScalar *r);`	$r[i] = \bar{x}^T * y[i]$
`VecMTDot(Vec x,PetscInt n,Vec y[],PetscScalar *r);`	$r[i] = x^T * y[i]$
`VecMAXPY(Vec y,PetscInt n, PetscScalar *a, Vec x[]);`	$y = y + \sum_i a_i * x[i]$
`VecMax(Vec x, PetscInt *idx, PetscReal *r);`	$r = \max x_i$
`VecMin(Vec x, PetscInt *idx, PetscReal *r);`	$r = \min x_i$
`VecAbs(Vec x);`	$x_i = \|x_i\|$
`VecReciprocal(Vec x);`	$x_i = 1/x_i$
`VecShift(Vec x,PetscScalar s);`	$x_i = s + x_i$
`VecSet(Vec x,PetscScalar alpha);`	$x_i = \alpha$

As listed in the table, we have chosen certain basic vector operations to support within the PETSc vector library. These operations were selected because they often arise in application codes. The `NormType` argument to `VecNorm()` is one of `NORM_1`, `NORM_2`, or `NORM_INFINITY`. The 1-norm is $\sum_i |x_i|$, the 2-norm is $(\sum_i x_i^2)^{1/2}$ and the infinity norm is $\max_i |x_i|$.

For parallel vectors that are distributed across the processes by ranges, it is possible to determine a process's local range with the routine

```
VecGetOwnershipRange(Vec vec,PetscInt *low,PetscInt *high);
```

The argument `low` indicates the first component owned by the local process, while `high` specifies *one more than* the last owned by the local process. This command is useful, for instance, in assembling parallel vectors.

On occasion, the user needs to access the actual elements of the vector. The routine `VecGetArray()` returns a pointer to the elements local to the process:

```
VecGetArray(Vec v,PetscScalar **array);
```

When access to the array is no longer needed, the user should call

```
VecRestoreArray(Vec v, PetscScalar **array);
```

If the values do not need to be modified, the routines `VecGetArrayRead()` and `VecRestoreArrayRead()` provide read-only access and should be used instead.

```
VecGetArrayRead(Vec v, const PetscScalar **array);
VecRestoreArrayRead(Vec v, const PetscScalar **array);
```

Minor differences exist in the Fortran interface for `VecGetArray()` and `VecRestoreArray()`, as discussed in *Array Arguments*. It is important to note that `VecGetArray()` and `VecRestoreArray()` do *not* copy the vector elements; they merely give users direct access to the vector elements. Thus, these routines require essentially no time to call and can be used efficiently.

The number of elements stored locally can be accessed with

```
VecGetLocalSize(Vec v,PetscInt *size);
```

The global vector length can be determined by

```
VecGetSize(Vec v,PetscInt *size);
```

In addition to `VecDot()` and `VecMDot()` and `VecNorm()`, PETSc provides split phase versions of these that allow several independent inner products and/or norms to share the same communication (thus improving parallel efficiency). For example, one may have code such as

```
VecDot(Vec x,Vec y,PetscScalar *dot);
VecMDot(Vec x,PetscInt nv, Vec y[],PetscScalar *dot);
VecNorm(Vec x,NormType NORM_2,PetscReal *norm2);
VecNorm(Vec x,NormType NORM_1,PetscReal *norm1);
```

This code works fine, but it performs three separate parallel communication operations. Instead, one can write

```
VecDotBegin(Vec x,Vec y,PetscScalar *dot);
VecMDotBegin(Vec x, PetscInt nv,Vec y[],PetscScalar *dot);
VecNormBegin(Vec x,NormType NORM_2,PetscReal *norm2);
VecNormBegin(Vec x,NormType NORM_1,PetscReal *norm1);
VecDotEnd(Vec x,Vec y,PetscScalar *dot);
VecMDotEnd(Vec x, PetscInt nv,Vec y[],PetscScalar *dot);
VecNormEnd(Vec x,NormType NORM_2,PetscReal *norm2);
VecNormEnd(Vec x,NormType NORM_1,PetscReal *norm1);
```

With this code, the communication is delayed until the first call to `VecxxxEnd()` at which a single MPI reduction is used to communicate all the required values. It is required that the calls to the `VecxxxEnd()` are performed in the same order as the calls to the `VecxxxBegin()`; however, if you mistakenly make the calls in the wrong order, PETSc will generate an error informing you of this. There are additional routines `VecTDotBegin()` and `VecTDotEnd()`, `VecMTDotBegin()`, `VecMTDotEnd()`.

Note: these routines use only MPI-1 functionality; they do not allow you to overlap computation and communication (assuming no threads are spawned within a MPI process). Once MPI-2 implementations are more common we'll improve these routines to allow overlap of inner product and norm calculations with other calculations. Also currently these routines only work for the PETSc built in vector types.

2.1.3 Indexing and Ordering

When writing parallel PDE codes, there is extra complexity caused by having multiple ways of indexing (numbering) and ordering objects such as vertices and degrees of freedom. For example, a grid generator or partitioner may renumber the nodes, requiring adjustment of the other data structures that refer to these objects; see Figure *Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processes)*. In addition, local numbering (on a single process) of objects may be different than the global (cross-process) numbering. PETSc provides a variety of tools to help to manage the mapping amongst the various numbering systems. The two most basic are the `AO` (application ordering), which enables mapping between different global (cross-process) numbering schemes and the `ISLocalToGlobalMapping`, which allows mapping between local (on-process) and global (cross-process) numbering.

Application Orderings

In many applications it is desirable to work with one or more "orderings" (or numberings) of degrees of freedom, cells, nodes, etc. Doing so in a parallel environment is complicated by the fact that each process cannot keep complete lists of the mappings between different orderings. In addition, the orderings used in the PETSc linear algebra routines (often contiguous ranges) may not correspond to the "natural" orderings for the application.

PETSc provides certain utility routines that allow one to deal cleanly and efficiently with the various orderings. To define a new application ordering (called an `AO` in PETSc), one can call the routine

```
AOCreateBasic(MPI_Comm comm,PetscInt n,const PetscInt apordering[],const PetscInt␣
↪petscordering[],AO *ao);
```

The arrays `apordering` and `petscordering`, respectively, contain a list of integers in the application ordering and their corresponding mapped values in the PETSc ordering. Each process can provide whatever subset of the ordering it chooses, but multiple processes should never contribute duplicate values. The argument `n` indicates the number of local contributed values.

For example, consider a vector of length 5, where node 0 in the application ordering corresponds to node 3 in the PETSc ordering. In addition, nodes 1, 2, 3, and 4 of the application ordering correspond, respectively, to nodes 2, 1, 4, and 0 of the PETSc ordering. We can write this correspondence as

$$\{0,1,2,3,4\} \to \{3,2,1,4,0\}.$$

The user can create the PETSc `AO` mappings in a number of ways. For example, if using two processes, one could call

```
AOCreateBasic(PETSC_COMM_WORLD,2,{0,3},{3,4},&ao);
```

on the first process and

```
AOCreateBasic(PETSC_COMM_WORLD,3,{1,2,4},{2,1,0},&ao);
```

on the other process.

Once the application ordering has been created, it can be used with either of the commands

```
AOPetscToApplication(AO ao,PetscInt n,PetscInt *indices);
AOApplicationToPetsc(AO ao,PetscInt n,PetscInt *indices);
```

Upon input, the `n`-dimensional array `indices` specifies the indices to be mapped, while upon output, `indices` contains the mapped values. Since we, in general, employ a parallel database for the `AO` mappings, it is crucial that all processes that called `AOCreateBasic()` also call these routines; these routines *cannot* be called by just a subset of processes in the MPI communicator that was used in the call to `AOCreateBasic()`.

An alternative routine to create the application ordering, `AO`, is

```
AOCreateBasicIS(IS apordering,IS petscordering,AO *ao);
```

where index sets (see *Index Sets*) are used instead of integer arrays.

The mapping routines

```
AOPetscToApplicationIS(AO ao,IS indices);
AOApplicationToPetscIS(AO ao,IS indices);
```

will map index sets (`IS` objects) between orderings. Both the `AOXxxToYyy()` and `AOXxxToYyyIS()` routines can be used regardless of whether the `AO` was created with a `AOCreateBasic()` or `AOCreateBasicIS()`.

The `AO` context should be destroyed with `AODestroy(AO *ao)` and viewed with `AOView(AO ao, PetscViewer viewer)`.

Although we refer to the two orderings as "PETSc" and "application" orderings, the user is free to use them both for application orderings and to maintain relationships among a variety of orderings by employing several `AO` contexts.

The `AOxxToxx()` routines allow negative entries in the input integer array. These entries are not mapped; they simply remain unchanged. This functionality enables, for example, mapping neighbor lists that use negative numbers to indicate nonexistent neighbors due to boundary conditions, etc.

Local to Global Mappings

In many applications one works with a global representation of a vector (usually on a vector obtained with `VecCreateMPI()`) and a local representation of the same vector that includes ghost points required for local computation. PETSc provides routines to help map indices from a local numbering scheme to the PETSc global numbering scheme. This is done via the following routines

```
ISLocalToGlobalMappingCreate(MPI_Comm comm,PetscInt bs,PetscInt N,PetscInt* globalnum,
↪PetscCopyMode mode,ISLocalToGlobalMapping* ctx);
ISLocalToGlobalMappingApply(ISLocalToGlobalMapping ctx,PetscInt n,PetscInt *in,
↪PetscInt *out);
ISLocalToGlobalMappingApplyIS(ISLocalToGlobalMapping ctx,IS isin,IS* isout);
ISLocalToGlobalMappingDestroy(ISLocalToGlobalMapping *ctx);
```

Here `N` denotes the number of local indices, `globalnum` contains the global number of each local number, and `ISLocalToGlobalMapping` is the resulting PETSc object that contains the information needed to apply the mapping with either `ISLocalToGlobalMappingApply()` or `ISLocalToGlobalMappingApplyIS()`.

Note that the `ISLocalToGlobalMapping` routines serve a different purpose than the `AO` routines. In the former case they provide a mapping from a local numbering scheme (including ghost points) to a global numbering scheme, while in the latter they provide a mapping between two global numbering schemes. In fact, many applications may use both `AO` and `ISLocalToGlobalMapping` routines. The `AO` routines are first used to map from an application global ordering (that has no relationship to parallel processing etc.) to the PETSc ordering scheme (where each process has a contiguous set of indices in the numbering). Then in order to perform function or Jacobian evaluations locally on each process, one works with a local numbering scheme that includes ghost points. The mapping from this local numbering scheme back to the global PETSc numbering can be handled with the `ISLocalToGlobalMapping` routines.

If one is given a list of block indices in a global numbering, the routine

```
ISGlobalToLocalMappingApplyBlock(ISLocalToGlobalMapping ctx,
↪ISGlobalToLocalMappingMode type,PetscInt nin,PetscInt idxin[],PetscInt *nout,
↪PetscInt idxout[]);
```

will provide a new list of indices in the local numbering. Again, negative values in `idxin` are left unmapped. But, in addition, if `type` is set to `IS_GTOLM_MASK` , then `nout` is set to `nin` and all global values in `idxin` that are not represented in the local to global mapping are replaced by -1. When `type` is set to `IS_GTOLM_DROP`, the values in `idxin` that are not represented locally in the mapping are not included in `idxout`, so that potentially `nout` is smaller than `nin`. One must pass in an array long enough to hold all the indices. One can call `ISGlobalToLocalMappingApplyBlock()` with `idxout` equal to

`NULL` to determine the required length (returned in `nout`) and then allocate the required space and call `ISGlobalToLocalMappingApplyBlock()` a second time to set the values.

Often it is convenient to set elements into a vector using the local node numbering rather than the global node numbering (e.g., each process may maintain its own sublist of vertices and elements and number them locally). To set values into a vector with the local numbering, one must first call

```
VecSetLocalToGlobalMapping(Vec v,ISLocalToGlobalMapping ctx);
```

and then call

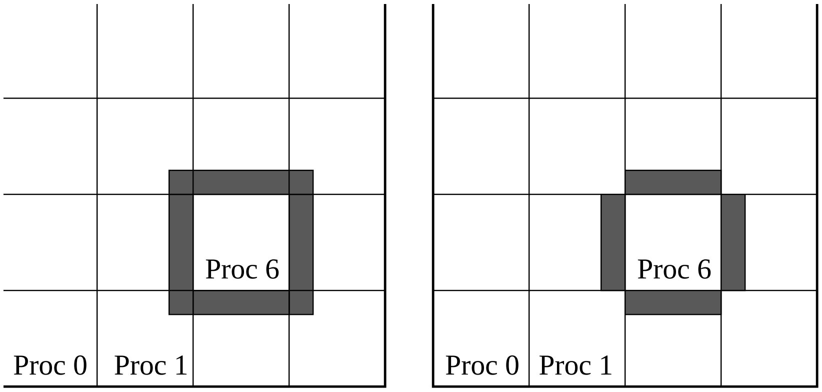
```
VecSetValuesLocal(Vec x,PetscInt n,const PetscInt indices[],const PetscScalar␣
↪values[],INSERT_VALUES);
```

Now the `indices` use the local numbering, rather than the global, meaning the entries lie in $[0, n)$ where n is the local size of the vector.

2.1.4 Structured Grids Using Distributed Arrays

Distributed arrays (DMDAs), which are used in conjunction with PETSc vectors, are intended for use with *logically regular rectangular grids* when communication of nonlocal data is needed before certain local computations can occur. PETSc distributed arrays are designed only for the case in which data can be thought of as being stored in a standard multidimensional array; thus, `DMDA`s are *not* intended for parallelizing unstructured grid problems, etc. DAs are intended for communicating vector (field) information; they are not intended for storing matrices.

For example, a typical situation one encounters in solving PDEs in parallel is that, to evaluate a local function, `f(x)`, each process requires its local portion of the vector `x` as well as its ghost points (the bordering portions of the vector that are owned by neighboring processes). Figure *Ghost Points for Two Stencil Types on the Seventh Process* illustrates the ghost points for the seventh process of a two-dimensional, regular parallel grid. Each box represents a process; the ghost points for the seventh process's local part of a parallel array are shown in gray.



Box-type stencil

Star-type stencil

Fig. 2.1: Ghost Points for Two Stencil Types on the Seventh Process

Creating Distributed Arrays

The PETSc `DMDA` object manages the parallel communication required while working with data stored in regular arrays. The actual data is stored in appropriately sized vector objects; the `DMDA` object only contains the parallel data layout information and communication information, however it may be used to create vectors and matrices with the proper layout.

One creates a distributed array communication data structure in two dimensions with the command

```
DMDACreate2d(MPI_Comm comm,DMBoundaryType xperiod,DMBoundaryType yperiod,
↪DMDAStencilType st,PetscInt M, PetscInt N,PetscInt m,PetscInt n,PetscInt dof,
↪PetscInt s,PetscInt *lx,PetscInt *ly,DM *da);
```

The arguments `M` and `N` indicate the global numbers of grid points in each direction, while `m` and `n` denote the process partition in each direction; `m*n` must equal the number of processes in the MPI communicator, `comm`. Instead of specifying the process layout, one may use `PETSC_DECIDE` for `m` and `n` so that PETSc will determine the partition using MPI. The type of periodicity of the array is specified by `xperiod` and `yperiod`, which can be `DM_BOUNDARY_NONE` (no periodicity), `DM_BOUNDARY_PERIODIC` (periodic in that direction), `DM_BOUNDARY_TWIST` (periodic in that direction, but identified in reverse order), `DM_BOUNDARY_GHOSTED` , or `DM_BOUNDARY_MIRROR`. The argument `dof` indicates the number of degrees of freedom at each array point, and `s` is the stencil width (i.e., the width of the ghost point region). The optional arrays `lx` and `ly` may contain the number of nodes along the x and y axis for each cell, i.e. the dimension of `lx` is `m` and the dimension of `ly` is `n`; alternately, `NULL` may be passed in.

Two types of distributed array communication data structures can be created, as specified by `st`. Star-type stencils that radiate outward only in the coordinate directions are indicated by `DMDA_STENCIL_STAR`, while box-type stencils are specified by `DMDA_STENCIL_BOX`. For example, for the two-dimensional case, `DMDA_STENCIL_STAR` with width 1 corresponds to the standard 5-point stencil, while `DMDA_STENCIL_BOX` with width 1 denotes the standard 9-point stencil. In both instances the ghost points are identical, the only difference being that with star-type stencils certain ghost points are ignored, decreasing substantially the number of messages sent. Note that the `DMDA_STENCIL_STAR` stencils can save interprocess communication in two and three dimensions.

These DMDA stencils have nothing directly to do with any finite difference stencils one might chose to use for a discretization; they only ensure that the correct values are in place for application of a user-defined finite difference stencil (or any other discretization technique).

The commands for creating distributed array communication data structures in one and three dimensions are analogous:

```
DMDACreate1d(MPI_Comm comm,DMBoundaryType xperiod,PetscInt M,PetscInt w,PetscInt s,
↪PetscInt *lc,DM *inra);
DMDACreate3d(MPI_Comm comm,DMBoundaryType xperiod,DMBoundaryType yperiod,
↪DMBoundaryType zperiod, DMDAStencilType stencil_type,PetscInt M,PetscInt N,PetscInt␣
↪P,PetscInt m,PetscInt n,PetscInt p,PetscInt w,PetscInt s,PetscInt *lx,PetscInt *ly,
↪PetscInt *lz,DM *inra);
```

The routines to create distributed arrays are collective, so that all processes in the communicator `comm` must call `DACreateXXX()`.

Local/Global Vectors and Scatters

Each `DMDA` object defines the layout of two vectors: a distributed global vector and a local vector that includes room for the appropriate ghost points. The `DMDA` object provides information about the size and layout of these vectors, but does not internally allocate any associated storage space for field values. Instead, the user can create vector objects that use the `DMDA` layout information with the routines

```
DMCreateGlobalVector(DM da,Vec *g);
DMCreateLocalVector(DM da,Vec *l);
```

These vectors will generally serve as the building blocks for local and global PDE solutions, etc. If additional vectors with such layout information are needed in a code, they can be obtained by duplicating `l` or `g` via `VecDuplicate()` or `VecDuplicateVecs()`.

We emphasize that a distributed array provides the information needed to communicate the ghost value information between processes. In most cases, several different vectors can share the same communication information (or, in other words, can share a given `DMDA`). The design of the `DMDA` object makes this easy, as each `DMDA` operation may operate on vectors of the appropriate size, as obtained via `DMCreateLocalVector()` and `DMCreateGlobalVector()` or as produced by `VecDuplicate()`. As such, the `DMDA` scatter/gather operations (e.g., `DMGlobalToLocalBegin()`) require vector input/output arguments, as discussed below.

PETSc currently provides no container for multiple arrays sharing the same distributed array communication; note, however, that the `dof` parameter handles many cases of interest.

At certain stages of many applications, there is a need to work on a local portion of the vector, including the ghost points. This may be done by scattering a global vector into its local parts by using the two-stage commands

```
DMGlobalToLocalBegin(DM da,Vec g,InsertMode iora,Vec l);
DMGlobalToLocalEnd(DM da,Vec g,InsertMode iora,Vec l);
```

which allow the overlap of communication and computation. Since the global and local vectors, given by `g` and `l`, respectively, must be compatible with the distributed array, `da`, they should be generated by `DMCreateGlobalVector()` and `DMCreateLocalVector()` (or be duplicates of such a vector obtained via `VecDuplicate()`). The `InsertMode` can be either `ADD_VALUES` or `INSERT_VALUES`.

One can scatter the local patches into the distributed vector with the command

```
DMLocalToGlobal(DM da,Vec l,InsertMode mode,Vec g);
```

or the commands

```
DMLocalToGlobalBegin(DM da,Vec l,InsertMode mode,Vec g);
/* (Computation to overlap with communication) */
DMLocalToGlobalEnd(DM da,Vec l,InsertMode mode,Vec g);
```

In general this is used with an `InsertMode` of `ADD_VALUES`, because if one wishes to insert values into the global vector they should just access the global vector directly and put in the values.

A third type of distributed array scatter is from a local vector (including ghost points that contain irrelevant values) to a local vector with correct ghost point values. This scatter may be done with the commands

```
DMLocalToLocalBegin(DM da,Vec l1,InsertMode iora,Vec l2);
DMLocalToLocalEnd(DM da,Vec l1,InsertMode iora,Vec l2);
```

Since both local vectors, `l1` and `l2`, must be compatible with the distributed array, `da`, they should be generated by `DMCreateLocalVector()` (or be duplicates of such vectors obtained via `VecDuplicate()`).

The `InsertMode` can be either `ADD_VALUES` or `INSERT_VALUES`.

It is possible to directly access the vector scatter contexts (see below) used in the local-to-global (`ltog`), global-to-local (`gtol`), and local-to-local (`ltol`) scatters with the command

```
DMDAGetScatter(DM da,VecScatter *ltog,VecScatter *gtol,VecScatter *ltol);
```

Most users should not need to use these contexts.

Local (Ghosted) Work Vectors

In most applications the local ghosted vectors are only needed during user "function evaluations". PETSc provides an easy, light-weight (requiring essentially no CPU time) way to obtain these work vectors and return them when they are no longer needed. This is done with the routines

```
DMGetLocalVector(DM da,Vec *l);
... use the local vector l ...
DMRestoreLocalVector(DM da,Vec *l);
```

Accessing the Vector Entries for DMDA Vectors

PETSc provides an easy way to set values into the DMDA Vectors and access them using the natural grid indexing. This is done with the routines

```
DMDAVecGetArray(DM da,Vec l,void *array);
... use the array indexing it with 1 or 2 or 3 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArray(DM da,Vec l,void *array);
DMDAVecGetArrayRead(DM da,Vec l,void *array);
... use the array indexing it with 1 or 2 or 3 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArrayRead(DM da,Vec l,void *array);
```

and

```
DMDAVecGetArrayDOF(DM da,Vec l,void *array);
... use the array indexing it with 1 or 2 or 3 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArrayDOF(DM da,Vec l,void *array);
DMDAVecGetArrayDOFRead(DM da,Vec l,void *array);
... use the array indexing it with 1 or 2 or 3 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArrayDOFRead(DM da,Vec l,void *array);
```

where `array` is a multidimensional C array with the same dimension as `da`. The vector `l` can be either a global vector or a local vector. The `array` is accessed using the usual *global* indexing on the entire grid, but the user may *only* refer to the local and ghost entries of this array as all other entries are undefined. For example, for a scalar problem in two dimensions one could use

```
PetscScalar **f,**u;
...
DMDAVecGetArray(DM da,Vec local,&u);
DMDAVecGetArray(DM da,Vec global,&f);
...
  f[i][j] = u[i][j] - ...
```

(continues on next page)

(continued from previous page)

```
...
DMDAVecRestoreArray(DM da,Vec local,&u);
DMDAVecRestoreArray(DM da,Vec global,&f);
```

The recommended approach for multi-component PDEs is to declare a `struct` representing the fields defined at each node of the grid, e.g.

```
typedef struct {
  PetscScalar u,v,omega,temperature;
} Node;
```

and write residual evaluation using

```
Node **f,**u;
DMDAVecGetArray(DM da,Vec local,&u);
DMDAVecGetArray(DM da,Vec global,&f);
 ...
    f[i][j].omega = ...
 ...
DMDAVecRestoreArray(DM da,Vec local,&u);
DMDAVecRestoreArray(DM da,Vec global,&f);
```

See SNES Tutorial ex5 for a complete example and see SNES Tutorial ex19 for an example for a multi-component PDE.

Grid Information

The global indices of the lower left corner of the local portion of the array as well as the local array size can be obtained with the commands

```
DMDAGetCorners(DM da,PetscInt *x,PetscInt *y,PetscInt *z,PetscInt *m,PetscInt *n,
↪PetscInt *p);
DMDAGetGhostCorners(DM da,PetscInt *x,PetscInt *y,PetscInt *z,PetscInt *m,PetscInt *n,
↪PetscInt *p);
```

The first version excludes any ghost points, while the second version includes them. The routine `DMDAGetGhostCorners()` deals with the fact that subarrays along boundaries of the problem domain have ghost points only on their interior edges, but not on their boundary edges.

When either type of stencil is used, `DMDA_STENCIL_STAR` or `DMDA_STENCIL_BOX`, the local vectors (with the ghost points) represent rectangular arrays, including the extra corner elements in the `DMDA_STENCIL_STAR` case. This configuration provides simple access to the elements by employing two- (or three-) dimensional indexing. The only difference between the two cases is that when `DMDA_STENCIL_STAR` is used, the extra corner components are *not* scattered between the processes and thus contain undefined values that should *not* be used.

To assemble global stiffness matrices, one can use these global indices with `MatSetValues()` or `MatSetValuesStencil()`. Alternately, the global node number of each local node, including the ghost nodes, can be obtained by calling

```
DMGetLocalToGlobalMapping(DM da,ISLocalToGlobalMapping *map);
```

followed by

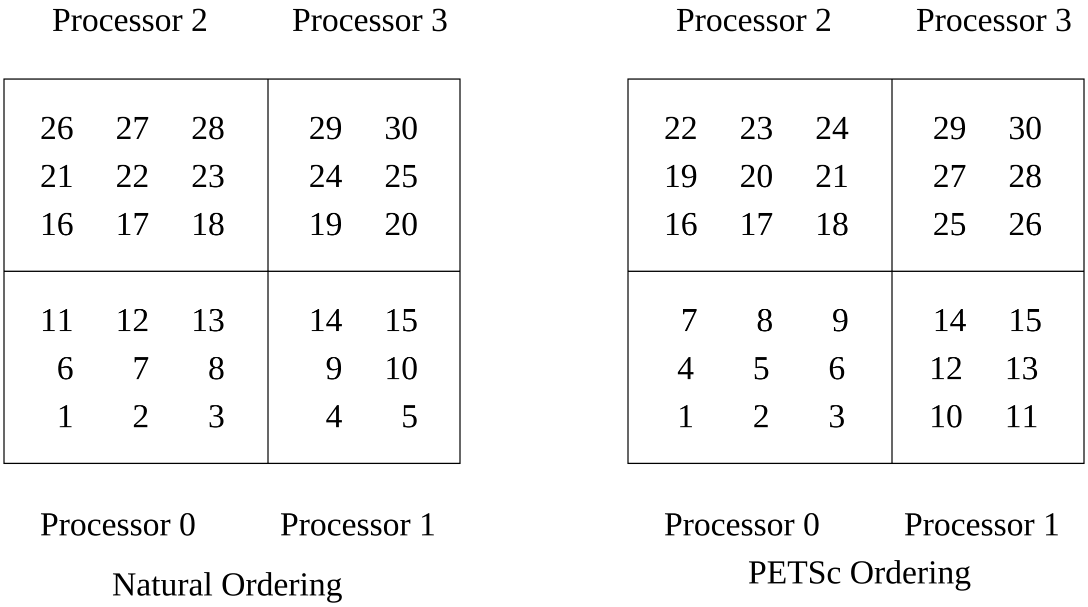
```
VecSetLocalToGlobalMapping(Vec v,ISLocalToGlobalMapping map);
MatSetLocalToGlobalMapping(Mat A,ISLocalToGlobalMapping rmapping,
↪ISLocalToGlobalMapping cmapping);
```

Now entries may be added to the vector and matrix using the local numbering and `VecSetValuesLocal()` and `MatSetValuesLocal()`.

Since the global ordering that PETSc uses to manage its parallel vectors (and matrices) does not usually correspond to the "natural" ordering of a two- or three-dimensional array, the `DMDA` structure provides an application ordering `AO` (see *Application Orderings*) that maps between the natural ordering on a rectangular grid and the ordering PETSc uses to parallelize. This ordering context can be obtained with the command

```
DMDAGetAO(DM da,AO *ao);
```

In Figure *Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processes)* we indicate the orderings for a two-dimensional distributed array, divided among four processes.



Natural Ordering

26	27	28	29	30
21	22	23	24	25
16	17	18	19	20
11	12	13	14	15
6	7	8	9	10
1	2	3	4	5

PETSc Ordering

22	23	24	29	30
19	20	21	27	28
16	17	18	25	26
7	8	9	14	15
4	5	6	12	13
1	2	3	10	11

Fig. 2.2: Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processes)

The example SNES Tutorial ex5 illustrates the use of a distributed array in the solution of a nonlinear problem. The analogous Fortran program is SNES Tutorial ex5f; see *SNES: Nonlinear Solvers* for a discussion of the nonlinear solvers.

Staggered Grids

For regular grids with staggered data (living on elements, faces, edges, and/or vertices), the `DMStag` object is available. It behaves much like `DMDA`; see the `DMSTAG` manual page for more information.

2.1.5 Vectors Related to Unstructured Grids

Index Sets

To facilitate general vector scatters and gathers used, for example, in updating ghost points for problems defined on unstructured grids[1], PETSc employs the concept of an *index set*, via the `IS` class. An index set, which is a generalization of a set of integer indices, is used to define scatters, gathers, and similar operations on vectors and matrices.

The following command creates an index set based on a list of integers:

```
ISCreateGeneral(MPI_Comm comm,PetscInt n,PetscInt *indices,PetscCopyMode mode, IS␣
↪*is);
```

When `mode` is `PETSC_COPY_VALUES`, this routine copies the `n` indices passed to it by the integer array `indices`. Thus, the user should be sure to free the integer array `indices` when it is no longer needed, perhaps directly after the call to `ISCreateGeneral()`. The communicator, `comm`, should consist of all processes that will be using the `IS`.

Another standard index set is defined by a starting point (`first`) and a stride (`step`), and can be created with the command

```
ISCreateStride(MPI_Comm comm,PetscInt n,PetscInt first,PetscInt step,IS *is);
```

Index sets can be destroyed with the command

```
ISDestroy(IS &is);
```

On rare occasions the user may need to access information directly from an index set. Several commands assist in this process:

```
ISGetSize(IS is,PetscInt *size);
ISStrideGetInfo(IS is,PetscInt *first,PetscInt *stride);
ISGetIndices(IS is,PetscInt **indices);
```

The function `ISGetIndices()` returns a pointer to a list of the indices in the index set. For certain index sets, this may be a temporary array of indices created specifically for a given routine. Thus, once the user finishes using the array of indices, the routine

```
ISRestoreIndices(IS is, PetscInt **indices);
```

should be called to ensure that the system can free the space it may have used to generate the list of indices.

A blocked version of the index sets can be created with the command

```
ISCreateBlock(MPI_Comm comm,PetscInt bs,PetscInt n,PetscInt *indices,PetscCopyMode␣
↪mode, IS *is);
```

This version is used for defining operations in which each element of the index set refers to a block of `bs` vector entries. Related routines analogous to those described above exist as well, including `ISBlockGetIndices()`, `ISBlockGetSize()`, `ISBlockGetLocalSize()`, `ISGetBlockSize()`. See the man pages for details.

---

[1] Also see `DMPlex` (*DMPlex: Unstructured Grids in PETSc*), an abstraction for working with unstructured grids.

Scatters and Gathers

PETSc vectors have full support for general scatters and gathers. One can select any subset of the components of a vector to insert or add to any subset of the components of another vector. We refer to these operations as *generalized scatters*, though they are actually a combination of scatters and gathers.

To copy selected components from one vector to another, one uses the following set of commands:

```
VecScatterCreate(Vec x,IS ix,Vec y,IS iy,VecScatter *ctx);
VecScatterBegin(VecScatter ctx,Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD);
VecScatterEnd(VecScatter ctx,Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD);
VecScatterDestroy(VecScatter *ctx);
```

Here `ix` denotes the index set of the first vector, while `iy` indicates the index set of the destination vector. The vectors can be parallel or sequential. The only requirements are that the number of entries in the index set of the first vector, `ix`, equals the number in the destination index set, `iy`, and that the vectors be long enough to contain all the indices referred to in the index sets. If both `x` and `y` are parallel, their communicator must have the same set of processes, but their process order can be different. The argument `INSERT_VALUES` specifies that the vector elements will be inserted into the specified locations of the destination vector, overwriting any existing values. To add the components, rather than insert them, the user should select the option `ADD_VALUES` instead of `INSERT_VALUES`. One can also use `MAX_VALUES` or `MIN_VALUES` to replace destination with the maximal or minimal of its current value and the scattered values.

To perform a conventional gather operation, the user simply makes the destination index set, `iy`, be a stride index set with a stride of one. Similarly, a conventional scatter can be done with an initial (sending) index set consisting of a stride. The scatter routines are collective operations (i.e. all processes that own a parallel vector *must* call the scatter routines). When scattering from a parallel vector to sequential vectors, each process has its own sequential vector that receives values from locations as indicated in its own index set. Similarly, in scattering from sequential vectors to a parallel vector, each process has its own sequential vector that makes contributions to the parallel vector.

Caution: When `INSERT_VALUES` is used, if two different processes contribute different values to the same component in a parallel vector, either value may end up being inserted. When `ADD_VALUES` is used, the correct sum is added to the correct location.

In some cases one may wish to “undo” a scatter, that is perform the scatter backwards, switching the roles of the sender and receiver. This is done by using

```
VecScatterBegin(VecScatter ctx,Vec y,Vec x,INSERT_VALUES,SCATTER_REVERSE);
VecScatterEnd(VecScatter ctx,Vec y,Vec x,INSERT_VALUES,SCATTER_REVERSE);
```

Note that the roles of the first two arguments to these routines must be swapped whenever the `SCATTER_REVERSE` option is used.

Once a `VecScatter` object has been created it may be used with any vectors that have the appropriate parallel data layout. That is, one can call `VecScatterBegin()` and `VecScatterEnd()` with different vectors than used in the call to `VecScatterCreate()` as long as they have the same parallel layout (number of elements on each process are the same). Usually, these “different” vectors would have been obtained via calls to `VecDuplicate()` from the original vectors used in the call to `VecScatterCreate()`.

There is a PETSc routine that is nearly the opposite of `VecSetValues()`, that is, `VecGetValues()`, but it can only get local values from the vector. To get off-process values, the user should create a new vector where the components are to be stored, and then perform the appropriate vector scatter. For example, if one desires to obtain the values of the 100th and 200th entries of a parallel vector, `p`, one could use a code such as that below. In this example, the values of the 100th and 200th components are placed in the array values. In this example each process now has the 100th and 200th component, but obviously each process could gather any elements it needed, or none by creating an index set with no entries.

```
Vec         p, x;         /* initial vector, destination vector */
VecScatter  scatter;      /* scatter context */
IS          from, to;     /* index sets that define the scatter */
PetscScalar *values;
PetscInt    idx_from[] = {100,200}, idx_to[] = {0,1};

VecCreateSeq(PETSC_COMM_SELF,2,&x);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_from,PETSC_COPY_VALUES,&from);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_to,PETSC_COPY_VALUES,&to);
VecScatterCreate(p,from,x,to,&scatter);
VecScatterBegin(scatter,p,x,INSERT_VALUES,SCATTER_FORWARD);
VecScatterEnd(scatter,p,x,INSERT_VALUES,SCATTER_FORWARD);
VecGetArray(x,&values);
ISDestroy(&from);
ISDestroy(&to);
VecScatterDestroy(&scatter);
```

The scatter comprises two stages, in order to allow overlap of communication and computation. The introduction of the `VecScatter` context allows the communication patterns for the scatter to be computed once and then reused repeatedly. Generally, even setting up the communication for a scatter requires communication; hence, it is best to reuse such information when possible.

Scattering Ghost Values

Generalized scatters provide a very general method for managing the communication of required ghost values for unstructured grid computations. One scatters the global vector into a local "ghosted" work vector, performs the computation on the local work vectors, and then scatters back into the global solution vector. In the simplest case this may be written as

```
VecScatterBegin(VecScatter scatter,Vec globalin,Vec localin,InsertMode INSERT_VALUES,␣
↪ScatterMode SCATTER_FORWARD);
VecScatterEnd(VecScatter scatter,Vec globalin,Vec localin,InsertMode INSERT_VALUES,
↪ScatterMode SCATTER_FORWARD);
/* For example, do local calculations from localin to localout */
 ...
VecScatterBegin(VecScatter scatter,Vec localout,Vec globalout,InsertMode ADD_VALUES,
↪ScatterMode SCATTER_REVERSE);
VecScatterEnd(VecScatter scatter,Vec localout,Vec globalout,InsertMode ADD_VALUES,
↪ScatterMode SCATTER_REVERSE);
```

Vectors with Locations for Ghost Values

There are two minor drawbacks to the basic approach described above:

- the extra memory requirement for the local work vector, `localin`, which duplicates the memory in `globalin`, and
- the extra time required to copy the local values from `localin` to `globalin`.

An alternative approach is to allocate global vectors with space preallocated for the ghost values; this may be done with either

```
VecCreateGhost(MPI_Comm comm,PetscInt n,PetscInt N,PetscInt nghost,PetscInt *ghosts,
↪Vec *vv)
```

or

```
VecCreateGhostWithArray(MPI_Comm comm,PetscInt n,PetscInt N,PetscInt nghost,PetscInt␣
↪*ghosts,PetscScalar *array,Vec *vv)
```

Here `n` is the number of local vector entries, `N` is the number of global entries (or `NULL`) and `nghost` is the number of ghost entries. The array `ghosts` is of size `nghost` and contains the global vector location for each local ghost location. Using `VecDuplicate()` or `VecDuplicateVecs()` on a ghosted vector will generate additional ghosted vectors.

In many ways, a ghosted vector behaves just like any other MPI vector created by `VecCreateMPI()`. The difference is that the ghosted vector has an additional "local" representation that allows one to access the ghost locations. This is done through the call to

```
VecGhostGetLocalForm(Vec g,Vec *l);
```

The vector `l` is a sequential representation of the parallel vector `g` that shares the same array space (and hence numerical values); but allows one to access the "ghost" values past "the end of the" array. Note that one access the entries in `l` using the local numbering of elements and ghosts, while they are accessed in `g` using the global numbering.

A common usage of a ghosted vector is given by

```
VecGhostUpdateBegin(Vec globalin,InsertMode INSERT_VALUES, ScatterMode SCATTER_
↪FORWARD);
VecGhostUpdateEnd(Vec globalin,InsertMode INSERT_VALUES, ScatterMode SCATTER_FORWARD);
VecGhostGetLocalForm(Vec globalin,Vec *localin);
VecGhostGetLocalForm(Vec globalout,Vec *localout);
...  Do local calculations from localin to localout ...
VecGhostRestoreLocalForm(Vec globalin,Vec *localin);
VecGhostRestoreLocalForm(Vec globalout,Vec *localout);
VecGhostUpdateBegin(Vec globalout,InsertMode ADD_VALUES, ScatterMode SCATTER_REVERSE);
VecGhostUpdateEnd(Vec globalout,InsertMode ADD_VALUES, ScatterMode SCATTER_REVERSE);
```

The routines `VecGhostUpdateBegin()` and `VecGhostUpdateEnd()` are equivalent to the routines `VecScatterBegin()` and `VecScatterEnd()` above except that since they are scattering into the ghost locations, they do not need to copy the local vector values, which are already in place. In addition, the user does not have to allocate the local work vector, since the ghosted vector already has allocated slots to contain the ghost values.

The input arguments `INSERT_VALUES` and `SCATTER_FORWARD` cause the ghost values to be correctly updated from the appropriate process. The arguments `ADD_VALUES` and `SCATTER_REVERSE` update the "local" portions of the vector from all the other processes' ghost values. This would be appropriate, for example, when performing a finite element assembly of a load vector. One can also use `MAX_VALUES` or `MIN_VALUES` with `SCATTER_REVERSE`.

Partitioning discusses the important topic of partitioning an unstructured grid.

2.2 Matrices

PETSc provides a variety of matrix implementations because no single matrix format is appropriate for all problems. Currently, we support dense storage and compressed sparse row storage (both sequential and parallel versions), as well as several specialized formats. Additional formats can be added.

This chapter describes the basics of using PETSc matrices in general (regardless of the particular format chosen) and discusses tips for efficient use of the several simple uniprocess and parallel matrix types. The use of PETSc matrices involves the following actions: create a particular type of matrix, insert values into it, process the matrix, use the matrix for various computations, and finally destroy the matrix. The application code does not need to know or care about the particular storage formats of the matrices.

2.2.1 Creating and Assembling Matrices

The simplest routine for forming a PETSc matrix, `A`, is followed by

```
MatCreate(MPI_Comm comm,Mat *A)
MatSetSizes(Mat A,PetscInt m,PetscInt n,PetscInt M,PetscInt N)
```

This routine generates a sequential matrix when running one process and a parallel matrix for two or more processes; the particular matrix format is set by the user via options database commands. The user specifies either the global matrix dimensions, given by `M` and `N` or the local dimensions, given by `m` and `n` while PETSc completely controls memory allocation. This routine facilitates switching among various matrix types, for example, to determine the format that is most efficient for a certain application. By default, `MatCreate()` employs the sparse AIJ format, which is discussed in detail *Sparse Matrices*. See the manual pages for further information about available matrix formats.

To insert or add entries to a matrix, one can call a variant of `MatSetValues()`, either

```
MatSetValues(Mat A,PetscInt m,const PetscInt idxm[],PetscInt n,const PetscInt idxn[],
↪const PetscScalar values[],INSERT_VALUES);
```

or

```
MatSetValues(Mat A,PetscInt m,const PetscInt idxm[],PetscInt n,const PetscInt idxn[],
↪const PetscScalar values[],ADD_VALUES);
```

This routine inserts or adds a logically dense subblock of dimension `m*n` into the matrix. The integer indices `idxm` and `idxn`, respectively, indicate the global row and column numbers to be inserted. `MatSetValues()` uses the standard C convention, where the row and column matrix indices begin with zero *regardless of the storage format employed*. The array `values` is logically two-dimensional, containing the values that are to be inserted. By default the values are given in row major order, which is the opposite of the Fortran convention, meaning that the value to be put in row `idxm[i]` and column `idxn[j]` is located in `values[i*n+j]`. To allow the insertion of values in column major order, one can call the command

```
MatSetOption(Mat A,MAT_ROW_ORIENTED,PETSC_FALSE);
```

Warning: Several of the sparse implementations do *not* currently support the column-oriented option.

This notation should not be a mystery to anyone. For example, to insert one matrix into another when using MATLAB, one uses the command `A(im,in) = B;` where `im` and `in` contain the indices for the rows and columns. This action is identical to the calls above to `MatSetValues()`.

When using the block compressed sparse row matrix format (`MATSEQBAIJ` or `MATMPIBAIJ`), one can insert elements more efficiently using the block variant, `MatSetValuesBlocked()` or `MatSetValuesBlockedLocal()`.

The function `MatSetOption()` accepts several other inputs; see the manual page for details.

After the matrix elements have been inserted or added into the matrix, they must be processed (also called “assembled”) before they can be used. The routines for matrix processing are

```
MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

By placing other code between these two calls, the user can perform computations while messages are in transit. Calls to `MatSetValues()` with the `INSERT_VALUES` and `ADD_VALUES` options *cannot* be mixed without intervening calls to the assembly routines. For such intermediate assembly calls the second routine argument typically should be `MAT_FLUSH_ASSEMBLY`, which omits some of the work of the full assembly process. `MAT_FINAL_ASSEMBLY` is required only in the last matrix assembly before a matrix is used.

Even though one may insert values into PETSc matrices without regard to which process eventually stores them, for efficiency reasons we usually recommend generating most entries on the process where they are destined to be stored. To help the application programmer with this task for matrices that are distributed across the processes by ranges, the routine

```
MatGetOwnershipRange(Mat A,PetscInt *first_row,PetscInt *last_row);
```

informs the user that all rows from `first_row` to `last_row-1` (since the value returned in `last_row` is one more than the global index of the last local row) will be stored on the local process.

In the sparse matrix implementations, once the assembly routines have been called, the matrices are compressed and can be used for matrix-vector multiplication, etc. Any space for preallocated nonzeros that was not filled by a call to `MatSetValues()` or a related routine is compressed out by assembling with `MAT_FINAL_ASSEMBLY`. If you intend to use that extra space later, be sure to insert explicit zeros before assembling with `MAT_FINAL_ASSEMBLY` so the space will not be compressed out. Once the matrix has been assembled, inserting new values will be expensive since it will require copies and possible memory allocation.

One may repeatedly assemble matrices that retain the same nonzero pattern (such as within a nonlinear or time-dependent problem). Where possible, data structures and communication information will be reused (instead of regenerated) during successive steps, thereby increasing efficiency. See KSP Tutorial ex5 for a simple example of solving two linear systems that use the same matrix data structure.

Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the Yale sparse matrix format or compressed sparse row format, CSR). This section discusses tips for *efficiently* using this matrix format for large-scale applications. Additional formats (such as block compressed row and block diagonal storage, which are generally much more efficient for problems with multiple degrees of freedom per node) are discussed below. Beginning users need not concern themselves initially with such details and may wish to proceed directly to *Basic Matrix Operations*. However, when an application code progresses to the point of tuning for efficiency and/or generating timing results, it is *crucial* to read this information.

Sequential AIJ Sparse Matrices

In the PETSc AIJ matrix formats, we store the nonzero elements by rows, along with an array of corresponding column numbers and an array of pointers to the beginning of each row. Note that the diagonal matrix entries are stored with the rest of the nonzeros (not separately).

To create a sequential AIJ sparse matrix, `A`, with `m` rows and `n` columns, one uses the command

```
MatCreateSeqAIJ(PETSC_COMM_SELF,PetscInt m,PetscInt n,PetscInt nz,PetscInt *nnz,Mat␣
↪*A);
```

where `nz` or `nnz` can be used to preallocate matrix memory, as discussed below. The user can set `nz=0` and `nnz=NULL` for PETSc to control all matrix memory allocation.

The sequential and parallel AIJ matrix storage formats by default employ *i-nodes* (identical nodes) when possible. We search for consecutive rows with the same nonzero structure, thereby reusing matrix information for increased efficiency. Related options database keys are `-mat_no_inode` (do not use inodes) and `-mat_inode_limit <limit>` (set inode limit (max limit=5)). Note that problems with a single degree of freedom per grid node will automatically not use I-nodes.

The internal data representation for the AIJ formats employs zero-based indexing.

Preallocation of Memory for Sequential AIJ Sparse Matrices

The dynamic process of allocating new memory and copying from the old storage to the new is *intrinsically very expensive*. Thus, to obtain good performance when assembling an AIJ matrix, it is crucial to preallocate the memory needed for the sparse matrix. The user has two choices for preallocating matrix memory via `MatCreateSeqAIJ()`.

One can use the scalar `nz` to specify the expected number of nonzeros for each row. This is generally fine if the number of nonzeros per row is roughly the same throughout the matrix (or as a quick and easy first step for preallocation). If one underestimates the actual number of nonzeros in a given row, then during the assembly process PETSc will automatically allocate additional needed space. However, this extra memory allocation can slow the computation,

If different rows have very different numbers of nonzeros, one should attempt to indicate (nearly) the exact number of elements intended for the various rows with the optional array, `nnz` of length `m`, where `m` is the number of rows, for example

```
PetscInt nnz[m];
nnz[0] = <nonzeros in row 0>
nnz[1] = <nonzeros in row 1>
....
nnz[m-1] = <nonzeros in row m-1>
```

In this case, the assembly process will require no additional memory allocations if the `nnz` estimates are correct. If, however, the `nnz` estimates are incorrect, PETSc will automatically obtain the additional needed space, at a slight loss of efficiency.

Using the array `nnz` to preallocate memory is especially important for efficient matrix assembly if the number of nonzeros varies considerably among the rows. One can generally set `nnz` either by knowing in advance the problem structure (e.g., the stencil for finite difference problems on a structured grid) or by precomputing the information by using a segment of code similar to that for the regular matrix assembly. The overhead of determining the `nnz` array will be quite small compared with the overhead of the inherently expensive `malloc`s and moves of data that are needed for dynamic allocation during matrix assembly. Always guess high if an exact value is not known (extra space is cheaper than too little).

Thus, when assembling a sparse matrix with very different numbers of nonzeros in various rows, one could proceed as follows for finite difference methods:

1. Allocate integer array `nnz`.
2. Loop over grid, counting the expected number of nonzeros for the row(s) associated with the various grid points.
3. Create the sparse matrix via `MatCreateSeqAIJ()` or alternative.
4. Loop over the grid, generating matrix entries and inserting in matrix via `MatSetValues()`.

For (vertex-based) finite element type calculations, an analogous procedure is as follows:

1. Allocate integer array `nnz`.
2. Loop over vertices, computing the number of neighbor vertices, which determines the number of nonzeros for the corresponding matrix row(s).

3. Create the sparse matrix via `MatCreateSeqAIJ()` or alternative.
4. Loop over elements, generating matrix entries and inserting in matrix via `MatSetValues()`.

The `-info` option causes the routines `MatAssemblyBegin()` and `MatAssemblyEnd()` to print information about the success of the preallocation. Consider the following example for the `MATSEQAIJ` matrix format:

```
MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:20 unneeded, 100 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 0
```

The first line indicates that the user preallocated 120 spaces but only 100 were used. The second line indicates that the user preallocated enough space so that PETSc did not have to internally allocate additional space (an expensive operation). In the next example the user did not preallocate sufficient space, as indicated by the fact that the number of mallocs is very large (bad for efficiency):

```
MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:47 unneeded, 1000 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 40000
```

Although at first glance such procedures for determining the matrix structure in advance may seem unusual, they are actually very efficient because they alleviate the need for dynamic construction of the matrix data structure, which can be very expensive.

Parallel AIJ Sparse Matrices

Parallel sparse matrices with the AIJ format can be created with the command

```
MatCreateAIJ=(MPI_Comm comm,PetscInt m,PetscInt n,PetscInt M,PetscInt N,PetscInt d_nz,
↪PetscInt *d_nnz, PetscInt o_nz,PetscInt *o_nnz,Mat *A);
```

`A` is the newly created matrix, while the arguments `m`, `M`, and `N`, indicate the number of local rows and the number of global rows and columns, respectively. In the PETSc partitioning scheme, all the matrix columns are local and `n` is the number of columns corresponding to local part of a parallel vector. Either the local or global parameters can be replaced with `PETSC_DECIDE`, so that PETSc will determine them. The matrix is stored with a fixed number of rows on each process, given by `m`, or determined by PETSc if `m` is `PETSC_DECIDE`.

If `PETSC_DECIDE` is not used for the arguments `m` and `n`, then the user must ensure that they are chosen to be compatible with the vectors. To do this, one first considers the matrix-vector product $y = Ax$. The `m` that is used in the matrix creation routine `MatCreateAIJ()` must match the local size used in the vector creation routine `VecCreateMPI()` for `y`. Likewise, the `n` used must match that used as the local size in `VecCreateMPI()` for `x`.

The user must set `d_nz=0`, `o_nz=0`, `d_nnz=`NULL, and `o_nnz=NULL` for PETSc to control dynamic allocation of matrix memory space. Analogous to `nz` and `nnz` for the routine `MatCreateSeqAIJ()`, these arguments optionally specify nonzero information for the diagonal (`d_nz` and `d_nnz`) and off-diagonal (`o_nz` and `o_nnz`) parts of the matrix. For a square global matrix, we define each process's diagonal portion to be its local rows and the corresponding columns (a square submatrix); each process's off-diagonal portion encompasses the remainder of the local matrix (a rectangular submatrix). The rank in the MPI communicator determines the absolute ordering of the blocks. That is, the process with rank 0 in the communicator given to `MatCreateAIJ()` contains the top rows of the matrix; the ith process in that communicator contains the ith block of the matrix.

Preallocation of Memory for Parallel AIJ Sparse Matrices

As discussed above, preallocation of memory is critical for achieving good performance during matrix assembly, as this reduces the number of allocations and copies required. We present an example for three processes to indicate how this may be done for the `MATMPIAIJ` matrix format. Consider the 8 by 8 matrix, which is partitioned by default with three rows on the first process, three on the second and two on the third.

$$\left(\begin{array}{ccc|ccc|cc}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
\hline
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
\hline
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34
\end{array}\right)$$

The “diagonal” submatrix, `d`, on the first process is given by

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{pmatrix},$$

while the “off-diagonal” submatrix, `o`, matrix is given by

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 7 & 0 & 0 & 8 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{pmatrix}.$$

For the first process one could set `d_nz` to 2 (since each row has 2 nonzeros) or, alternatively, set `d_nnz` to $\{2, 2, 2\}$. The `o_nz` could be set to 2 since each row of the `o` matrix has 2 nonzeros, or `o_nnz` could be set to $\{2, 2, 2\}$.

For the second process the `d` submatrix is given by

$$\begin{pmatrix} 15 & 16 & 17 \\ 19 & 20 & 21 \\ 22 & 23 & 0 \end{pmatrix}.$$

Thus, one could set `d_nz` to 3, since the maximum number of nonzeros in each row is 3, or alternatively one could set `d_nnz` to $\{3, 3, 2\}$, thereby indicating that the first two rows will have 3 nonzeros while the third has 2. The corresponding `o` submatrix for the second process is

$$\begin{pmatrix} 13 & 0 & 14 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 \\ 0 & 0 & 0 & 24 & 0 \end{pmatrix}$$

so that one could set `o_nz` to 2 or `o_nnz` to {2,1,1}.

Note that the user never directly works with the `d` and `o` submatrices, except when preallocating storage space as indicated above. Also, the user need not preallocate exactly the correct amount of space; as long as a sufficiently close estimate is given, the high efficiency for matrix assembly will remain.

As described above, the option `-info` will print information about the success of preallocation during matrix assembly. For the `MATMPIAIJ` and `MATMPIBAIJ` formats, PETSc will also list the number of elements owned by on each process that were generated on a different process. For example, the statements

```
MatAssemblyBegin_MPIAIJ:Stash has 10 entries, uses 0 mallocs
MatAssemblyBegin_MPIAIJ:Stash has 3 entries, uses 0 mallocs
MatAssemblyBegin_MPIAIJ:Stash has 5 entries, uses 0 mallocs
```

indicate that very few values have been generated on different processes. On the other hand, the statements

```
MatAssemblyBegin_MPIAIJ:Stash has 100000 entries, uses 100 mallocs
MatAssemblyBegin_MPIAIJ:Stash has 77777 entries, uses 70 mallocs
```

indicate that many values have been generated on the "wrong" processes. This situation can be very inefficient, since the transfer of values to the "correct" process is generally expensive. By using the command `MatGetOwnershipRange()` in application codes, the user should be able to generate most entries on the owning process.

Note: It is fine to generate some entries on the "wrong" process. Often this can lead to cleaner, simpler, less buggy codes. One should never make code overly complicated in order to generate all values locally. Rather, one should organize the code in such a way that *most* values are generated locally.

Limited-Memory Variable Metric (LMVM) Matrices

Variable metric methods, also known as quasi-Newton methods, are frequently used for root finding problems and approximate Jacobian matrices or their inverses via sequential nonlinear updates based on the secant condition. The limited-memory variants do not store the full explicit Jacobian, and instead compute forward products and inverse applications based on a fixed number of stored update vectors.

Table 2.2: PETSc LMVM matrix implementations.

Method	PETSc Type	Name	Property
"Good" Broyden [Gri12]	`MATLMVMBrdn`	`lmvmbrdn`	Square
"Bad" Broyden [Gri12]	`MATLMVMBadBrdn`	`lmvmbadbrdn`	Square
Symmetric Rank-1 [NW99]	`MATLMVMSR1`	`lmvmsr1`	Symmetric
Davidon-Fletcher-Powell (DFP) [NW99]	`MATLMVMDFP`	`lmvmdfp`	SPD
Broyden-Fletcher-Goldfarb-Shanno (BFGS) [NW99]	`MATLMVMBFGS`	`lmvmbfgs`	SPD
Restricted Broyden Family [EM17]	`MATLMVMSymBrdn`	`lmvmsymbrdn`	SPD
Restricted Broyden Family (full-memory diagonal)	`MATLMVMDiagBrdn`	`lmvmdiagbrdn`	SPD

PETSc implements seven different LMVM matrices listed in the table above. They can be created using the `MatCreate()` and `MatSetType()` workflow, and share a number of common interface functions. We will review the most important ones below:

- `MatLMVMAllocate(Mat B, Vec X, Vec F)` – Creates the internal data structures necessary to store nonlinear updates and compute forward/inverse applications. The `X` vector defines the solution space while the `F` defines the function space for the history of updates.
- `MatLMVMUpdate(MatB, Vec X, Vec F)` – Applies a nonlinear update to the approximate Jacobian such that $s_k = x_k - x_{k-1}$ and $y_k = f(x_k) - f(x_{k-1})$, where k is the index for the update.

- `MatLMVMReset(Mat B, PetscBool destructive)` – Flushes the accumulated nonlinear updates and resets the matrix to the initial state. If `destructive = PETSC_TRUE`, the reset also destroys the internal data structures and necessitates another allocation call before the matrix can be updated and used for products and solves.
- `MatLMVMSetJ0(Mat B, Mat J0)` – Defines the initial Jacobian to apply the updates to. If no initial Jacobian is provided, the updates are applied to an identity matrix.

LMVM matrices can be applied to vectors in forward mode via `MatMult()` or `MatMultAdd()`, and in inverse mode via `MatSolve()`. They also support `MatGetVecs()`, `MatDuplicate()` and `MatCopy()` operations. The maximum number of s_k and y_k update vectors stored can be changed via `-mat_lmvm_num_vecs` option.

Restricted Broyden Family, DFP and BFGS methods additionally implement special Jacobian initialization and scaling options available via `-mat_lmvm_scale_type <none,scalar,diagonal>`. We describe these choices below:

- `none` – Sets the initial Jacobian to be equal to the identity matrix. No extra computations are required when obtaining the search direction or updating the approximation. However, the number of function evaluations required to converge the Newton solution is typically much larger than what is required when using other initializations.
- `scalar` – Defines the initial Jacobian as a scalar multiple of the identity matrix. The scalar value σ is chosen by solving the one dimensional optimization problem

  $$\min_\sigma \|\sigma^\alpha Y - \sigma^{\alpha-1} S\|_F^2,$$

  where S and Y are the matrices whose columns contain a subset of update vectors s_k and y_k, and $\alpha \in [0, 1]$ is defined by the user via `-mat_lmvm_alpha` and has a different default value for each LMVM implementation (e.g.: default $\alpha = 1$ for BFGS produces the well-known $y_k^T s_k / y_k^T y_k$ scalar initialization). The number of updates to be used in the S and Y matrices is 1 by default (i.e.: the latest update only) and can be changed via `-mat_lmvm_scalar_hist`. This technique is inspired by Gilbert and Lemarechal [GL89].
- `diagonal` – Uses a full-memory restricted Broyden update formula to construct a diagonal matrix for the Jacobian initialization. Although the full-memory formula is utilized, the actual memory footprint is restricted to only the vector representing the diagonal and some additional work vectors used in its construction. The diagonal terms are also re-scaled with every update as suggested in [GL89]. This initialization requires the most computational effort of the available choices but typically results in a significant reduction in the number of function evaluations taken to compute a solution.

Note that the user-provided initial Jacobian via `MatLMVMSetJ0()` overrides and disables all built-in initialization methods.

Dense Matrices

PETSc provides both sequential and parallel dense matrix formats, where each process stores its entries in a column-major array in the usual Fortran style. To create a sequential, dense PETSc matrix, `A` of dimensions `m` by `n`, the user should call

```
MatCreateSeqDense(PETSC_COMM_SELF,PetscInt m,PetscInt n,PetscScalar *data,Mat *A);
```

The variable `data` enables the user to optionally provide the location of the data for matrix storage (intended for Fortran users who wish to allocate their own storage space). Most users should merely set `data` to `NULL` for PETSc to control matrix memory allocation. To create a parallel, dense matrix, `A`, the user should call

```
MatCreateDense(MPI_Comm comm,PetscInt m,PetscInt n,PetscInt M,PetscInt N,PetscScalar␣
↪*data,Mat *A)
```

The arguments `m`, `n`, `M`, and `N`, indicate the number of local rows and columns and the number of global rows and columns, respectively. Either the local or global parameters can be replaced with `PETSC_DECIDE`, so that PETSc will determine them. The matrix is stored with a fixed number of rows on each process, given by `m`, or determined by PETSc if `m` is `PETSC_DECIDE`.

PETSc does not provide parallel dense direct solvers, instead interfacing to external packages that provide these solvers. Our focus is on sparse iterative solvers.

Block Matrices

Block matrices arise when coupling variables with different meaning, especially when solving problems with constraints (e.g. incompressible flow) and "multi-physics" problems. Usually the number of blocks is small and each block is partitioned in parallel. We illustrate for a 3×3 system with components labeled a, b, c. With some numbering of unknowns, the matrix could be written as

$$\begin{pmatrix} A_{aa} & A_{ab} & A_{ac} \\ A_{ba} & A_{bb} & A_{bc} \\ A_{ca} & A_{cb} & A_{cc} \end{pmatrix}.$$

There are two fundamentally different ways that this matrix could be stored, as a single assembled sparse matrix where entries from all blocks are merged together ("monolithic"), or as separate assembled matrices for each block ("nested"). These formats have different performance characteristics depending on the operation being performed. In particular, many preconditioners require a monolithic format, but some that are very effective for solving block systems (see *Solving Block Matrices*) are more efficient when a nested format is used. In order to stay flexible, we would like to be able to use the same code to assemble block matrices in both monolithic and nested formats. Additionally, for software maintainability and testing, especially in a multi-physics context where different groups might be responsible for assembling each of the blocks, it is desirable to be able to use exactly the same code to assemble a single block independently as to assemble it as part of a larger system. To do this, we introduce the four spaces shown in Fig. 2.3.

- The monolithic global space is the space in which the Krylov and Newton solvers operate, with collective semantics across the entire block system.
- The split global space splits the blocks apart, but each split still has collective semantics.
- The split local space adds ghost points and separates the blocks. Operations in this space can be performed with no parallel communication. This is often the most natural, and certainly the most powerful, space for matrix assembly code.
- The monolithic local space can be thought of as adding ghost points to the monolithic global space, but it is often more natural to use it simply as a concatenation of split local spaces on each process. It is not common to explicitly manipulate vectors or matrices in this space (at least not during assembly), but it is a useful for declaring which part of a matrix is being assembled.

The key to format-independent assembly is the function

```
MatGetLocalSubMatrix(Mat A,IS isrow,IS iscol,Mat *submat);
```

which provides a "view" `submat` into a matrix `A` that operates in the monolithic global space. The `submat` transforms from the split local space defined by `iscol` to the split local space defined by `isrow`. The index sets specify the parts of the monolithic local space that `submat` should operate in. If a nested matrix format is used, then `MatGetLocalSubMatrix()` finds the nested block and returns it without making any copies. In this case, `submat` is fully functional and has a parallel communicator. If a monolithic matrix format is used, then `MatGetLocalSubMatrix()` returns a proxy matrix on `PETSC_COMM_SELF`

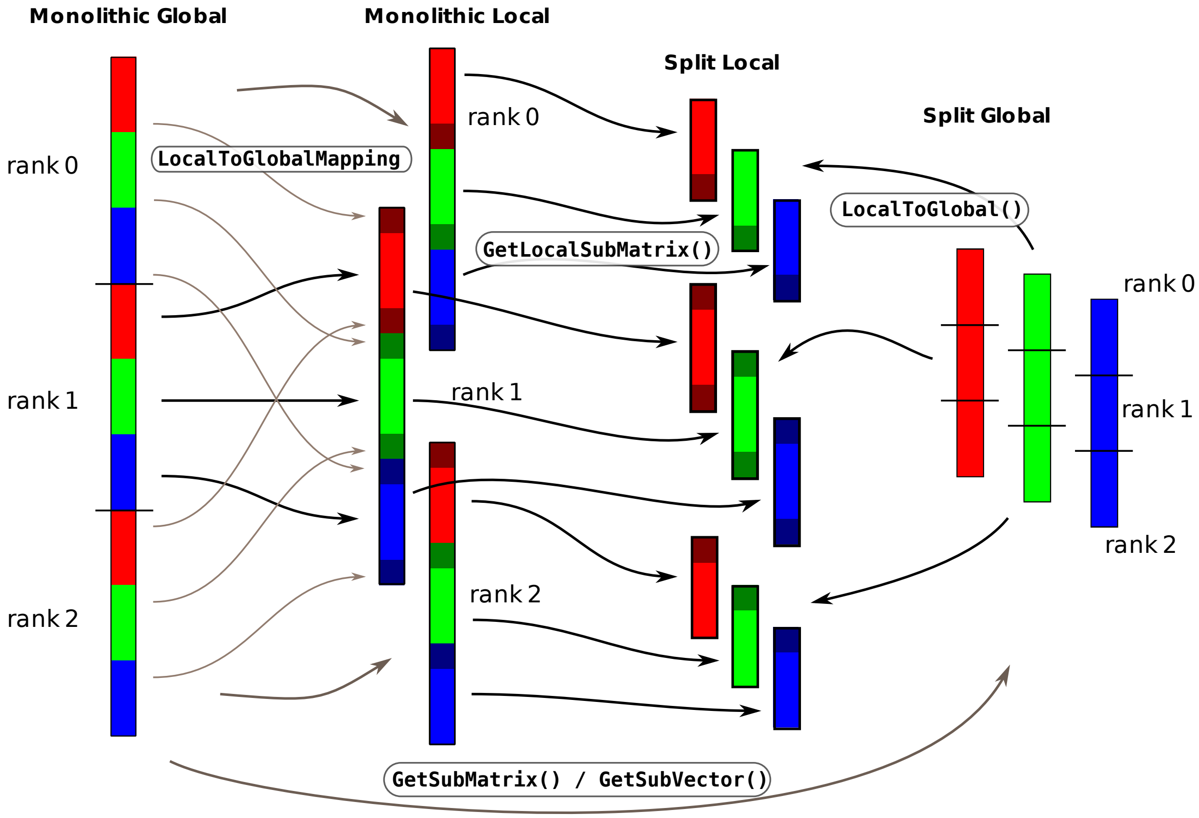

Monolithic Global
Monolithic Local
Split Local
Split Global
rank 0
rank 1
rank 2
LocalToGlobalMapping
GetLocalSubMatrix()
LocalToGlobal()
rank 0
rank 1
rank 2
rank 0
rank 1
rank 2
GetSubMatrix() / GetSubVector()

Fig. 2.3: The relationship between spaces used for coupled assembly.

that does not provide values or implement `MatMult()`, but does implement `MatSetValuesLocal()` and, if `isrow,iscol` have a constant block size, `MatSetValuesBlockedLocal()`. Note that although `submat` may not be a fully functional matrix and the caller does not even know a priori which communicator it will reside on, it always implements the local assembly functions (which are not collective). The index sets `isrow,iscol` can be obtained using `DMCompositeGetLocalISs()` if `DMComposite` is being used. DMComposite can also be used to create matrices, in which case the MATNEST format can be specified using `-prefix_dm_mat_type nest` and MATAIJ can be specified using `-prefix_dm_mat_type aij`. See SNES Tutorail ex28 for a simple example using this interface.

2.2.2 Basic Matrix Operations

Table *2.2* summarizes basic PETSc matrix operations. We briefly discuss a few of these routines in more detail below.

The parallel matrix can multiply a vector with `n` local entries, returning a vector with `m` local entries. That is, to form the product

```
MatMult(Mat A,Vec x,Vec y);
```

the vectors `x` and `y` should be generated with

```
VecCreateMPI(MPI_Comm comm,n,N,&x);
VecCreateMPI(MPI_Comm comm,m,M,&y);
```

By default, if the user lets PETSc decide the number of components to be stored locally (by passing in `PETSC_DECIDE` as the second argument to `VecCreateMPI()` or using `VecCreate()`), vectors and matrices of the same dimension are automatically compatible for parallel matrix-vector operations.

Along with the matrix-vector multiplication routine, there is a version for the transpose of the matrix,

```
MatMultTranspose(Mat A,Vec x,Vec y);
```

There are also versions that add the result to another vector:

```
MatMultAdd(Mat A,Vec x,Vec y,Vec w);
MatMultTransposeAdd(Mat A,Vec x,Vec y,Vec w);
```

These routines, respectively, produce $w = A * x + y$ and $w = A^T * x + y$. In C it is legal for the vectors `y` and `w` to be identical. In Fortran, this situation is forbidden by the language standard, but we allow it anyway.

One can print a matrix (sequential or parallel) to the screen with the command

```
MatView(Mat mat,PETSC_VIEWER_STDOUT_WORLD);
```

Other viewers can be used as well. For instance, one can draw the nonzero structure of the matrix into the default X-window with the command

```
MatView(Mat mat,PETSC_VIEWER_DRAW_WORLD);
```

Also one can use

```
MatView(Mat mat,PetscViewer viewer);
```

where `viewer` was obtained with `PetscViewerDrawOpen()`. Additional viewers and options are given in the `MatView()` man page and *Viewers: Looking at PETSc Objects*.

Table 2.3: PETSc Matrix Operations

Function Name	Operation
`MatAXPY(Mat Y, PetscScalar a, Mat X, MatStructure s);`	$Y = Y + a * X$
`MatMult(Mat A,Vec x, Vec y);`	$y = A * x$
`MatMultAdd(Mat A,Vec x, Vec y,Vec z);`	$z = y + A * x$
`MatMultTranspose(Mat A,Vec x, Vec y);`	$y = A^T * x$
`MatMultTransposeAdd(Mat A, Vec x, Vec y, Vec z);`	$z = y + A^T * x$
`MatNorm(Mat A,NormType type, PetscReal *r);`	$r = A_{type}$
`MatDiagonalScale(Mat A,Vec l,Vec r);`	$A = \text{diag}(l) * A * \text{diag}(r)$
`MatScale(Mat A,PetscScalar a);`	$A = a * A$
`MatConvert(Mat A, MatType type, Mat *B);`	$B = A$
`MatCopy(Mat A, Mat B, MatStructure s);`	$B = A$
`MatGetDiagonal(Mat A, Vec x);`	$x = \text{diag}(A)$
`MatTranspose(Mat A, MatReuse, Mat* B);`	$B = A^T$
`MatZeroEntries(Mat A);`	$A = 0$
`MatShift(Mat Y, PetscScalar a);`	$Y = Y + a * I$

The `NormType` argument to `MatNorm()` is one of `NORM_1`, `NORM_INFINITY`, and `NORM_FROBENIUS`.

2.2.3 Matrix-Free Matrices

Some people like to use matrix-free methods, which do not require explicit storage of the matrix, for the numerical solution of partial differential equations. To support matrix-free methods in PETSc, one can use the following command to create a `Mat` structure without ever actually generating the matrix:

```
MatCreateShell(MPI_Comm comm,PetscInt m,PetscInt n,PetscInt M,PetscInt N,void *ctx,
↪Mat *mat);
```

Here `M` and `N` are the global matrix dimensions (rows and columns), `m` and `n` are the local matrix dimensions, and `ctx` is a pointer to data needed by any user-defined shell matrix operations; the manual page has additional details about these parameters. Most matrix-free algorithms require only the application of the linear operator to a vector. To provide this action, the user must write a routine with the calling sequence

```
UserMult(Mat mat,Vec x,Vec y);
```

and then associate it with the matrix, `mat`, by using the command

```
MatShellSetOperation(Mat mat,MatOperation MATOP_MULT, (void(*)(void)) PetscErrorCode␣
↪(*UserMult)(Mat,Vec,Vec));
```

Here `MATOP_MULT` is the name of the operation for matrix-vector multiplication. Within each user-defined routine (such as `UserMult()`), the user should call `MatShellGetContext()` to obtain the user-defined context, `ctx`, that was set by `MatCreateShell()`. This shell matrix can be used with the iterative linear equation solvers discussed in the following chapters.

The routine `MatShellSetOperation()` can be used to set any other matrix operations as well. The file `$PETSC_DIR/include/petscmat.h` (source). provides a complete list of matrix operations, which have the form `MATOP_<OPERATION>`, where `<OPERATION>` is the name (in all capital letters) of the user interface routine (for example, `MatMult()` → `MATOP_MULT`). All user-provided functions have the same calling sequence as the usual matrix interface routines, since the user-defined functions are intended to be accessed through the same interface, e.g., `MatMult(Mat,Vec,Vec)` → `UserMult(Mat,Vec,Vec)`. The final argument for `MatShellSetOperation()` needs to be cast to a `void *`, since the final argument could (depending on the MatOperation) be a variety of different functions.

Note that `MatShellSetOperation()` can also be used as a "backdoor" means of introducing user-defined changes in matrix operations for other storage formats (for example, to override the default LU factorization routine supplied within PETSc for the `MATSEQAIJ` format). However, we urge anyone who introduces such changes to use caution, since it would be very easy to accidentally create a bug in the new routine that could affect other routines as well.

See also *Matrix-Free Methods* for details on one set of helpful utilities for using the matrix-free approach for nonlinear solvers.

2.2.4 Other Matrix Operations

In many iterative calculations (for instance, in a nonlinear equations solver), it is important for efficiency purposes to reuse the nonzero structure of a matrix, rather than determining it anew every time the matrix is generated. To retain a given matrix but reinitialize its contents, one can employ

```
MatZeroEntries(Mat A);
```

This routine will zero the matrix entries in the data structure but keep all the data that indicates where the nonzeros are located. In this way a new matrix assembly will be much less expensive, since no memory allocations or copies will be needed. Of course, one can also explicitly set selected matrix elements to zero by calling `MatSetValues()`.

By default, if new entries are made in locations where no nonzeros previously existed, space will be allocated for the new entries. To prevent the allocation of additional memory and simply discard those new entries, one can use the option

```
MatSetOption(Mat A,MAT_NEW_NONZERO_LOCATIONS,PETSC_FALSE);
```

Once the matrix has been assembled, one can factor it numerically without repeating the ordering or the symbolic factorization. This option can save some computational time, although it does require that the factorization is not done in-place.

In the numerical solution of elliptic partial differential equations, it can be cumbersome to deal with Dirichlet boundary conditions. In particular, one would like to assemble the matrix without regard to boundary conditions and then at the end apply the Dirichlet boundary conditions. In numerical analysis classes this process is usually presented as moving the known boundary conditions to the right-hand side and then solving a smaller linear system for the interior unknowns. Unfortunately, implementing this requires extracting a large submatrix from the original matrix and creating its corresponding data structures. This process can be expensive in terms of both time and memory.

One simple way to deal with this difficulty is to replace those rows in the matrix associated with known boundary conditions, by rows of the identity matrix (or some scaling of it). This action can be done with the command

```
MatZeroRows(Mat A,PetscInt numRows,PetscInt rows[],PetscScalar diag_value,Vec x,Vec␣
↪b),
```

or equivalently,

```
MatZeroRowsIS(Mat A,IS rows,PetscScalar diag_value,Vec x,Vec b);
```

For sparse matrices this removes the data structures for certain rows of the matrix. If the pointer `diag_value` is `NULL`, it even removes the diagonal entry. If the pointer is not null, it uses that given value at the pointer location in the diagonal entry of the eliminated rows.

One nice feature of this approach is that when solving a nonlinear problem such that at each iteration the Dirichlet boundary conditions are in the same positions and the matrix retains the same nonzero structure, the user can call `MatZeroRows()` in the first iteration. Then, before generating the matrix in the second iteration the user should call

```
MatSetOption(Mat A,MAT_NEW_NONZERO_LOCATIONS,PETSC_FALSE);
```

From that point, no new values will be inserted into those (boundary) rows of the matrix.

The functions `MatZeroRowsLocal()` and `MatZeroRowsLocalIS()` can also be used if for each process one provides the Dirichlet locations in the local numbering of the matrix. A drawback of `MatZeroRows()` is that it destroys the symmetry of a matrix. Thus one can use

```
MatZeroRowsColumns(Mat A,PetscInt numRows,PetscInt rows[],PetscScalar diag_value,Vec␣
↪x,Vec b),
```

or equivalently,

```
MatZeroRowsColumnsIS(Mat A,IS rows,PetscScalar diag_value,Vec x,Vec b);
```

Note that with all of these for a given assembled matrix it can be only called once to update the x and b vector. It cannot be used if one wishes to solve multiple right hand side problems for the same matrix since the matrix entries needed for updating the b vector are removed in its first use.

Once the zeroed rows are removed the new matrix has possibly many rows with only a diagonal entry affecting the parallel load balancing. The `PCREDISTRIBUTE` preconditioner removes all the zeroed rows (and associated columns and adjusts the right hand side based on the removed columns) and then rebalances the resulting rows of smaller matrix across the processes. Thus one can use `MatZeroRows()` to set the Dirichlet points and then solve with the preconditioner `PCREDISTRIBUTE`. Note if the original matrix was symmetric the smaller solved matrix will also be symmetric.

Another matrix routine of interest is

```
MatConvert(Mat mat,MatType newtype,Mat *M)
```

which converts the matrix `mat` to new matrix, `M`, that has either the same or different format. Set `newtype` to `MATSAME` to copy the matrix, keeping the same matrix format. See `$PETSC_DIR/include/petscmat.h` (source) for other available matrix types; standard ones are `MATSEQDENSE`, `MATSEQAIJ`, `MATMPIAIJ`, `MATSEQBAIJ` and `MATMPIBAIJ`.

In certain applications it may be necessary for application codes to directly access elements of a matrix. This may be done by using the the command (for local rows only)

```
MatGetRow(Mat A,PetscInt row, PetscInt *ncols,const PetscInt (*cols)[],const␣
↪PetscScalar (*vals)[]);
```

The argument `ncols` returns the number of nonzeros in that row, while `cols` and `vals` returns the column indices (with indices starting at zero) and values in the row. If only the column indices are needed (and not the corresponding matrix elements), one can use `NULL` for the `vals` argument. Similarly, one can use `NULL` for the `cols` argument. The user can only examine the values extracted with `MatGetRow()`; the values *cannot* be altered. To change the matrix entries, one must use `MatSetValues()`.

Once the user has finished using a row, he or she *must* call

```
MatRestoreRow(Mat A,PetscInt row,PetscInt *ncols,PetscInt **cols,PetscScalar **vals);
```

to free any space that was allocated during the call to `MatGetRow()`.

2.2.5 Partitioning

For almost all unstructured grid computation, the distribution of portions of the grid across the process's work load and memory can have a very large impact on performance. In most PDE calculations the grid partitioning and distribution across the processes can (and should) be done in a "pre-processing" step before the numerical computations. However, this does not mean it need be done in a separate, sequential program; rather, it should be done before one sets up the parallel grid data structures in the actual program. PETSc provides an interface to the ParMETIS (developed by George Karypis; see the PETSc installation instructions. for directions on installing PETSc to use ParMETIS) to allow the partitioning to be done in parallel. PETSc does not currently provide directly support for dynamic repartitioning, load balancing by migrating matrix entries between processes, etc. For problems that require mesh refinement, PETSc uses the "rebuild the data structure" approach, as opposed to the "maintain dynamic data structures that support the insertion/deletion of additional vector and matrix rows and columns entries" approach.

Partitioning in PETSc is organized around the `MatPartitioning` object. One first creates a parallel matrix that contains the connectivity information about the grid (or other graph-type object) that is to be partitioned. This is done with the command

```
MatCreateMPIAdj(MPI_Comm comm,int mlocal,PetscInt n,const PetscInt ia[],const␣
↪PetscInt ja[],PetscInt *weights,Mat *Adj);
```

The argument `mlocal` indicates the number of rows of the graph being provided by the given process, `n` is the total number of columns; equal to the sum of all the `mlocal`. The arguments `ia` and `ja` are the row pointers and column pointers for the given rows; these are the usual format for parallel compressed sparse row storage, using indices starting at 0, *not* 1.

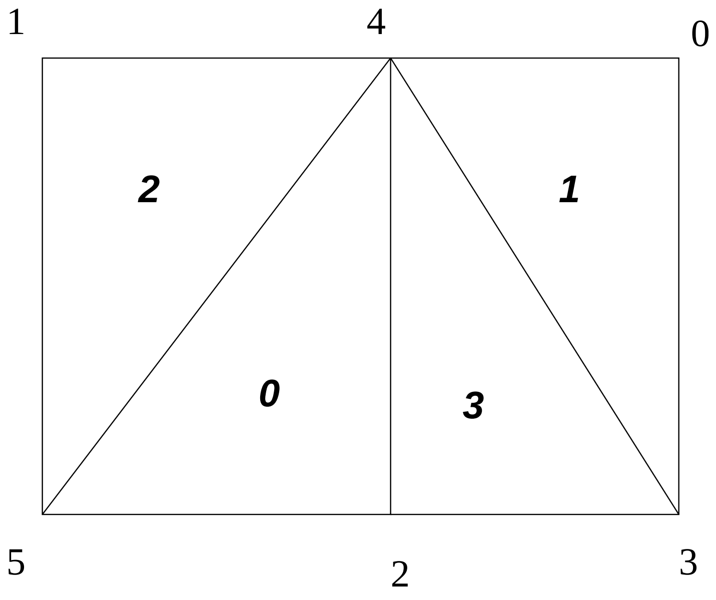


Fig. 2.4: Numbering on Simple Unstructured Grid

This, of course, assumes that one has already distributed the grid (graph) information among the processes. The details of this initial distribution is not important; it could be simply determined by assigning to the first process the first n_0 nodes from a file, the second process the next n_1 nodes, etc.

For example, we demonstrate the form of the `ia` and `ja` for a triangular grid where we

(1) partition by element (triangle)

- Process 0: `mlocal = 2`, `n = 4`, `ja ={2,3, 3}`, `ia = {0,2,3}`

- Process 1: `mlocal = 2`, `n = 4`, `ja ={0, 0,1}`, `ia ={0,1,3}`

Note that elements are not connected to themselves and we only indicate edge connections (in some contexts single vertex connections between elements may also be included). We use a space above to denote the transition between rows in the matrix.

and (2) partition by vertex.

- Process 0: `mlocal = 3`, `n = 6`, `ja ={3,4, 4,5, 3,4,5}`, `ia ={0, 2, 4, 7}`
- Process 1: `mlocal = 3`, `n = 6`, `ja ={0,2, 4, 0,1,2,3,5, 1,2,4}`, `ia ={0, 3, 8, 11}`.

Once the connectivity matrix has been created the following code will generate the renumbering required for the new partition

```
MatPartitioningCreate(MPI_Comm comm,MatPartitioning *part);
MatPartitioningSetAdjacency(MatPartitioning part,Mat Adj);
MatPartitioningSetFromOptions(MatPartitioning part);
MatPartitioningApply(MatPartitioning part,IS *is);
MatPartitioningDestroy(MatPartitioning *part);
MatDestroy(Mat *Adj);
ISPartitioningToNumbering(IS is,IS *isg);
```

The resulting `isg` contains for each local node the new global number of that node. The resulting `is` contains the new process number that each local node has been assigned to.

Now that a new numbering of the nodes has been determined, one must renumber all the nodes and migrate the grid information to the correct process. The command

```
AOCreateBasicIS(isg,NULL,&ao);
```

generates, see *Application Orderings*, an AO object that can be used in conjunction with the `is` and `isg` to move the relevant grid information to the correct process and renumber the nodes etc. In this context, the new ordering is the "application" ordering so `AOPetscToApplication()` converts old global indices to new global indices and `AOApplicationToPetsc()` converts new global indices back to old global indices.

PETSc does not currently provide tools that completely manage the migration and node renumbering, since it will be dependent on the particular data structure you use to store the grid information and the type of grid information that you need for your application. We do plan to include more support for this in the future, but designing the appropriate general user interface and providing a scalable implementation that can be used for a wide variety of different grids requires a great deal of time.

2.3 KSP: Linear System Solvers

The `KSP` object is the heart of PETSc, because it provides uniform and efficient access to all of the package's linear system solvers, including parallel and sequential, direct and iterative. `KSP` is intended for solving systems of the form

$$Ax = b, \tag{2.1}$$

where A denotes the matrix representation of a linear operator, b is the right-hand-side vector, and x is the solution vector. `KSP` uses the same calling sequence for both direct and iterative solution of a linear system. In addition, particular solution techniques and their associated options can be selected at runtime.

The combination of a Krylov subspace method and a preconditioner is at the center of most modern numerical codes for the iterative solution of linear systems. Many textbooks (e.g. [FGN92] [vdV03], or [Saa03]) provide an overview of the theory of such methods. The `KSP` package, discussed in *Krylov Methods*, provides many popular Krylov subspace iterative methods; the `PC` module, described in *Preconditioners*, includes a variety of preconditioners.

2.3.1 Using KSP

To solve a linear system with `KSP`, one must first create a solver context with the command

```
KSPCreate(MPI_Comm comm,KSP *ksp);
```

Here `comm` is the MPI communicator and `ksp` is the newly formed solver context. Before actually solving a linear system with `KSP`, the user must call the following routine to set the matrices associated with the linear system:

```
KSPSetOperators(KSP ksp,Mat Amat,Mat Pmat);
```

The argument `Amat`, representing the matrix that defines the linear system, is a symbolic placeholder for any kind of matrix or operator. In particular, `KSP` *does* support matrix-free methods. The routine `MatCreateShell()` in *Matrix-Free Matrices* provides further information regarding matrix-free methods. Typically, the matrix from which the preconditioner is to be constructed, `Pmat`, is the same as the matrix that defines the linear system, `Amat`; however, occasionally these matrices differ (for instance, when a preconditioning matrix is obtained from a lower order method than that employed to form the linear system matrix).

Much of the power of `KSP` can be accessed through the single routine

```
KSPSetFromOptions(KSP ksp);
```

This routine accepts the option `-help` as well as any of the `KSP` and `PC` options discussed below. To solve a linear system, one sets the rhs and solution vectors using and executes the command

```
KSPSolve(KSP ksp,Vec b,Vec x);
```

where `b` and `x` respectively denote the right-hand-side and solution vectors. On return, the iteration number at which the iterative process stopped can be obtained using

```
KSPGetIterationNumber(KSP ksp, PetscInt *its);
```

Note that this does not state that the method converged at this iteration: it can also have reached the maximum number of iterations, or have diverged.

Convergence Tests gives more details regarding convergence testing. Note that multiple linear solves can be performed by the same `KSP` context. Once the `KSP` context is no longer needed, it should be destroyed with the command

```
KSPDestroy(KSP *ksp);
```

The above procedure is sufficient for general use of the `KSP` package. One additional step is required for users who wish to customize certain preconditioners (e.g., see *Block Jacobi and Overlapping Additive Schwarz Preconditioners*) or to log certain performance data using the PETSc profiling facilities (as discussed in *Profiling*). In this case, the user can optionally explicitly call

```
KSPSetUp(KSP ksp);
```

before calling `KSPSolve()` to perform any setup required for the linear solvers. The explicit call of this routine enables the separate monitoring of any computations performed during the set up phase, such as incomplete factorization for the ILU preconditioner.

The default solver within `KSP` is restarted GMRES, preconditioned for the uniprocess case with ILU(0), and for the multiprocess case with the block Jacobi method (with one block per process, each of which is solved with ILU(0)). A variety of other solvers and options are also available. To allow application programmers to set any of the preconditioner or Krylov subspace options directly within the code, we provide routines that extract the `PC` and `KSP` contexts,

```
KSPGetPC(KSP ksp,PC *pc);
```

The application programmer can then directly call any of the `PC` or `KSP` routines to modify the corresponding default options.

To solve a linear system with a direct solver (currently supported by PETSc for sequential matrices, and by several external solvers through PETSc interfaces (see *Using External Linear Solvers*)) one may use the options `-ksp_type preonly -pc_type lu` (see below).

By default, if a direct solver is used, the factorization is *not* done in-place. This approach prevents the user from the unexpected surprise of having a corrupted matrix after a linear solve. The routine `PCFactorSetUseInPlace()`, discussed below, causes factorization to be done in-place.

2.3.2 Solving Successive Linear Systems

When solving multiple linear systems of the same size with the same method, several options are available. To solve successive linear systems having the *same* preconditioner matrix (i.e., the same data structure with exactly the same matrix elements) but different right-hand-side vectors, the user should simply call `KSPSolve()`, multiple times. The preconditioner setup operations (e.g., factorization for ILU) will be done during the first call to `KSPSolve()` only; such operations will *not* be repeated for successive solves.

To solve successive linear systems that have *different* preconditioner matrices (i.e., the matrix elements and/or the matrix data structure change), the user *must* call `KSPSetOperators()` and `KSPSolve()` for each solve. See *Using KSP* for a description of various flags for `KSPSetOperators()` that can save work for such cases.

2.3.3 Krylov Methods

The Krylov subspace methods accept a number of options, many of which are discussed below. First, to set the Krylov subspace method that is to be used, one calls the command

```
KSPSetType(KSP ksp,KSPType method);
```

The type can be one of `KSPRICHARDSON`, `KSPCHEBYSHEV`, `KSPCG`, `KSPGMRES`, `KSPTCQMR`, `KSPBCGS`, `KSPCGS`, `KSPTFQMR`, `KSPCR`, `KSPLSQR`, `KSPBICG`, `KSPPREONLY`. or others; see *KSP Objects* or the `KSPType` man page for more. The `KSP` method can also be set with the options database command `-ksp_type`, followed by one of the options `richardson`, `chebyshev`, `cg`, `gmres`, `tcqmr`, `bcgs`, `cgs`, `tfqmr`, `cr`, `lsqr`, `bicg`, `preonly.`, or others (see *KSP Objects* or the `KSPType` man page) There are method-specific options. For instance, for the Richardson, Chebyshev, and GMRES methods:

```
KSPRichardsonSetScale(KSP ksp,PetscReal scale);
KSPChebyshevSetEigenvalues(KSP ksp,PetscReal emax,PetscReal emin);
KSPGMRESSetRestart(KSP ksp,PetscInt max_steps);
```

The default parameter values are `damping_factor=1.0, emax=0.01, emin=100.0`, and `max_steps=30`. The GMRES restart and Richardson damping factor can also be set with the options `-ksp_gmres_restart <n>` and `-ksp_richardson_scale <factor>`.

The default technique for orthogonalization of the Hessenberg matrix in GMRES is the unmodified (classical) Gram-Schmidt method, which can be set with

```
KSPGMRESSetOrthogonalization(KSP ksp,KSPGMRESClassicalGramSchmidtOrthogonalization);
```

or the options database command `-ksp_gmres_classicalgramschmidt`. By default this will *not* use iterative refinement to improve the stability of the orthogonalization. This can be changed with the option

```
KSPGMRESSetCGSRefinementType(KSP ksp,KSPGMRESCGSRefinementType type)
```

or via the options database with

```
-ksp_gmres_cgs_refinement_type none,ifneeded,always
```

The values for `KSPGMRESCGSRefinementType()` are `KSP_GMRES_CGS_REFINEMENT_NONE`, `KSP_GMRES_CGS_REFINEMENT_IFNEEDED` and `KSP_GMRES_CGS_REFINEMENT_ALWAYS`.

One can also use modified Gram-Schmidt, by using the orthogonalization routine `KSPGMRESModifiedGramSchmidtOrthogonalization()` or by using the command line option `-ksp_gmres_modifiedgramschmidt`.

For the conjugate gradient method with complex numbers, there are two slightly different algorithms depending on whether the matrix is Hermitian symmetric or truly symmetric (the default is to assume that it is Hermitian symmetric). To indicate that it is symmetric, one uses the command

```
KSPCGSetType(KSP ksp,KSPCGType KSP_CG_SYMMETRIC);
```

Note that this option is not valid for all matrices.

The LSQR algorithm does not involve a preconditioner; any preconditioner set to work with the `KSP` object is ignored if `KSPLSQR` was selected.

By default, `KSP` assumes an initial guess of zero by zeroing the initial value for the solution vector that is given; this zeroing is done at the call to `KSPSolve()`. To use a nonzero initial guess, the user *must* call

```
KSPSetInitialGuessNonzero(KSP ksp,PetscBool flg);
```

Preconditioning within KSP

Since the rate of convergence of Krylov projection methods for a particular linear system is strongly dependent on its spectrum, preconditioning is typically used to alter the spectrum and hence accelerate the convergence rate of iterative techniques. Preconditioning can be applied to the system *(1)* by

$$(M_L^{-1} A M_R^{-1}) \, (M_R x) = M_L^{-1} b, \tag{2.2}$$

where M_L and M_R indicate preconditioning matrices (or, matrices from which the preconditioner is to be constructed). If $M_L = I$ in *(2)*, right preconditioning results, and the residual of *(1)*,

$$r \equiv b - Ax = b - A M_R^{-1} \, M_R x,$$

is preserved. In contrast, the residual is altered for left ($M_R = I$) and symmetric preconditioning, as given by

$$r_L \equiv M_L^{-1} b - M_L^{-1} A x = M_L^{-1} r.$$

By default, most KSP implementations use left preconditioning. Some more naturally use other options, though. For instance, `KSPQCG` defaults to use symmetric preconditioning and `KSPFGMRES` uses right preconditioning by default. Right preconditioning can be activated for some methods by using the options database command `-ksp_pc_side right` or calling the routine

```
KSPSetPCSide(KSP ksp,PCSide PC_RIGHT);
```

Attempting to use right preconditioning for a method that does not currently support it results in an error message of the form

```
KSPSetUp_Richardson:No right preconditioning for KSPRICHARDSON
```

We summarize the defaults for the residuals used in KSP convergence monitoring within *KSP Objects*. Details regarding specific convergence tests and monitoring routines are presented in the following sections. The preconditioned residual is used by default for convergence testing of all left-preconditioned `KSP` methods. For the conjugate gradient, Richardson, and Chebyshev methods the true residual can be used by the options database command `ksp_norm_type unpreconditioned` or by calling the routine

```
KSPSetNormType(KSP ksp,KSP_NORM_UNPRECONDITIONED);
```

Table 2.4: KSP Objects

Method	KSPType	Options Database Name
Richardson	`KSPRICHARDSON`	`richardson`
Chebyshev	`KSPCHEBYSHEV`	`chebyshev`
Conjugate Gradient [HS52]	`KSPCG`	`cg`
Pipelined Conjugate Gradients [GV14]	`KSPPIPECG`	`pipecg`
Pipelined Conjugate Gradients (Gropp)	`KSPGROPPCG`	`groppcg`
Pipelined Conjugate Gradients with Residual Replacement	`KSPPIPECGRR`	`pipecgrr`
Conjugate Gradients for the Normal Equations	`KSPCGNE`	`cgne`
Flexible Conjugate Gradients [Not00]	`KSPFCG`	`fcg`
Pipelined, Flexible Conjugate Gradients [SSM16]	`KSPPIPEFCG`	`pipefcg`
Conjugate Gradients for Least Squares	`KSPCGLS`	`cgls`
Conjugate Gradients with Constraint (1)	`KSPNASH`	`nash`
Conjugate Gradients with Constraint (2)	`KSPSTCG`	`stcg`
Conjugate Gradients with Constraint (3)	`KSPGLTR`	`gltr`
Conjugate Gradients with Constraint (4)	`KSPQCG`	`qcg`
BiConjugate Gradient	`KSPBICG`	`bicg`
BiCGSTAB [vandVorst92]	`KSPBCGS`	`bcgs`
Improved BiCGSTAB	`KSPIBCGS`	`ibcgs`
Flexible BiCGSTAB	`KSPFBCGS`	`fbcgs`
Flexible BiCGSTAB (variant)	`KSPFBCGSR`	`fbcgsr`
Enhanced BiCGSTAB(L)	`KSPBCGSL`	`bcgsl`
Minimal Residual Method [PS75]	`KSPMINRES`	`minres`
Generalized Minimal Residual [SS86]	`KSPGMRES`	`gmres`
Flexible Generalized Minimal Residual [Saa93]	`KSPFGMRES`	`fgmres`
Deflated Generalized Minimal Residual	`KSPDGMRES`	`dgmres`
Pipelined Generalized Minimal Residual [GAMV13]	`KSPPGMRES`	`pgmres`
Pipelined, Flexible Generalized Minimal Residual [SSM16]	`KSPPIPEFGMRES`	`pipefgmres`
Generalized Minimal Residual with Accelerated Restart	`KSPLGMRES`	`lgmres`
Conjugate Residual [EES83]	`KSPCR`	`cr`
Generalized Conjugate Residual	`KSPGCR`	`gcr`
Pipelined Conjugate Residual	`KSPPIPECR`	`pipecr`
Pipelined, Flexible Conjugate Residual [SSM16]	`KSPPIPEGCR`	`pipegcr`
FETI-DP	`KSPFETIDP`	`fetidp`
Conjugate Gradient Squared [Son89]	`KSPCGS`	`cgs`
Transpose-Free Quasi-Minimal Residual (1) [Fre93]	`KSPTFQMR`	`tfqmr`

Continued on next page

Table 2.4 – continued from previous page

Method	KSPType	Options Database Name
Transpose-Free Quasi-Minimal Residual (2)	`KSPTCQMR`	`tcqmr`
Least Squares Method	`KSPLSQR`	`lsqr`
Symmetric LQ Method [PS75]	`KSPSYMMLQ`	`symmlq`
TSIRM	`KSPTSIRM`	`tsirm`
Python Shell	`KSPPYTHON`	`python`
Shell for no `KSP` method	`KSPPREONLY`	`preonly`

Note: the bi-conjugate gradient method requires application of both the matrix and its transpose plus the preconditioner and its transpose. Currently not all matrices and preconditioners provide this support and thus the `KSPBICG` cannot always be used.

Note: PETSc implements the FETI-DP (Finite Element Tearing and Interconnecting Dual-Primal) method as an implementation of `KSP` since it recasts the original problem into a contstrained minimization one with Lagrange multipliers. The only matrix type supported is `MATIS`. Support for saddle point problems is provided. See the man page for `KSPFETIDP` for further details.

Convergence Tests

The default convergence test, `KSPConvergedDefault()`, is based on the l_2-norm of the residual. Convergence (or divergence) is decided by three quantities: the decrease of the residual norm relative to the norm of the right hand side, `rtol`, the absolute size of the residual norm, `atol`, and the relative increase in the residual, `dtol`. Convergence is detected at iteration k if

$$\|r_k\|_2 < \max(\text{rtol} * \|b\|_2, \text{atol}),$$

where $r_k = b - Ax_k$. Divergence is detected if

$$\|r_k\|_2 > \text{dtol} * \|b\|_2.$$

These parameters, as well as the maximum number of allowable iterations, can be set with the routine

```
KSPSetTolerances(KSP ksp,PetscReal rtol,PetscReal atol,PetscReal dtol,PetscInt␣
↪maxits);
```

The user can retain the default value of any of these parameters by specifying `PETSC_DEFAULT` as the corresponding tolerance; the defaults are `rtol=1e-5`, `atol=1e-50`, `dtol=1e5`, and `maxits=1e4`. These parameters can also be set from the options database with the commands `-ksp_rtol <rtol>`, `-ksp_atol <atol>`, `-ksp_divtol <dtol>`, and `-ksp_max_it <its>`.

In addition to providing an interface to a simple convergence test, `KSP` allows the application programmer the flexibility to provide customized convergence-testing routines. The user can specify a customized routine with the command

```
KSPSetConvergenceTest(KSP ksp,PetscErrorCode (*test)(KSP ksp,PetscInt it,PetscReal␣
↪rnorm, KSPConvergedReason *reason,void *ctx),void *ctx,PetscErrorCode␣
↪(*destroy)(void *ctx));
```

The final routine argument, `ctx`, is an optional context for private data for the user-defined convergence routine, `test`. Other `test` routine arguments are the iteration number, `it`, and the residual's l_2 norm, `rnorm`. The routine for detecting convergence, `test`, should set `reason` to positive for convergence, 0 for no convergence, and negative for failure to converge. A full list of possible values for `KSPConvergedReason` is given in `include/petscksp.h`. You can use `KSPGetConvergedReason()` after `KSPSolve()` to see why convergence/divergence was detected.

Convergence Monitoring

By default, the Krylov solvers run silently without displaying information about the iterations. The user can indicate that the norms of the residuals should be displayed by using `-ksp_monitor` within the options database. To display the residual norms in a graphical window (running under X Windows), one should use `-ksp_monitor_lg_residualnorm [x,y,w,h]`, where either all or none of the options must be specified. Application programmers can also provide their own routines to perform the monitoring by using the command

```
KSPMonitorSet(KSP ksp,PetscErrorCode (*mon)(KSP ksp,PetscInt it,PetscReal rnorm,void␣
↪*ctx),void *ctx,PetscErrorCode (*mondestroy)(void**));
```

The final routine argument, `ctx`, is an optional context for private data for the user-defined monitoring routine, `mon`. Other `mon` routine arguments are the iteration number (`it`) and the residual's l_2 norm (`rnorm`). A helpful routine within user-defined monitors is `PetscObjectGetComm((PetscObject)ksp,MPI_Comm *comm)`, which returns in `comm` the MPI communicator for the `KSP` context. See *Writing PETSc Programs* for more discussion of the use of MPI communicators within PETSc.

Several monitoring routines are supplied with PETSc, including

```
KSPMonitorDefault(KSP,PetscInt,PetscReal, void *);
KSPMonitorSingularValue(KSP,PetscInt,PetscReal,void *);
KSPMonitorTrueResidualNorm(KSP,PetscInt,PetscReal, void *);
```

The default monitor simply prints an estimate of the l_2-norm of the residual at each iteration. The routine `KSPMonitorSingularValue()` is appropriate only for use with the conjugate gradient method or GMRES, since it prints estimates of the extreme singular values of the preconditioned operator at each iteration. Since `KSPMonitorTrueResidualNorm()` prints the true residual at each iteration by actually computing the residual using the formula $r = b - Ax$, the routine is slow and should be used only for testing or convergence studies, not for timing. These monitors may be accessed with the command line options `-ksp_monitor`, `-ksp_monitor_singular_value`, and `-ksp_monitor_true_residual`.

To employ the default graphical monitor, one should use the commands

```
PetscDrawLG lg;
KSPMonitorLGResidualNormCreate(MPI_Comm comm,char *display,char *title,PetscInt x,
↪PetscInt y,PetscInt w,PetscInt h,PetscDrawLG *lg);
KSPMonitorSet(KSP ksp,KSPMonitorLGResidualNorm,lg,0);
```

When no longer needed, the line graph should be destroyed with the command

```
PetscDrawLGDestroy(PetscDrawLG *lg);
```

The user can change aspects of the graphs with the `PetscDrawLG*()` and `PetscDrawAxis*()` routines. One can also access this functionality from the options database with the command `-ksp_monitor_lg_residualnorm [x,y,w,h].` , where `x, y, w, h` are the optional location and size of the window.

One can cancel hardwired monitoring routines for KSP at runtime with `-ksp_monitor_cancel`.

Unless the Krylov method converges so that the residual norm is small, say 10^{-10}, many of the final digits printed with the `-ksp_monitor` option are meaningless. Worse, they are different on different machines; due to different round-off rules used by, say, the IBM RS6000 and the Sun SPARC. This makes testing between different machines difficult. The option `-ksp_monitor_short` causes PETSc to print fewer of the digits of the residual norm as it gets smaller; thus on most of the machines it will always print the same numbers making cross system testing easier.

Understanding the Operator's Spectrum

Since the convergence of Krylov subspace methods depends strongly on the spectrum (eigenvalues) of the preconditioned operator, PETSc has specific routines for eigenvalue approximation via the Arnoldi or Lanczos iteration. First, before the linear solve one must call

```
KSPSetComputeEigenvalues(KSP ksp,PETSC_TRUE);
```

Then after the `KSP` solve one calls

```
KSPComputeEigenvalues(KSP ksp,PetscInt n,PetscReal *realpart,PetscReal *complexpart,
↪PetscInt *neig);
```

Here, `n` is the size of the two arrays and the eigenvalues are inserted into those two arrays. `neig` is the number of eigenvalues computed; this number depends on the size of the Krylov space generated during the linear system solution, for GMRES it is never larger than the restart parameter. There is an additional routine

```
KSPComputeEigenvaluesExplicitly(KSP ksp, PetscInt n,PetscReal *realpart,PetscReal␣
↪*complexpart);
```

that is useful only for very small problems. It explicitly computes the full representation of the preconditioned operator and calls LAPACK to compute its eigenvalues. It should be only used for matrices of size up to a couple hundred. The `PetscDrawSP*()` routines are very useful for drawing scatter plots of the eigenvalues.

The eigenvalues may also be computed and displayed graphically with the options data base commands `-ksp_view_eigenvalues draw` and `-ksp_view_eigenvalues_explicitly draw`. Or they can be dumped to the screen in ASCII text via `-ksp_view_eigenvalues` and `-ksp_view_eigenvalues_explicitly`.

Other KSP Options

To obtain the solution vector and right hand side from a `KSP` context, one uses

```
KSPGetSolution(KSP ksp,Vec *x);
KSPGetRhs(KSP ksp,Vec *rhs);
```

During the iterative process the solution may not yet have been calculated or it may be stored in a different location. To access the approximate solution during the iterative process, one uses the command

```
KSPBuildSolution(KSP ksp,Vec w,Vec *v);
```

where the solution is returned in `v`. The user can optionally provide a vector in `w` as the location to store the vector; however, if `w` is `NULL`, space allocated by PETSc in the `KSP` context is used. One should not destroy this vector. For certain `KSP` methods, (e.g., GMRES), the construction of the solution is expensive, while for many others it doesn't evenrequire a vector copy.

Access to the residual is done in a similar way with the command

```
KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v);
```

Again, for GMRES and certain other methods this is an expensive operation.

2.3.4 Preconditioners

As discussed in *Preconditioning within KSP*, Krylov subspace methods are typically used in conjunction with a preconditioner. To employ a particular preconditioning method, the user can either select it from the options database using input of the form `-pc_type <methodname>` or set the method with the command

```
PCSetType(PC pc,PCType method);
```

In *PETSc Preconditioners (partial list)* we summarize the basic preconditioning methods supported in PETSc. See the `PCType` manual page for a complete list. The `PCSHELL` preconditioner uses a specific, application-provided preconditioner. The direct preconditioner, `PCLU` , is, in fact, a direct solver for the linear system that uses LU factorization. `PCLU` is included as a preconditioner so that PETSc has a consistent interface among direct and iterative linear solvers.

Table 2.5: PETSc Preconditioners (partial list)

Method	PCType	Options Database Name
Jacobi	`PCJACOBI`	`jacobi`
Block Jacobi	`PCBJACOBI`	`bjacobi`
SOR (and SSOR)	`PCSOR`	`sor`
SOR with Eisenstat trick	`PCEISENSTAT`	`eisenstat`
Incomplete Cholesky	`PCICC`	`icc`
Incomplete LU	`PCILU`	`ilu`
Additive Schwarz	`PCASM`	`asm`
Generalized Additive Schwarz	`PCGASM`	`gasm`
Algebraic Multigrid	`PCGAMG`	`gamg`
Balancing Domain Decomposition by Constraints	`PCBDDC`	`bddc`
Linear solver	`PCKSP`	`ksp`
Combination of preconditioners	`PCCOMPOSITE`	`composite`
LU	`PCLU`	`lu`
Cholesky	`PCCHOLESKY`	`cholesky`
No preconditioning	`PCNONE`	`none`
Shell for user-defined `PC`	`PCSHELL`	`shell`

Each preconditioner may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. Such routine names and commands are all of the form `PC<TYPE><Option>` and `-pc_<type>_<option> [value]`. A complete list can be found by consulting the `PCType` manual page; we discuss just a few in the sections below.

ILU and ICC Preconditioners

Some of the options for ILU preconditioner are

```
PCFactorSetLevels(PC pc,PetscInt levels);
PCFactorSetReuseOrdering(PC pc,PetscBool  flag);
PCFactorSetDropTolerance(PC pc,PetscReal dt,PetscReal dtcol,PetscInt dtcount);
PCFactorSetReuseFill(PC pc,PetscBool  flag);
PCFactorSetUseInPlace(PC pc,PetscBool flg);
PCFactorSetAllowDiagonalFill(PC pc,PetscBool flg);
```

When repeatedly solving linear systems with the same `KSP` context, one can reuse some information computed during the first linear solve. In particular, `PCFactorSetReuseOrdering()` causes the ordering (for example, set with `-pc_factor_mat_ordering_type order`) computed in the first factorization to be reused for later factorizations. `PCFactorSetUseInPlace()` is often used with `PCASM` or `PCBJACOBI` when zero fill is used, since it reuses the matrix space to store the incomplete factorization it saves memory and copying time. Note that in-place factorization is not appropriate with any ordering besides natural and cannot be used with the drop tolerance factorization. These options may be set in the database with

- `-pc_factor_levels <levels>`
- `-pc_factor_reuse_ordering`
- `-pc_factor_reuse_fill`
- `-pc_factor_in_place`
- `-pc_factor_nonzeros_along_diagonal`
- `-pc_factor_diagonal_fill`

See *Memory Allocation for Sparse Matrix Factorization* for information on preallocation of memory for anticipated fill during factorization. By alleviating the considerable overhead for dynamic memory allocation, such tuning can significantly enhance performance.

PETSc supports incomplete factorization preconditioners for several matrix types for sequential matrices (for example `MATSEQAIJ`, `MATSEQBAIJ`, and `MATSEQSBAIJ`).

SOR and SSOR Preconditioners

PETSc only provides only a sequential SOR preconditioner; it can only be used with sequential matrices or as the subblock preconditioner when using block Jacobi or ASM preconditioning (see below).

The options for SOR preconditioning with `PCSOR` are

```
PCSORSetOmega(PC pc,PetscReal omega);
PCSORSetIterations(PC pc,PetscInt its,PetscInt lits);
PCSORSetSymmetric(PC pc,MatSORType type);
```

The first of these commands sets the relaxation factor for successive over (under) relaxation. The second command sets the number of inner iterations `its` and local iterations `lits` (the number of smoothing sweeps on a process before doing a ghost point update from the other processes) to use between steps of the Krylov space method. The total number of SOR sweeps is given by `its*lits`. The third command sets the kind of SOR sweep, where the argument `type` can be one of `SOR_FORWARD_SWEEP`, `SOR_BACKWARD_SWEEP` or `SOR_SYMMETRIC_SWEEP`, the default being `SOR_FORWARD_SWEEP`. Setting the type to be `SOR_SYMMETRIC_SWEEP` produces the SSOR method. In addition, each process can locally and independently perform the specified variant of SOR with the types `SOR_LOCAL_FORWARD_SWEEP`, `SOR_LOCAL_BACKWARD_SWEEP`, and `SOR_LOCAL_SYMMETRIC_SWEEP`. These variants can also be set with the options `-pc_sor_omega <omega>`, `-pc_sor_its <its>`, `-pc_sor_lits <lits>`, `-pc_sor_backward`, `-pc_sor_symmetric`, `-pc_sor_local_forward`, `-pc_sor_local_backward`, and `-pc_sor_local_symmetric`.

The Eisenstat trick [Eis81] for SSOR preconditioning can be employed with the method `PCEISENSTAT` (`-pc_type eisenstat`). By using both left and right preconditioning of the linear system, this variant of SSOR requires about half of the floating-point operations for conventional SSOR. The option `-pc_eisenstat_no_diagonal_scaling`) (or the routine `PCEisenstatSetNoDiagonalScaling()`) turns off diagonal scaling in conjunction with Eisenstat SSOR method, while the option `-pc_eisenstat_omega <omega>` (or the routine `PCEisenstatSetOmega(PC pc,PetscReal omega)`) sets the SSOR relaxation coefficient, `omega`, as discussed above.

LU Factorization

The LU preconditioner provides several options. The first, given by the command

```
PCFactorSetUseInPlace(PC pc,PetscBool flg);
```

causes the factorization to be performed in-place and hence destroys the original matrix. The options database variant of this command is `-pc_factor_in_place`. Another direct preconditioner option is selecting the ordering of equations with the command `-pc_factor_mat_ordering_type <ordering>`. The possible orderings are

- `MATORDERINGNATURAL` - Natural
- `MATORDERINGND` - Nested Dissection
- `MATORDERING1WD` - One-way Dissection
- `MATORDERINGRCM` - Reverse Cuthill-McKee
- `MATORDERINGQMD` - Quotient Minimum Degree

These orderings can also be set through the options database by specifying one of the following: `-pc_factor_mat_ordering_type natural`, or `nd`, or `1wd`, or `rcm`, or `qmd`. In addition, see `MatGetOrdering()`, discussed in *Matrix Factorization*.

The sparse LU factorization provided in PETSc does not perform pivoting for numerical stability (since they are designed to preserve nonzero structure), and thus occasionally a LU factorization will fail with a zero pivot when, in fact, the matrix is non-singular. The option `-pc_factor_nonzeros_along_diagonal <tol>` will often help eliminate the zero pivot, by preprocessing the column ordering to remove small values from the diagonal. Here, `tol` is an optional tolerance to decide if a value is nonzero; by default it is `1.e-10`.

In addition, *Memory Allocation for Sparse Matrix Factorization* provides information on preallocation of memory for anticipated fill during factorization. Such tuning can significantly enhance performance, since it eliminates the considerable overhead for dynamic memory allocation.

Block Jacobi and Overlapping Additive Schwarz Preconditioners

The block Jacobi and overlapping additive Schwarz methods in PETSc are supported in parallel; however, only the uniprocess version of the block Gauss-Seidel method is currently in place. By default, the PETSc implementations of these methods employ ILU(0) factorization on each individual block (that is, the default solver on each subblock is `PCType=PCILU`, `KSPType=KSPPREONLY`); the user can set alternative linear solvers via the options `-sub_ksp_type` and `-sub_pc_type`. In fact, all of the `KSP` and `PC` options can be applied to the subproblems by inserting the prefix `-sub_` at the beginning of the option name. These options database commands set the particular options for *all* of the blocks within the global problem. In addition, the routines

```
PCBJacobiGetSubKSP(PC pc,PetscInt *n_local,PetscInt *first_local,KSP **subksp);
PCASMGetSubKSP(PC pc,PetscInt *n_local,PetscInt *first_local,KSP **subksp);
```

extract the `KSP` context for each local block. The argument `n_local` is the number of blocks on the calling process, and `first_local` indicates the global number of the first block on the process. The blocks are numbered successively by processes from zero through $b_g - 1$, where b_g is the number of global blocks. The array of `KSP` contexts for the local blocks is given by `subksp`. This mechanism enables the user to set different solvers for the various blocks. To set the appropriate data structures, the user *must* explicitly call `KSPSetUp()` before calling `PCBJacobiGetSubKSP()` or `PCASMGetSubKSP()`. For further details, see KSP Tutorial ex7 or KSP Tutorial ex8.

The block Jacobi, block Gauss-Seidel, and additive Schwarz preconditioners allow the user to set the number of blocks into which the problem is divided. The options database commands to set this value are `-pc_bjacobi_blocks n` and `-pc_bgs_blocks n`, and, within a program, the corresponding routines are

```
PCBJacobiSetTotalBlocks(PC pc,PetscInt blocks,PetscInt *size);
PCASMSetTotalSubdomains(PC pc,PetscInt n,IS *is,IS *islocal);
PCASMSetType(PC pc,PCASMType type);
```

The optional argument `size` is an array indicating the size of each block. Currently, for certain parallel matrix formats, only a single block per process is supported. However, the `MATMPIAIJ` and `MATMPIBAIJ` formats support the use of general blocks as long as no blocks are shared among processes. The `is` argument contains the index sets that define the subdomains.

The object `PCASMType` is one of `PC_ASM_BASIC`, `PC_ASM_INTERPOLATE`, `PC_ASM_RESTRICT`, or`PC_ASM_NONE` and may also be set with the options database `-pc_asm_type [basic`, `interpolate`, `restrict`, `none]`. The type `PC_ASM_BASIC` (or `-pc_asm_type basic`) corresponds to the standard additive Schwarz method that uses the full restriction and interpolation operators. The type `PC_ASM_RESTRICT` (or `-pc_asm_type restrict`) uses a full restriction operator, but during the interpolation process ignores the off-process values. Similarly, `PC_ASM_INTERPOLATE` (or `-pc_asm_type interpolate`) uses a limited restriction process in conjunction with a full interpolation, while `PC_ASM_NONE` (or `-pc_asm_type none`) ignores off-process values for both restriction and interpolation. The ASM types with limited restriction or interpolation were suggested by Xiao-Chuan Cai and Marcus Sarkis [CS97]. `PC_ASM_RESTRICT` is the PETSc default, as it saves substantial communication and for many problems has the added benefit of requiring fewer iterations for convergence than the standard additive Schwarz method.

The user can also set the number of blocks and sizes on a per-process basis with the commands

```
PCBJacobiSetLocalBlocks(PC pc,PetscInt blocks,PetscInt *size);
PCASMSetLocalSubdomains(PC pc,PetscInt N,IS *is,IS *islocal);
```

For the ASM preconditioner one can use the following command to set the overlap to compute in constructing the subdomains.

```
PCASMSetOverlap(PC pc,PetscInt overlap);
```

The overlap defaults to 1, so if one desires that no additional overlap be computed beyond what may have been set with a call to `PCASMSetTotalSubdomains()` or `PCASMSetLocalSubdomains()`, then `overlap` must be set to be 0. In particular, if one does *not* explicitly set the subdomains in an application code, then all overlap would be computed internally by PETSc, and using an overlap of 0 would result in an ASM variant that is equivalent to the block Jacobi preconditioner. Note that one can define initial index sets `is` with *any* overlap via `PCASMSetTotalSubdomains()` or `PCASMSetLocalSubdomains()`; the routine `PCASMSetOverlap()` merely allows PETSc to extend that overlap further if desired.

`PCGASM` is an experimental generalization of `PCASM` that allows the user to specify subdomains that span multiple MPI ranks. This can be useful for problems where small subdomains result in poor convergence. To be effective, the multirank subproblems must be solved using a sufficient strong subsolver, such as LU, for which `SuperLU_DIST` or a similar parallel direct solver could be used; other choices may include a multigrid solver on the subdomains.

The interface for `PCGASM` is similar to that of `PCASM`. In particular, `PCGASMType` is one of `PC_GASM_BASIC`, `PC_GASM_INTERPOLATE`, `PC_GASM_RESTRICT`, `PC_GASM_NONE`. These options have the same meaning as with `PCASM` and may also be set with the options database `-pc_gasm_type [basic`, `interpolate`, `restrict`, `none]`.

Unlike `PCASM`, however, `PCGASM` allows the user to define subdomains that span multiple MPI ranks. The simplest way to do this is using a call to `PCGASMSetTotalSubdomains(PC pc,PetscPetscInt N)` with the total number of subdomains `N` that is smaller than the MPI communicator `size`. In this case `PCGASM` will coalesce `size/N` concecutive single-rank subdomains into a single multi-rank subdomain. The single-rank subdomains contain the degrees of freedom corresponding to the locally-owned rows of the `PCGASM` preconditioning matrix – these are the subdomains `PCASM` and `PCGASM` use by default.

Each of the multirank subdomain subproblems is defined on the subcommunicator that contains the coalesced `PCGASM` ranks. In general this might not result in a very good subproblem if the single-rank problems corresponding to the coalesced ranks are not very strongly connected. In the future this will be addressed with a hierarchical partitioner that generates well-connected coarse subdomains first before subpartitioning them into the single-rank subdomains.

In the meantime the user can provide his or her own multi-rank subdomains by calling`PCGASMSetSubdomains(PC,IS[],IS[])` where each of the `IS` objects on the list defines the inner (without the overlap) or the outer (including the overlap) subdomain on the subcommunicator of the `IS` object. A helper subroutine `PCGASMCreateSubdomains2D()` is similar to PCASM's but is capable of constructing multi-rank subdomains that can be then used with `PCGASMSetSubdomains()`. An alternative way of creating multi-rank subdomains is by using the underlying DM object, if it is capable of generating such decompositions via `DMCreateDomainDecomposition()`. Ordinarily the decomposition specified by the user via `PCGASMSetSubdomains()` takes precedence, unless `PCGASMSetUseDMSubdomains()` instructs `PCGASM` to prefer `DM`-created decompositions.

Currently there is no support for increasing the overlap of multi-rank subdomains via`PCGASMSetOverlap()` – this functionality works only for subdomains that fit within a single MPI rank, exactly as in `PCASM`.

Examples of the described PCGASM usage can be found in KSP Tutorial ex62. In particular, `runex62_superlu_dist` illustrates the use of `SuperLU_DIST` as the subdomain solver on coalesced multi-rank subdomains. The `runex62_2D_*` examples illustrate the use of `PCGASMCreateSubdomains2D()`.

Algebraic Multigrid (AMG) Preconditioners

PETSc has a native algebraic multigrid preconditioner `PCGAMG` – *gamg* – and interfaces to two external AMG packages: *hypre* and *ML*. *Hypre* is relatively monolithic in that a PETSc matrix is into a hypre matrix and then *hypre* is called to do the entire solve. *ML* is more modular in that PETSc only has *ML* generate the coarse grid spaces (columns of the prolongation operator), which is core of an AMG method, and then constructs a `PCMG` with Galerkin coarse grid operator construction. GAMG is designed from the beginning to be modular, to allow for new components to be added easily and also populates a multigrid preconditioner `PCMG` so generic multigrid parameters are used. PETSc provides a fully supported (smoothed) aggregation AMG, (`-pc_type gamg -pc_gamg_type agg` or `PCSetType(pc,PCGAMG)` and `PCGAMGSetType(pc,PCGAMGAGG)`, as well as reference implementations of a classical AMG method (`-pc_gamg_type classical`), a hybrid geometric AMG method (`-pc_gamg_type geo`), and a 2.5D AMG method DofColumns [ISG15]. GAMG does require the use of (MPI)AIJ matrices. For instance, BAIJ matrices are not supported. One can use AIJ instead of BAIJ without changing any code other than the constructor (or the `-mat_type` from the command line). For instance, `MatSetValuesBlocked` works with AIJ matrices.

GAMG provides unsmoothed aggregation (`-pc_gamg_agg_nsmooths 0`) and smoothed aggregation (`-pc_gamg_agg_nsmooths 1` or `PCGAMGSetNSmooths(pc,1)`). Smoothed aggregation (SA) is recommended for symmetric positive definite systems. Unsmoothed aggregation can be useful for asymmetric problems and problems where highest eigen estimates are problematic. If poor convergence rates are observed using the smoothed version one can test unsmoothed aggregation.

Eigenvalue estimates: The parameters for the KSP eigen estimator, use for SA, can be set with `-pc_gamg_esteig_ksp_max_it` and `-pc_gamg_esteig_ksp_type`. For example CG generally converges to the highest eigenvalue fast than GMRES (the default for KSP) if your problem is symmetric positive definite. One can specify CG with `-pc_gamg_esteig_ksp_type cg`. The default for `-pc_gamg_esteig_ksp_max_it` is 10, which we have found is pretty safe with a (default) safety factor of 1.1. One can specify the range of real eigenvalues, in the same way that one can for Chebyshev KSP solvers (smoothers), with `-pc_gamg_eigenvalues <emin,emax>`. GAMG sets the MG smoother type to chebyshev by default. By default, GAMG uses its eigen estimate, if it has one, for Chebyshev smoothers if the smoother uses Jacobi preconditioning. This can be overridden with `-pc_gamg_use_sa_esteig <true,false>`.

AMG methods requires knowledge of the number of degrees of freedom per vertex, the default is one (a scalar problem). Vector problems like elasticity should set the block size of the matrix appropriately with `-mat_block_size bs` or `MatSetBlockSize(mat,bs)`. Equations must be ordered in "vertex-major" ordering (e.g., $x_1, y_1, z_1, x_2, y_2, ...$).

Near null space: Smoothed aggregation requires an explicit representation of the (near) null space of the operator for optimal performance. One can provide an orthonormal set of null space vectors with `MatSetNearNullSpace()`. The vector of all ones is the default, for each variable given by the block size (e.g., the translational rigid body modes). For elasticity, where rotational rigid body modes are required to complete the near null space you can use `MatNullSpaceCreateRigidBody()` to create the null space vectors and then `MatSetNearNullSpace()`.

Coarse grid data model: The GAMG framework provides for reducing the number of active processes on coarse grids to reduce communication costs when there is not enough parallelism to keep relative communication costs down. Most AMG solver reduce to just one active process on the coarsest grid (the PETSc MG framework also supports redundantly solving the coarse grid on all processes to potentially reduce communication costs), although this forcing to one process can be overridden if one wishes to use a parallel coarse grid solver. GAMG generalizes this by reducing the active number of processes on other coarse grids as well. GAMG will select the number of active processors by fitting the desired number of equation per process (set with `-pc_gamg_process_eq_limit <50>,`) at each level given that size of each level. If $P_i < P$ processors are desired on a level i then the first P_i ranks are populated with the grid and the remaining are empty on that grid. One can, and probably should, repartition the coarse grids with `-pc_gamg_repartition <true>,`, otherwise an integer process reduction factor (q) is selected and the equations on the first q processes are move to process 0, and so on. As mentioned multigrid generally coarsens the problem until it is small enough to be solved with an exact solver (eg, LU or SVD) in a relatively small time. GAMG will stop coarsening when the number of equation on a grid falls below at threshold give by `-pc_gamg_coarse_eq_limit <50>,`.

Coarse grid parameters: There are several options to provide parameters to the coarsening algorithm and parallel data layout. Run a code that uses GAMG with `-help` to get full listing of GAMG parameters with short parameter descriptions. The rate of coarsening is critical in AMG performance – too slow of coarsening will result in an overly expensive solver per iteration and too fast coarsening will result in decrease in the convergence rate. `-pc_gamg_threshold <0>` and `-pc_gamg_square_graph <1>,` are the primary parameters that control coarsening rates, which is very important for AMG performance. A greedy maximal independent set (MIS) algorithm is used in coarsening. Squaring the graph implements so called MIS-2, the root vertex in an aggregate is more than two edges away from another root vertex, instead of more than one in MIS. The threshold parameter sets a normalized threshold for which edges are removed from the MIS graph, thereby coarsening slower. Zero will keep all non-zero edges, a negative number will keep zero edges, a positive number will drop small edges. Typical finite threshold values are in the range of $0.01 - 0.05$. There are additional parameters for changing the weights on coarse grids. Note, the parallel algorithm requires symmetric weights/matrix. You must use `-pc_gamg_sym_graph <true>` to symmetrize the graph if your problem is not symmetric.

Trouble shooting algebraic multigrid methods: If *GAMG*, *ML*, or *hypre* does not perform well the first thing to try is one of the other methods. Often the default parameters or just the strengths of different algorithms can fix performance problems or provide useful information to guide further debugging. There are several sources of poor performance of AMG solvers and often special purpose methods must be developed to achieve the full potential of multigrid. To name just a few sources of performance degradation that may not be fixed with parameters in PETSc currently: non-elliptic operators, curl/curl operators, highly stretched grids or highly anisotropic problems, large jumps in material coefficients with complex geometry (AMG is particularly well suited to jumps in coefficients but it is not a perfect solution), highly incompressible elasticity, not to mention ill-posed problems, and many others. For Grad-Div and Curl-Curl operators, you may want to try the Auxiliary-space Maxwell Solver (AMS, `-pc_type hypre -pc_hypre_type ams`) or the Auxiliary-space Divergence Solver (ADS, `-pc_type hypre -pc_hypre_type ads`) solvers. These solvers need some additional information on the underlying mesh; specifically, AMS needs the discrete gradient operator, which can be specified via `PCHYPRESetDiscreteGradient()`. In addition to the discrete gradient, ADS also needs the specification of the discrete curl operator, which can be set using `PCHYPRESetDiscreteCurl()`.

I am converging slowly, what do I do? AMG methods are sensitive to coarsening rates and methods; for GAMG use `-pc_gamg_threshold <x>` to regulate coarsening rates and PCGAMGSetThreshold, higher values decrease coarsening rate. Squaring the graph is the second mechanism for increasing coarsening rate. Use `-pc_gamg_square_graph <N>,`, or `PCGAMGSetSquareGraph(pc,N)`, to square the graph on the finest N levels. A high threshold (e.g., $x = 0.08$) will result in an expensive but potentially powerful preconditioner, and a low threshold (e.g., $x = 0.0$) will result in faster coarsening, fewer levels, cheaper solves, and generally worse convergence rates.

One can run with `-info` and grep for "GAMG" to get some statistics on each level, which can be used to see if you are coarsening at an appropriate rate. With smoothed aggregation you generally want to coarse at about a rate of 3:1 in each dimension. Coarsening too slow will result in large numbers of non-zeros per row on coarse grids (this is reported). The number of non-zeros can go up very high, say about 300 (times the degrees-of-freedom per vertex) on a 3D hex mesh. One can also look at the grid complexity, which is also reported (the ration of the total number of matrix entries for all levels to the number of matrix entries on the fine level). Grid complexity should be well under 2.0 and preferably around 1.3 or lower. If convergence is poor and the Galerkin coarse grid construction is much smaller than the time for each solve then one can safely decrease the coarsening rate. `-pc_gamg_threshold` 0.0 is the simplest and most robust option, and is recommended if poor convergence rates are observed, at least until the source of the problem is discovered. In conclusion, if convergence is slow then decreasing the coarsening rate (increasing the threshold) should be tried.

A note on Chebyshev smoothers. Chebyshev solvers are attractive as multigrid smoothers because they can target a specific interval of the spectrum which is the purpose of a smoother. The spectral bounds for Chebyshev solvers are simple to compute because they rely on the highest eigenvalue of your (diagonally preconditioned) operator, which is conceptually simple to compute. However, if this highest eigenvalue estimate is not accurate (too low) then the solvers can fail with and indefinite preconditioner message. One can run with `-info` and grep for "GAMG" to get these estimates or use `-ksp_view`. These highest eigenvalues are generally between 1.5-3.0. For symmetric positive definite systems CG is a better eigenvalue estimator `-mg_levels_esteig_ksp_type cg`. Indefinite matrix messages are often caused by bad Eigen estimates. Explicitly damped Jacobi or Krylov smoothers can provide an alternative to Chebyshev and *hypre* has alternative smoothers.

Now am I solving alright, can I expect better? If you find that you are getting nearly on digit in reduction of the residual per iteration and are using a modest number of point smoothing steps (e.g., 1-4 iterations of SOR), then you may be fairly close to textbook multigrid efficiency. Although you also need to check the setup costs. This can be determined by running with `-log_view` and check that the time for the Galerkin coarse grid construction (`MatPtAP`) is not (much) more than the time spent in each solve (`KSPSolve`). If the `MatPtAP` time is too large then one can increase the coarsening rate by decreasing the threshold and squaring the coarsening graph (`-pc_gamg_square_graph <N>`, squares the graph on the finest N levels). Likewise if your `MatPtAP` time is small and your convergence rate is not ideal then you could decrease the coarsening rate.

PETSc's AMG solver is constructed as a framework for developers to easily add AMG capabilities, like a new AMG methods or an AMG component like a matrix triple product. Contact us directly if you are interested in contributing.

Balancing Domain Decomposition by Constraints

PETSc provides the Balancing Domain Decomposition by Constraints (BDDC) method for preconditioning parallel finite element problems stored in unassembled format (see `MATIS`). BDDC is a 2-level non-overlapping domain decomposition method which can be easily adapted to different problems and discretizations by means of few user customizations. The application of the preconditioner to a vector consists in the static condensation of the residual at the interior of the subdomains by means of local Dirichet solves, followed by an additive combination of Neumann local corrections and the solution of a global coupled coarse problem. Command line options for the underlying `KSP` objects are prefixed by `-pc_bddc_dirichlet`, `-pc_bddc_neumann`, and `-pc_bddc_coarse` respectively.

The current implementation supports any kind of linear system, and assumes a one-to-one mapping between subdomains and MPI processes. Complex numbers are supported as well. For non-symmetric problems, use the runtime option `-pc_bddc_symmetric 0`.

Unlike conventional non-overlapping methods that iterates just on the degrees of freedom at the interface between subdomain, `PCBDDC` iterates on the whole set of degrees of freedom, allowing the use of approximate subdomain solvers. When using approximate solvers, the command line switches `-pc_bddc_dirichlet_approximate` and/or `-pc_bddc_neumann_approximate` should be used to inform `PCBDDC`. If any of the local problems is singular, the nullspace of the local operator should be attached to the local matrix via `MatSetNullSpace()`.

At the basis of the method there's the analysis of the connected components of the interface for the detection of vertices, edges and faces equivalence classes. Additional information on the degrees of freedom can be supplied to `PCBDDC` by using the following functions:

- `PCBDDCSetDofsSplitting()`
- `PCBDDCSetLocalAdjacencyGraph()`
- `PCBDDCSetPrimalVerticesLocalIS()`
- `PCBDDCSetNeumannBoundaries()`
- `PCBDDCSetDirichletBoundaries()`
- `PCBDDCSetNeumannBoundariesLocal()`
- `PCBDDCSetDirichletBoundariesLocal()`

Crucial for the convergence of the iterative process is the specification of the primal constraints to be imposed at the interface between subdomains. `PCBDDC` uses by default vertex continuities and edge arithmetic averages, which are enough for the three-dimensional Poisson problem with constant coefficients. The user can switch on and off the usage of vertices, edges or face constraints by using the command line switches `-pc_bddc_use_vertices`, `-pc_bddc_use_edges`, `-pc_bddc_use_faces`. A customization of the constraints is available by attaching a `MatNullSpace` object to the preconditioning matrix via `MatSetNearNullSpace()`. The vectors of the `MatNullSpace` object should represent the constraints in the form of quadrature rules; quadrature rules for different classes of the interface can be listed in the same vector. The number of vectors of the `MatNullSpace` object corresponds to the maximum number of constraints that can be imposed for each class. Once all the quadrature rules for a given interface class have been extracted, an SVD operation is performed to retain the non-singular modes. As an example, the rigid body modes represent an effective choice for elasticity, even in the almost incompressible case. For particular problems, e.g. edge-based discretization with Nedelec elements, a user defined change of basis of the degrees of freedom can be beneficial for `PCBDDC`; use `PCBDDCSetChangeOfBasisMat()` to customize the change of basis.

The BDDC method is usually robust with respect to jumps in the material parameters aligned with the interface; for PDEs with more than one material parameter you may also consider to use the so-called deluxe scaling, available via the command line switch `-pc_bddc_use_deluxe_scaling`. Other scalings are available, see `PCISSetSubdomainScalingFactor()`, `PCISSetSubdomainDiagonalScaling()` or

`PCISSetUseStiffnessScaling()`. However, the convergence properties of the BDDC method degrades in presence of large jumps in the material coefficients not aligned with the interface; for such cases, PETSc has the capability of adaptively computing the primal constraints. Adaptive selection of constraints could be requested by specifying a threshold value at command line by using `-pc_bddc_adaptive_threshold x`. Valid values for the threshold `x` ranges from 1 to infinity, with smaller values corresponding to more robust preconditioners. For SPD problems in 2D, or in 3D with only face degrees of freedom (like in the case of Raviart-Thomas or Brezzi-Douglas-Marini elements), such a threshold is a very accurate estimator of the condition number of the resulting preconditioned operator. Since the adaptive selection of constraints for BDDC methods is still an active topic of research, its implementation is currently limited to SPD problems; moreover, because the technique requires the explicit knowledge of the local Schur complements, it needs the external package MUMPS.

When solving problems decomposed in thousands of subdomains or more, the solution of the BDDC coarse problem could become a bottleneck; in order to overcome this issue, the user could either consider to solve the parallel coarse problem on a subset of the communicator associated with `PCBDDC` by using the command line switch `-pc_bddc_coarse_redistribute`, or instead use a multilevel approach. The latter can be requested by specifying the number of requested level at command line (`-pc_bddc_levels`) or by using `PCBDDCSetLevels()`. An additional parameter (see `PCBDDCSetCoarseningRatio()`) controls the number of subdomains that will be generated at the next level; the larger the coarsening ratio, the lower the number of coarser subdomains.

For further details, see the example KSP Tutorial ex59 and the online documentation for `PCBDDC`.

Shell Preconditioners

The shell preconditioner simply uses an application-provided routine to implement the preconditioner. To set this routine, one uses the command

```
PCShellSetApply(PC pc,PetscErrorCode (*apply)(PC,Vec,Vec));
```

Often a preconditioner needs access to an application-provided data structured. For this, one should use

```
PCShellSetContext(PC pc,void *ctx);
```

to set this data structure and

```
PCShellGetContext(PC pc,void **ctx);
```

to retrieve it in `apply`. The three routine arguments of `apply()` are the `PC`, the input vector, and the output vector, respectively.

For a preconditioner that requires some sort of "setup" before being used, that requires a new setup every time the operator is changed, one can provide a routine that is called every time the operator is changed (usually via `KSPSetOperators()`).

```
PCShellSetSetUp(PC pc,PetscErrorCode (*setup)(PC));
```

The argument to the `setup` routine is the same `PC` object which can be used to obtain the operators with `PCGetOperators()` and the application-provided data structure that was set with `PCShellSetContext()`.

Combining Preconditioners

The `PC` type `PCCOMPOSITE` allows one to form new preconditioners by combining already-defined preconditioners and solvers. Combining preconditioners usually requires some experimentation to find a combination of preconditioners that works better than any single method. It is a tricky business and is not recommended until your application code is complete and running and you are trying to improve performance. In many cases using a single preconditioner is better than a combination; an exception is the multigrid/multilevel preconditioners (solvers) that are always combinations of some sort, see *Multigrid Preconditioners*.

Let B_1 and B_2 represent the application of two preconditioners of type `type1` and `type2`. The preconditioner $B = B_1 + B_2$ can be obtained with

```
PCSetType(pc,PCCOMPOSITE);
PCCompositeAddPC(pc,type1);
PCCompositeAddPC(pc,type2);
```

Any number of preconditioners may added in this way.

This way of combining preconditioners is called additive, since the actions of the preconditioners are added together. This is the default behavior. An alternative can be set with the option

```
PCCompositeSetType(PC pc,PCCompositeType PC_COMPOSITE_MULTIPLICATIVE);
```

In this form the new residual is updated after the application of each preconditioner and the next preconditioner applied to the next residual. For example, with two composed preconditioners: B_1 and B_2; $y = Bx$ is obtained from

$$\begin{aligned} y &= B_1 x \\ w_1 &= x - Ay \\ y &= y + B_2 w_1 \end{aligned}$$

Loosely, this corresponds to a Gauss-Seidel iteration, while additive corresponds to a Jacobi iteration.

Under most circumstances, the multiplicative form requires one-half the number of iterations as the additive form; however, the multiplicative form does require the application of A inside the preconditioner.

In the multiplicative version, the calculation of the residual inside the preconditioner can be done in two ways: using the original linear system matrix or using the matrix used to build the preconditioners B_1, B_2, etc. By default it uses the “preconditioner matrix”, to use the `Amat` matrix use the option

```
PCSetUseAmat(PC pc);
```

The individual preconditioners can be accessed (in order to set options) via

```
PCCompositeGetPC(PC pc,PetscInt count,PC *subpc);
```

For example, to set the first sub preconditioners to use ILU(1)

```
PC subpc;
PCCompositeGetPC(pc,0,&subpc);
PCFactorSetFill(subpc,1);
```

One can also change the operator that is used to construct a particular PC in the composite PC call PCSetOperators() on the obtained PC.

These various options can also be set via the options database. For example, `-pc_type composite -pc_composite_pcs jacobi,ilu` causes the composite preconditioner to be used with two preconditioners: Jacobi and ILU. The option `-pc_composite_type multiplicative` initiates the multiplicative version of the algorithm, while `-pc_composite_type additive` the additive version. Using the `Amat` matrix is obtained with the option `-pc_use_amat`. One sets options for the sub-preconditioners with the extra prefix `-sub_N_` where `N` is the number of the sub-preconditioner. For example, `-sub_0_pc_ifactor_fill 0`.

PETSc also allows a preconditioner to be a complete linear solver. This is achieved with the `PCKSP` type.

```
PCSetType(PC pc,PCKSP PCKSP);
PCKSPGetKSP(pc,&ksp);
 /* set any KSP/PC options */
```

From the command line one can use 5 iterations of biCG-stab with ILU(0) preconditioning as the preconditioner with `-pc_type ksp -ksp_pc_type ilu -ksp_ksp_max_it 5 -ksp_ksp_type bcgs`.

By default the inner `KSP` solver uses the outer preconditioner matrix, `Pmat`, as the matrix to be solved in the linear system; to use the matrix that defines the linear system, `Amat` use the option

```
PCSetUseAmat(PC pc);
```

or at the command line with `-pc_use_amat`.

Naturally, one can use a `PCKSP` preconditioner inside a composite preconditioner. For example, `-pc_type composite -pc_composite_pcs ilu,ksp -sub_1_pc_type jacobi -sub_1_ksp_max_it 10` uses two preconditioners: ILU(0) and 10 iterations of GMRES with Jacobi preconditioning. However, it is not clear whether one would ever wish to do such a thing.

Multigrid Preconditioners

A large suite of routines is available for using geometric multigrid as a preconditioner[2]. In the `PC` framework, the user is required to provide the coarse grid solver, smoothers, restriction and interpolation operators, and code to calculate residuals. The `PC` package allows these components to be encapuslated within a PETSc-compliant preconditioner. We fully support both matrix-free and matrix-based multigrid solvers.

A multigrid preconditioner is created with the four commands

```
KSPCreate(MPI_Comm comm,KSP *ksp);
KSPGetPC(KSP ksp,PC *pc);
PCSetType(PC pc,PCMG);
PCMGSetLevels(pc,PetscInt levels,MPI_Comm *comms);
```

A large number of parameters affect the multigrid behavior. The command

```
PCMGSetType(PC pc,PCMGType mode);
```

indicates which form of multigrid to apply [SBjorstadG96].

For standard V or W-cycle multigrids, one sets the `mode` to be `PC_MG_MULTIPLICATIVE`; for the additive form (which in certain cases reduces to the BPX method, or additive multilevel Schwarz, or multilevel diagonal scaling), one uses `PC_MG_ADDITIVE` as the `mode`. For a variant of full multigrid, one can use `PC_MG_FULL`, and for the Kaskade algorithm `PC_MG_KASKADE`. For the multiplicative and full multigrid options, one can use a W-cycle by calling

```
PCMGSetCycleType(PC pc,PCMGCycleType ctype);
```

with a value of `PC_MG_CYCLE_W` for `ctype`. The commands above can also be set from the options database. The option names are `-pc_mg_type [multiplicative, additive, full, kaskade]`, and `-pc_mg_cycle_type <ctype>`.

---

[2] See *Algebraic Multigrid (AMG) Preconditioners* for information on using algebraic multigrid.

The user can control the amount of smoothing by configuring the solvers on the levels. By default, the up and down smoothers are identical. If separate configuration of up and down smooths is required, it can be requested with the option `-pc_mg_distinct_smoothup` or the routine

```
PCMGSetDistinctSmoothUp(PC pc);
```

The multigrid routines, which determine the solvers and interpolation/restriction operators that are used, are mandatory. To set the coarse grid solver, one must call

```
PCMGGetCoarseSolve(PC pc,KSP *ksp);
```

and set the appropriate options in `ksp`. Similarly, the smoothers are controlled by first calling

```
PCMGGetSmoother(PC pc,PetscInt level,KSP *ksp);
```

and then setting the various options in the `ksp.` For example,

```
PCMGGetSmoother(pc,1,&ksp);
KSPSetOperators(ksp,A1,A1);
```

sets the matrix that defines the smoother on level 1 of the multigrid. While

```
PCMGGetSmoother(pc,1,&ksp);
KSPGetPC(ksp,&pc);
PCSetType(pc,PCSOR);
```

sets SOR as the smoother to use on level 1.

To use a different pre- or postsmoother, one should call the following routines instead.

```
PCMGGetSmootherUp(PC pc,PetscInt level,KSP *upksp);
PCMGGetSmootherDown(PC pc,PetscInt level,KSP *downksp);
```

Use

```
PCMGSetInterpolation(PC pc,PetscInt level,Mat P);
```

and

```
PCMGSetRestriction(PC pc,PetscInt level,Mat R);
```

to define the intergrid transfer operations. If only one of these is set, its transpose will be used for the other.

It is possible for these interpolation operations to be matrix free (see *Matrix-Free Matrices*); One should then make sure that these operations are defined for the (matrix-free) matrices passed in. Note that this system is arranged so that if the interpolation is the transpose of the restriction, you can pass the same `mat` argument to both `PCMGSetRestriction()` and `PCMGSetInterpolation()`.

On each level except the coarsest, one must also set the routine to compute the residual. The following command suffices:

```
PCMGSetResidual(PC pc,PetscInt level,PetscErrorCode (*residual)(Mat,Vec,Vec,Vec),Mat␣
↪mat);
```

The `residual()` function normally does not need to be set if one's operator is stored in `Mat` format. In certain circumstances, where it is much cheaper to calculate the residual directly, rather than through the usual formula $b - Ax$, the user may wish to provide an alternative.

Finally, the user may provide three work vectors for each level (except on the finest, where only the residual work vector is required). The work vectors are set with the commands

```
PCMGSetRhs(PC pc,PetscInt level,Vec b);
PCMGSetX(PC pc,PetscInt level,Vec x);
PCMGSetR(PC pc,PetscInt level,Vec r);
```

The `PC` references these vectors, so you should call `VecDestroy()` when you are finished with them. If any of these vectors are not provided, the preconditioner will allocate them.

One can control the `KSP` and `PC` options used on the various levels (as well as the coarse grid) using the prefix `mg_levels_` (`mg_coarse_` for the coarse grid). For example, `-mg_levels_ksp_type cg` will cause the CG method to be used as the Krylov method for each level. Or `-mg_levels_pc_type ilu -mg_levels_pc_factor_levels 2` will cause the ILU preconditioner to be used on each level with two levels of fill in the incomplete factorization.

2.3.5 Solving Block Matrices

Block matrices represent an important class of problems in numerical linear algebra and offer the possibility of far more efficient iterative solvers than just treating the entire matrix as black box. In this section we use the common linear algebra definition of block matrices where matrices are divided in a small, problem-size independent (two, three or so) number of very large blocks. These blocks arise naturally from the underlying physics or discretization of the problem, for example, the velocity and pressure. Under a certain numbering of unknowns the matrix can be written as

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{pmatrix},$$

where each A_{ij} is an entire block. On a parallel computer the matrices are not explicitly stored this way. Instead, each process will own some of the rows of A_{0*}, A_{1*} etc. On a process, the blocks may be stored one block followed by another

$$\begin{pmatrix} A_{00_{00}} & A_{00_{01}} & A_{00_{02}} & \dots & A_{01_{00}} & A_{01_{02}} & \dots \\ A_{00_{10}} & A_{00_{11}} & A_{00_{12}} & \dots & A_{01_{10}} & A_{01_{12}} & \dots \\ A_{00_{20}} & A_{00_{21}} & A_{00_{22}} & \dots & A_{01_{20}} & A_{01_{22}} & \dots \\ \dots \\ A_{10_{00}} & A_{10_{01}} & A_{10_{02}} & \dots & A_{11_{00}} & A_{11_{02}} & \dots \\ A_{10_{10}} & A_{10_{11}} & A_{10_{12}} & \dots & A_{11_{10}} & A_{11_{12}} & \dots \\ \dots \end{pmatrix}$$

or interlaced, for example with two blocks

$$\begin{pmatrix} A_{00_{00}} & A_{01_{00}} & A_{00_{01}} & A_{01_{01}} & \dots \\ A_{10_{00}} & A_{11_{00}} & A_{10_{01}} & A_{11_{01}} & \dots \\ \dots \\ A_{00_{10}} & A_{01_{10}} & A_{00_{11}} & A_{01_{11}} & \dots \\ A_{10_{10}} & A_{11_{10}} & A_{10_{11}} & A_{11_{11}} & \dots \\ \dots \end{pmatrix}.$$

Note that for interlaced storage the number of rows/columns of each block must be the same size. Matrices obtained with `DMCreateMatrix()` where the `DM` is a `DMDA` are always stored interlaced. Block matrices can also be stored using the `MATNEST` format which holds separate assembled blocks. Each of these nested matrices is itself distributed in parallel. It is more efficient to use `MATNEST` with the methods described in this section because there are fewer copies and better formats (e.g. `BAIJ` or `SBAIJ`) can be used for the components, but it is not possible to use many other methods with `MATNEST`. See *Block Matrices* for more on assembling block matrices without depending on a specific matrix format.

The PETSc `PCFIELDSPLIT` preconditioner is used to implement the "block" solvers in PETSc. There are three ways to provide the information that defines the blocks. If the matrices are stored as interlaced then `PCFieldSplitSetFields()` can be called repeatedly to indicate which fields belong to each block. More generally `PCFieldSplitSetIS()` can be used to indicate exactly which rows/columns of the matrix belong to a particular block. You can provide names for each block with these routines, if you do not provide names they are numbered from 0. With these two approaches the blocks may overlap (though generally they will not). If only one block is defined then the complement of the matrices is used to define the other block. Finally the option `-pc_fieldsplit_detect_saddle_point` causes two diagonal blocks to be found, one associated with all rows/columns that have zeros on the diagonals and the rest.

For simplicity in the rest of the section we restrict our matrices to two by two blocks. So the matrix is

$$\left(\begin{array}{cc} A_{00} & A_{01} \\ A_{10} & A_{11} \end{array}\right).$$

On occasion the user may provide another matrix that is used to construct parts of the preconditioner

$$\left(\begin{array}{cc} Ap_{00} & Ap_{01} \\ Ap_{10} & Ap_{11} \end{array}\right).$$

For notational simplicity define $\text{ksp}(A, Ap)$ to mean approximately solving a linear system using `KSP` with operator A and preconditioner built from matrix Ap.

For matrices defined with any number of blocks there are three "block" algorithms available: block Jacobi,

$$\left(\begin{array}{cc} \text{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & \text{ksp}(A_{11}, Ap_{11}) \end{array}\right)$$

block Gauss-Seidel,

$$\left(\begin{array}{cc} I & 0 \\ 0 & A_{11}^{-1} \end{array}\right)\left(\begin{array}{cc} I & 0 \\ -A_{10} & I \end{array}\right)\left(\begin{array}{cc} A_{00}^{-1} & 0 \\ 0 & I \end{array}\right)$$

which is implemented[3] as

$$\left(\begin{array}{cc} I & 0 \\ 0 & \text{ksp}(A_{11}, Ap_{11}) \end{array}\right)\left[\left(\begin{array}{cc} 0 & 0 \\ 0 & I \end{array}\right)+\left(\begin{array}{cc} I & 0 \\ -A_{10} & -A_{11} \end{array}\right)\left(\begin{array}{cc} I & 0 \\ 0 & 0 \end{array}\right)\right]\left(\begin{array}{cc} \text{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & I \end{array}\right)$$

and symmetric block Gauss-Seidel

$$\left(\begin{array}{cc} A_{00}^{-1} & 0 \\ 0 & I \end{array}\right)\left(\begin{array}{cc} I & -A_{01} \\ 0 & I \end{array}\right)\left(\begin{array}{cc} A_{00} & 0 \\ 0 & A_{11}^{-1} \end{array}\right)\left(\begin{array}{cc} I & 0 \\ -A_{10} & I \end{array}\right)\left(\begin{array}{cc} A_{00}^{-1} & 0 \\ 0 & I \end{array}\right).$$

These can be accessed with `-pc_fieldsplit_type<additive,multiplicative, symmetric_multiplicative>` or the function `PCFieldSplitSetType()`. The option prefixes for the internal KSPs are given by `-fieldsplit_name_`.

By default blocks A_{00}, A_{01} and so on are extracted out of `Pmat`, the matrix that the `KSP` uses to build the preconditioner, and not out of `Amat` (i.e., A itself). As discussed above in *Combining Preconditioners*, however, it is possible to use `Amat` instead of `Pmat` by calling `PCSetUseAmat(pc)` or using `-pc_use_amat` on the command line. Alternatively, you can have `PCFieldSplit` extract the diagonal blocks A_{00}, A_{11} etc. out of `Amat` by calling `PCFieldSplitSetDiagUseAmat(pc,PETSC_TRUE)` or supplying command-line argument `-pc_fieldsplit_diag_use_amat`. Similarly, `PCFieldSplitSetOffDiagUseAmat(pc, {PETSC_TRUE)` or `-pc_fieldsplit_off_diag_use_amat` will cause the off-diagonal blocks A_{01}, A_{10} etc. to be extracted out of `Amat`.

---

[3] This may seem an odd way to implement since it involves the "extra" multiply by $-A_{11}$. The reason is this is implemented this way is that this approach works for any number of blocks that may overlap.

For two by two blocks only there are another family of solvers, based on Schur complements. The inverse of the Schur complement factorization is

$$\begin{gathered}
\left[\begin{pmatrix} I & 0 \\ A_{10}A_{00}^{-1} & I \end{pmatrix}\begin{pmatrix} A_{00} & 0 \\ 0 & S \end{pmatrix}\begin{pmatrix} I & A_{00}^{-1}A_{01} \\ 0 & I \end{pmatrix}\right]^{-1} \\
\begin{pmatrix} I & A_{00}^{-1}A_{01} \\ 0 & I \end{pmatrix}^{-1}\begin{pmatrix} A_{00}^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix}\begin{pmatrix} I & 0 \\ A_{10}A_{00}^{-1} & I \end{pmatrix}^{-1} \\
\begin{pmatrix} I & -A_{00}^{-1}A_{01} \\ 0 & I \end{pmatrix}\begin{pmatrix} A_{00}^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix}\begin{pmatrix} I & 0 \\ -A_{10}A_{00}^{-1} & I \end{pmatrix} \\
\begin{pmatrix} A_{00}^{-1} & 0 \\ 0 & I \end{pmatrix}\begin{pmatrix} I & -A_{01} \\ 0 & I \end{pmatrix}\begin{pmatrix} A_{00} & 0 \\ 0 & S^{-1} \end{pmatrix}\begin{pmatrix} I & 0 \\ -A_{10} & I \end{pmatrix}\begin{pmatrix} A_{00}^{-1} & 0 \\ 0 & I \end{pmatrix}.
\end{gathered}$$

The preconditioner is accessed with `-pc_fieldsplit_type schur` and is implemented as

$$\begin{pmatrix} \mathrm{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & I \end{pmatrix}\begin{pmatrix} I & -A_{01} \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ 0 & \mathrm{ksp}(\hat{S}, \hat{S}p) \end{pmatrix}\begin{pmatrix} I & 0 \\ -A_{10}\mathrm{ksp}(A_{00}, Ap_{00}) & I \end{pmatrix}.$$

Where $\hat{S} = A_{11} - A_{10}\mathrm{ksp}(A_{00}, Ap_{00})A_{01}$ is the approximate Schur complement.

There are several variants of the Schur complement preconditioner obtained by dropping some of the terms, these can be obtained with `-pc_fieldsplit_schur_fact_type <diag,lower,upper,full>` or the function `PCFieldSplitSetSchurFactType()`. Note that the `diag` form uses the preconditioner

$$\begin{pmatrix} \mathrm{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & -\mathrm{ksp}(\hat{S}, \hat{S}p) \end{pmatrix}.$$

This is done to ensure the preconditioner is positive definite for a common class of problems, saddle points with a positive definite A_{00}: for these the Schur complement is negative definite.

The effectiveness of the Schur complement preconditioner depends on the availability of a good preconditioner $\hat{S}p$ for the Schur complement matrix. In general, you are responsible for supplying $\hat{S}p$ via `PCFieldSplitSchurPrecondition(pc,PC_FIELDSPLIT_SCHUR_PRE_USER,Sp)`. In the absence of a good problem-specific $\hat{S}p$, you can use some of the built-in options.

Using `-pc_fieldsplit_schur_precondition user` on the command line activates the matrix supplied programmatically as explained above.

With `-pc_fieldsplit_schur_precondition a11` (default) $\hat{S}p = A_{11}$ is used to build a preconditioner for $\hat{S}$.

Otherwise, `-pc_fieldsplit_schur_precondition self` will set $\hat{S}p = \hat{S}$ and use the Schur complement matrix itself to build the preconditioner.

The problem with the last approach is that $\hat{S}$ is used in unassembled, matrix-free form, and many preconditioners (e.g., ILU) cannot be built out of such matrices. Instead, you can *assemble* an approximation to $\hat{S}$ by inverting A_{00}, but only approximately, so as to ensure the sparsity of $\hat{S}p$ as much as possible. Specifically, using `-pc_fieldsplit_schur_precondition selfp` will assemble $\hat{S}p = A_{11} - A_{10}\mathrm{inv}(A_{00})A_{01}$.

By default $\mathrm{inv}(A_{00})$ is the inverse of the diagonal of A_{00}, but using `-fieldsplit_1_mat_schur_complement_ainv_type lump` will lump A_{00} first. Using `-fieldsplit_1_mat_schur_complement_ainv_type blockdiag` will use the inverse of the block diagonal of A_{00}. Option `-mat_schur_complement_ainv_type` applies to any matrix of `MatSchurComplement` type and here it is used with the prefix `-fieldsplit_1` of the linear system in the second split.

Finally, you can use the `PCLSC` preconditioner for the Schur complement with `-pc_fieldsplit_type schur -fieldsplit_1_pc_type lsc`. This uses for the preconditioner to $\hat{S}$ the operator

$$\mathrm{ksp}(A_{10}A_{01}, A_{10}A_{01})A_{10}A_{00}A_{01}\mathrm{ksp}(A_{10}A_{01}, A_{10}A_{01})$$

which, of course, introduces two additional inner solves for each application of the Schur complement. The options prefix for this inner `KSP` is `-fieldsplit_1_lsc_`. Instead of constructing the matrix $A_{10}A_{01}$ the user can provide their own matrix. This is done by attaching the matrix/matrices to the Sp matrix they provide with

```
PetscObjectCompose((PetscObject)Sp,"LSC_L",(PetscObject)L);
PetscObjectCompose((PetscObject)Sp,"LSC_Lp",(PetscObject)Lp);
```

2.3.6 Solving Singular Systems

Sometimes one is required to solver singular linear systems. In this case, the system matrix has a nontrivial null space. For example, the discretization of the Laplacian operator with Neumann boundary conditions has a null space of the constant functions. PETSc has tools to help solve these systems.

First, one must know what the null space is and store it using an orthonormal basis in an array of PETSc Vecs. The constant functions can be handled separately, since they are such a common case). Create a `MatNullSpace` object with the command

```
MatNullSpaceCreate(MPI_Comm,PetscBool hasconstants,PetscInt dim,Vec *basis,
↪MatNullSpace *nsp);
```

Here, `dim` is the number of vectors in `basis` and `hasconstants` indicates if the null space contains the constant functions. If the null space contains the constant functions you do not need to include it in the `basis` vectors you provide, nor in the count `dim`.

One then tells the `KSP` object you are using what the null space is with the call

```
MatSetNullSpace(Mat Amat,MatNullSpace nsp);
MatSetTransposeNullSpace(Mat Amat,MatNullSpace nsp);
```

The `Amat` should be the *first* matrix argument used with `KSPSetOperators()`, `SNESSetJacobian()`, or `TSSetIJacobian()`. You can also use `KSPSetNullspace()`. The PETSc solvers will now handle the null space during the solution process.

If one chooses a direct solver (or an incomplete factorization) it may still detect a zero pivot. You can run with the additional options or `-pc_factor_shift_type NONZERO` `-pc_factor_shift_amount <dampingfactor>` to prevent the zero pivot. A good choice for the `dampingfactor` is 1.e-10.

2.3.7 Using External Linear Solvers

PETSc interfaces to several external linear solvers (also see *Acknowledgments*) at the beginning of this manual). To use these solvers, one may:

1. Run `./configure` with the additional options `--download-packagename` e.g. `--download-superlu_dist --download-parmetis` (SuperLU_DIST needs ParMetis) or `--download-mumps --download-scalapack` (MUMPS requires ScaLAPACK).
2. Build the PETSc libraries.
3. Use the runtime option: `-ksp_type preonly -pc_type <pctype> -pc_factor_mat_solver_type <packagename>`. For eg: `-ksp_type preonly -pc_type lu -pc_factor_mat_solver_type superlu_dist`.

Table 2.6: Options for External Solvers

MatType	PCType	MatSolverType	Package (-pc_factor_mat_solver_type)
seqaij	lu	MATSOLVERESSL	essl
seqaij	lu	MATSOLVERLUSOL	lusol
seqaij	lu	MATSOLVERMATLAB	matlab
aij	lu	MATSOLVERMUMPS	mumps
aij	cholesky	•	•
sbaij	cholesky	•	•
seqaij	lu	MATSOLVERSUPERLU	superlu
aij	lu	MATSOLVERSUPERLU_DIST	superlu_dist
seqaij	lu	MATSOLVERUMFPACK	umfpack
seqaij	cholesky	MATSOLVERCHOLMOD	cholmod
aij	lu	MATSOLVERCLIQUE	clique
seqaij	lu	MATSOLVERKLU	klu
dense	lu	MATSOLVERELEMENTAL	elemental
dense	cholesky	•	•
seqaij	lu	MATSOLVERMKL_PARDISO	mkl_pardiso
aij	lu	MATSOLVERMKL_CPARDISO	mkl_cpardiso
aij	lu	MATSOLVERPASTIX	pastix
aij	cholesky	MATSOLVERBAS	bas
aijcusparse	lu	MATSOLVERCUSPARSE	cusparse
aijcusparse	cholesky	•	•
aij	lu, cholesky	MATSOLVERPETSC	petsc
baij	•	•	•
aijcrl	•	•	•
aijperm	•	•	•
seqdense	•	•	•
aij	•	•	•
baij	•	•	•

Continued on next page

Table 2.6 – continued from previous page

MatType	PCType	MatSolverType	Package (-pc_factor_mat_solver_type)
`aijcrl`	•	•	•
`aijperm`	•	•	•
`seqdense`	•	•	•

The default and available input options for each external software can be found by specifying `-help` at runtime.

As an alternative to using runtime flags to employ these external packages, procedural calls are provided for some packages. For example, the following procedural calls are equivalent to runtime options `-ksp_type preonly -pc_type lu -pc_factor_mat_solver_type mumps -mat_mumps_icntl_7 2`:

```
KSPSetType(ksp,KSPPREONLY);
KSPGetPC(ksp,&pc);
PCSetType(pc,PCLU);
PCFactorSetMatSolverType(pc,MATSOLVERMUMPS);
PCFactorGetMatrix(pc,&F);
icntl=7; ival = 2;
MatMumpsSetIcntl(F,icntl,ival);
```

One can also create matrices with the appropriate capabilities by calling `MatCreate()` followed by `MatSetType()` specifying the desired matrix type from *Options for External Solvers*. These matrix types inherit capabilities from their PETSc matrix parents: `seqaij`, `mpiaij`, etc. As a result, the preallocation routines `MatSeqAIJSetPreallocation()`, `MatMPIAIJSetPreallocation()`, etc. and any other type specific routines of the base class are supported. One can also call `MatConvert()` inplace to convert the matrix to and from its base class without performing an expensive data copy. `MatConvert()` cannot be called on matrices that have already been factored.

In *Options for External Solvers*, the base class `aij` refers to the fact that inheritance is based on `MATSEQAIJ` when constructed with a single process communicator, and from `MATMPIAIJ` otherwise. The same holds for `baij` and `sbaij`. For codes that are intended to be run as both a single process or with multiple processes, depending on the `mpiexec` command, it is recommended that both sets of preallocation routines are called for these communicator morphing types. The call for the incorrect type will simply be ignored without any harm or message.

2.4 SNES: Nonlinear Solvers

Note: This chapter is being cleaned up by Jed Brown. Contributions are welcome.

The solution of large-scale nonlinear problems pervades many facets of computational science and demands robust and flexible solution strategies. The `SNES` library of PETSc provides a powerful suite of data-structure-neutral numerical routines for such problems. Built on top of the linear solvers and data structures discussed in preceding chapters, `SNES` enables the user to easily customize the nonlinear solvers according to the application at hand. Also, the `SNES` interface is *identical* for the uniprocess and parallel cases; the only difference in the parallel version is that each process typically forms only its local contribution to various matrices and vectors.

The `SNES` class includes methods for solving systems of nonlinear equations of the form

$$\mathbf{F}(\mathbf{x}) = 0, \tag{2.3}$$

where $\mathbf{F} : \Re^n \rightarrow \Re^n$. Newton-like methods provide the core of the package, including both line search and trust region techniques. A suite of nonlinear Krylov methods and methods based upon problem decomposition are also included. The solvers are discussed further in *The Nonlinear Solvers*. Following the PETSc design philosophy, the interfaces to the various solvers are all virtually identical. In addition, the `SNES` software is completely flexible, so that the user can at runtime change any facet of the solution process.

PETSc's default method for solving the nonlinear equation is Newton's method. The general form of the n-dimensional Newton's method for solving (2.3) is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k), \quad k = 0, 1, \dots, \tag{2.4}$$

where $\mathbf{x}_0$ is an initial approximation to the solution and $\mathbf{J}(\mathbf{x}_k) = \mathbf{F}'(\mathbf{x}_k)$, the Jacobian, is nonsingular at each iteration. In practice, the Newton iteration (2.4) is implemented by the following two steps:

1.(Approximately) solve $\mathbf{J}(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{F}(\mathbf{x}_k)$.
2.Update $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \Delta\mathbf{x}_k$.

Other defect-correction algorithms can be implemented by using different choices for $J(\mathbf{x}_k)$.

2.4.1 Basic SNES Usage

In the simplest usage of the nonlinear solvers, the user must merely provide a C, C++, or Fortran routine to evaluate the nonlinear function (2.3). The corresponding Jacobian matrix can be approximated with finite differences. For codes that are typically more efficient and accurate, the user can provide a routine to compute the Jacobian; details regarding these application-provided routines are discussed below. To provide an overview of the use of the nonlinear solvers, browse the concrete example in *ex1.c* or skip ahead to the discussion.

Listing: `src/snes/tutorials/ex1.c`

```
static char help[] = "Newton's method for a two-variable system, sequential.\n\n";

/*T
   Concepts: SNES^basic example
T*/



/*
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petscsys.h       - base PETSc routines   petscvec.h - vectors
     petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace methods
     petscviewer.h - viewers               petscpc.h  - preconditioners
     petscksp.h   - linear solvers
*/
```

(continues on next page)

(continued from previous page)

```
/*F
This examples solves either
\begin{equation}
  F\genfrac{(}{)}{0pt}{}{x_0}{x_1} = \genfrac{(}{)}{0pt}{}{x^2_0 + x_0 x_1 - 3}{x_0 x_
↪1 + x^2_1 - 6}
\end{equation}
or if the {\tt -hard} options is given
\begin{equation}
  F\genfrac{(}{)}{0pt}{}{x_0}{x_1} = \genfrac{(}{)}{0pt}{}{\sin(3 x_0) + x_0}{x_1}
\end{equation}
F*/
#include <petscsnes.h>

/*
   User-defined routines
*/
extern PetscErrorCode FormJacobian1(SNES,Vec,Mat,Mat,void*);
extern PetscErrorCode FormFunction1(SNES,Vec,Vec,void*);
extern PetscErrorCode FormJacobian2(SNES,Vec,Mat,Mat,void*);
extern PetscErrorCode FormFunction2(SNES,Vec,Vec,void*);

int main(int argc,char **argv)
{
  SNES           snes;         /* nonlinear solver context */
  KSP            ksp;          /* linear solver context */
  PC             pc;           /* preconditioner context */
  Vec            x,r;          /* solution, residual vectors */
  Mat            J;            /* Jacobian matrix */
  PetscErrorCode ierr;
  PetscMPIInt    size;
  PetscScalar    pfive = .5,*xx;
  PetscBool      flg;

  ierr = PetscInitialize(&argc,&argv,(char*)0,help);if (ierr) return ierr;
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size > 1) SETERRQ(PETSC_COMM_WORLD,PETSC_ERR_SUP,"Example is only for␣
↪sequential runs");

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  ierr = SNESCreate(PETSC_COMM_WORLD,&snes);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create matrix and vector data structures; set corresponding routines
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Create vectors for solution and nonlinear function
  */
  ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,2);CHKERRQ(ierr);
  ierr = VecSetFromOptions(x);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&r);CHKERRQ(ierr);

  /*
     Create Jacobian matrix data structure
```

(continues on next page)

(continued from previous page)

```
*/
ierr = MatCreate(PETSC_COMM_WORLD,&J);CHKERRQ(ierr);
ierr = MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,2,2);CHKERRQ(ierr);
ierr = MatSetFromOptions(J);CHKERRQ(ierr);
ierr = MatSetUp(J);CHKERRQ(ierr);

ierr = PetscOptionsHasName(NULL,NULL,"-hard",&flg);CHKERRQ(ierr);
if (!flg) {
  /*
   Set function evaluation routine and vector.
  */
  ierr = SNESSetFunction(snes,r,FormFunction1,NULL);CHKERRQ(ierr);

  /*
   Set Jacobian matrix data structure and Jacobian evaluation routine
  */
  ierr = SNESSetJacobian(snes,J,J,FormJacobian1,NULL);CHKERRQ(ierr);
} else {
  ierr = SNESSetFunction(snes,r,FormFunction2,NULL);CHKERRQ(ierr);
  ierr = SNESSetJacobian(snes,J,J,FormJacobian2,NULL);CHKERRQ(ierr);
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Customize nonlinear solver; set runtime options
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
   Set linear solver defaults for this problem. By extracting the
   KSP and PC contexts from the SNES context, we can then
   directly call any KSP and PC routines to set various options.
*/
ierr = SNESGetKSP(snes,&ksp);CHKERRQ(ierr);
ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = PCSetType(pc,PCNONE);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-4,PETSC_DEFAULT,PETSC_DEFAULT,20);CHKERRQ(ierr);

/*
   Set SNES/KSP/KSP/PC runtime options, e.g.,
       -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
   These options will override those specified above as long as
   SNESSetFromOptions() is called _after_ any other customization
   routines.
*/
ierr = SNESSetFromOptions(snes);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Evaluate initial guess; then solve nonlinear system
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
if (!flg) {
  ierr = VecSet(x,pfive);CHKERRQ(ierr);
} else {
  ierr  = VecGetArray(x,&xx);CHKERRQ(ierr);
  xx[0] = 2.0; xx[1] = 3.0;
  ierr  = VecRestoreArray(x,&xx);CHKERRQ(ierr);
}
/*
   Note: The user should initialize the vector, x, with the initial guess
```

(continues on next page)

(continued from previous page)

```
     for the nonlinear solver prior to calling SNESSolve().  In particular,
     to employ an initial guess of zero, the user should explicitly set
     this vector to zero by calling VecSet().
  */

  ierr = SNESSolve(snes,NULL,x);CHKERRQ(ierr);
  if (flg) {
    Vec f;
    ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
    ierr = SNESGetFunction(snes,&f,0,0);CHKERRQ(ierr);
    ierr = VecView(r,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
  }


  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = VecDestroy(&x);CHKERRQ(ierr); ierr = VecDestroy(&r);CHKERRQ(ierr);
  ierr = MatDestroy(&J);CHKERRQ(ierr); ierr = SNESDestroy(&snes);CHKERRQ(ierr);
  ierr = PetscFinalize();
  return ierr;
}
/* ------------------------------------------------------------------- */
/*
   FormFunction1 - Evaluates nonlinear function, F(x).

   Input Parameters:
.  snes - the SNES context
.  x    - input vector
.  ctx  - optional user-defined context

   Output Parameter:
.  f - function vector
 */
PetscErrorCode FormFunction1(SNES snes,Vec x,Vec f,void *ctx)
{
  PetscErrorCode    ierr;
  const PetscScalar *xx;
  PetscScalar       *ff;

  /*
   Get pointers to vector data.
      - For default PETSc vectors, VecGetArray() returns a pointer to
        the data array.  Otherwise, the routine is implementation dependent.
      - You MUST call VecRestoreArray() when you no longer need access to
        the array.
   */
  ierr = VecGetArrayRead(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);

  /* Compute function */
  ff[0] = xx[0]*xx[0] + xx[0]*xx[1] - 3.0;
  ff[1] = xx[0]*xx[1] + xx[1]*xx[1] - 6.0;
```

(continues on next page)

(continued from previous page)

```
  /* Restore vectors */
  ierr = VecRestoreArrayRead(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
  return 0;
}
/* ------------------------------------------------------------------- */
/*
   FormJacobian1 - Evaluates Jacobian matrix.

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  dummy - optional user-defined context (not used here)

   Output Parameters:
.  jac - Jacobian matrix
.  B - optionally different preconditioning matrix
.  flag - flag indicating matrix structure
*/
PetscErrorCode FormJacobian1(SNES snes,Vec x,Mat jac,Mat B,void *dummy)
{
  const PetscScalar *xx;
  PetscScalar       A[4];
  PetscErrorCode    ierr;
  PetscInt          idx[2] = {0,1};

  /*
     Get pointer to vector data
  */
  ierr = VecGetArrayRead(x,&xx);CHKERRQ(ierr);

  /*
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
        the matrix at once.
  */
  A[0]  = 2.0*xx[0] + xx[1]; A[1] = xx[0];
  A[2]  = xx[1]; A[3] = xx[0] + 2.0*xx[1];
  ierr  = MatSetValues(B,2,idx,2,idx,A,INSERT_VALUES);CHKERRQ(ierr);

  /*
     Restore vector
  */
  ierr = VecRestoreArrayRead(x,&xx);CHKERRQ(ierr);

  /*
     Assemble matrix
  */
  ierr = MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  if (jac != B) {
    ierr = MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  }
  return 0;
}
```

(continues on next page)

(continued from previous page)

```
/* ------------------------------------------------------------------- */
PetscErrorCode FormFunction2(SNES snes,Vec x,Vec f,void *dummy)
{
  PetscErrorCode    ierr;
  const PetscScalar *xx;
  PetscScalar       *ff;

  /*
     Get pointers to vector data.
       - For default PETSc vectors, VecGetArray() returns a pointer to
         the data array.  Otherwise, the routine is implementation dependent.
       - You MUST call VecRestoreArray() when you no longer need access to
         the array.
  */
  ierr = VecGetArrayRead(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);

  /*
     Compute function
  */
  ff[0] = PetscSinScalar(3.0*xx[0]) + xx[0];
  ff[1] = xx[1];

  /*
     Restore vectors
  */
  ierr = VecRestoreArrayRead(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
  return 0;
}
/* ------------------------------------------------------------------- */
PetscErrorCode FormJacobian2(SNES snes,Vec x,Mat jac,Mat B,void *dummy)
{
  const PetscScalar *xx;
  PetscScalar       A[4];
  PetscErrorCode    ierr;
  PetscInt          idx[2] = {0,1};

  /*
     Get pointer to vector data
  */
  ierr = VecGetArrayRead(x,&xx);CHKERRQ(ierr);

  /*
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
        the matrix at once.
  */
  A[0]  = 3.0*PetscCosScalar(3.0*xx[0]) + 1.0; A[1] = 0.0;
  A[2]  = 0.0;                     A[3] = 1.0;
  ierr  = MatSetValues(B,2,idx,2,idx,A,INSERT_VALUES);CHKERRQ(ierr);

  /*
     Restore vector
  */
```

(continues on next page)

(continued from previous page)

```
  ierr = VecRestoreArrayRead(x,&xx);CHKERRQ(ierr);

  /*
     Assemble matrix
  */
  ierr = MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  if (jac != B) {
    ierr = MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  }
  return 0;
}
```

To create a `SNES` solver, one must first call `SNESCreate()` as follows:

```
SNESCreate(MPI_Comm comm,SNES *snes);
```

The user must then set routines for evaluating the residual function (2.3) and its associated Jacobian matrix, as discussed in the following sections.

To choose a nonlinear solution method, the user can either call

```
SNESSetType(SNES snes,SNESType method);
```

or use the option `-snes_type <method>`, where details regarding the available methods are presented in *The Nonlinear Solvers*. The application code can take complete control of the linear and nonlinear techniques used in the Newton-like method by calling

```
SNESSetFromOptions(snes);
```

This routine provides an interface to the PETSc options database, so that at runtime the user can select a particular nonlinear solver, set various parameters and customized routines (e.g., specialized line search variants), prescribe the convergence tolerance, and set monitoring routines. With this routine the user can also control all linear solver options in the `KSP`, and `PC` modules, as discussed in *KSP: Linear System Solvers*.

After having set these routines and options, the user solves the problem by calling

```
SNESSolve(SNES snes,Vec b,Vec x);
```

where `x` should be initialized to the initial guess before calling and contains the solution on return. In particular, to employ an initial guess of zero, the user should explicitly set this vector to zero by calling `VecZeroEntries(x)`. Finally, after solving the nonlinear system (or several systems), the user should destroy the `SNES` context with

```
SNESDestroy(SNES *snes);
```

Nonlinear Function Evaluation

When solving a system of nonlinear equations, the user must provide a a residual function (2.3), which is set using

```
SNESSetFunction(SNES snes,Vec f,PetscErrorCode (*FormFunction)(SNES snes,Vec x,Vec f,
↪void *ctx),void *ctx);
```

The argument `f` is an optional vector for storing the solution; pass `NULL` to have the `SNES` allocate it for you. The argument `ctx` is an optional user-defined context, which can store any private, application-specific data required by the function evaluation routine; `NULL` should be used if such information is not needed. In C and C++, a user-defined context is merely a structure in which various objects can be stashed; in Fortran a user context can be an integer array that contains both parameters and pointers to PETSc objects. SNES Tutorial ex5 and SNES Tutorial ex5f give examples of user-defined application contexts in C and Fortran, respectively.

Jacobian Evaluation

The user must also specify a routine to form some approximation of the Jacobian matrix, `A`, at the current iterate, `x`, as is typically done with

```
SNESSetJacobian(SNES snes,Mat Amat,Mat Pmat,PetscErrorCode (*FormJacobian)(SNES snes,
↪Vec x,Mat A,Mat B,void *ctx),void *ctx);
```

The arguments of the routine `FormJacobian()` are the current iterate, `x`; the (approximate) Jacobian matrix, `Amat`; the matrix from which the preconditioner is constructed, `Pmat` (which is usually the same as `Amat`); and an optional user-defined Jacobian context, `ctx`, for application-specific data. Note that the `SNES` solvers are all data-structure neutral, so the full range of PETSc matrix formats (including "matrix-free" methods) can be used. *Matrices* discusses information regarding available matrix formats and options, while *Matrix-Free Methods* focuses on matrix-free methods in `SNES`. We briefly touch on a few details of matrix usage that are particularly important for efficient use of the nonlinear solvers.

A common usage paradigm is to assemble the problem Jacobian in the preconditioner storage `B`, rather than `A`. In the case where they are identical, as in many simulations, this makes no difference. However, it allows us to check the analytic Jacobian we construct in `FormJacobian()` by passing the `-snes_mf_operator` flag. This causes PETSc to approximate the Jacobian using finite differencing of the function evaluation (discussed in *Finite Difference Jacobian Approximations*), and the analytic Jacobian becomes merely the preconditioner. Even if the analytic Jacobian is incorrect, it is likely that the finite difference approximation will converge, and thus this is an excellent method to verify the analytic Jacobian. Moreover, if the analytic Jacobian is incomplete (some terms are missing or approximate), `-snes_mf_operator` may be used to obtain the exact solution, where the Jacobian approximation has been transferred to the preconditioner.

One such approximate Jacobian comes from "Picard linearization" which writes the nonlinear system as

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b} = 0$$

where $\mathbf{A}(\mathbf{x})$ usually contains the lower-derivative parts of the equation. For example, the nonlinear diffusion problem

$$-\nabla \cdot (\kappa(u)\nabla u) = 0$$

would be linearized as

$$A(u)v \simeq -\nabla \cdot (\kappa(u)\nabla v).$$

Usually this linearization is simpler to implement than Newton and the linear problems are somewhat easier to solve. In addition to using `-snes_mf_operator` with this approximation to the Jacobian, the Picard iterative procedure can be performed by defining $\mathbf{J}(\mathbf{x})$ to be $\mathbf{A}(\mathbf{x})$. Sometimes this iteration exhibits better global convergence than Newton linearization.

During successive calls to `FormJacobian()`, the user can either insert new matrix contexts or reuse old ones, depending on the application requirements. For many sparse matrix formats, reusing the old space (and merely changing the matrix elements) is more efficient; however, if the matrix structure completely changes, creating an entirely new matrix context may be preferable. Upon subsequent calls to the `FormJacobian()` routine, the user may wish to reinitialize the matrix entries to zero by calling `MatZeroEntries()`. See *Other Matrix Operations* for details on the reuse of the matrix context.

The directory `${PETSC_DIR}/src/snes/tutorials` provides a variety of examples.

2.4.2 The Nonlinear Solvers

As summarized in Table *PETSc Nonlinear Solvers*, `SNES` includes several Newton-like nonlinear solvers based on line search techniques and trust region methods. Also provided are several nonlinear Krylov methods, as well as nonlinear methods involving decompositions of the problem.

Each solver may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. A complete list can be found by consulting the manual pages or by running a program with the `-help` option; we discuss just a few in the sections below.

Table 2.7: PETSc Nonlinear Solvers

Method	SNESType	Options Name	Default Line Search
Line Search Newton	`SNESNEWTONLS`	`newtonls`	`SNESLINESEARCHBT`
Trust region Newton	`SNESNEWTONTR`	`newtontr`	—
Nonlinear Richardson	`SNESNRICHARDSON`	`nrichardson`	`SNESLINESEARCHL2`
Nonlinear CG	`SNESNCG`	`ncg`	`SNESLINESEARCHCP`
Nonlinear GMRES	`SNESNGMRES`	`ngmres`	`SNESLINESEARCHL2`
Quasi-Newton	`SNESQN`	`qn`	see *PETSc quasi-Newton solvers*
Full Approximation Scheme	`SNESFAS`	`fas`	—
Nonlinear ASM	`SNESNASM`	`nasm`	–
ASPIN	`SNESASPIN`	`aspin`	`SNESLINESEARCHBT`
Nonlinear Gauss-Seidel	`SNESNGS`	`ngs`	–
Anderson Mixing	`SNESANDERSON`	`anderson`	–
Newton with constraints (1)	`SNESVINEWTONRSLS`	`vinewtonrsls`	`SNESLINESEARCHBT`
Newton with constraints (2)	`SNESVINEWTONSSLS`	`vinewtonssls`	`SNESLINESEARCHBT`
Multi-stage Smoothers	`SNESMS`	`ms`	–
Composite	`SNESCOMPOSITE`	`composite`	–
Linear solve only	`SNESKSPONLY`	`ksponly`	–
Python Shell	`SNESPYTHON`	`python`	–
Shell (user-defined)	`SNESSHELL`	`shell`	–

Line Search Newton

The method `SNESNEWTONLS` (`-snes_type newtonls`) provides a line search Newton method for solving systems of nonlinear equations. By default, this technique employs cubic backtracking [DS83]. Alternative line search techniques are listed in Table *PETSc Line Search Methods*.

Table 2.8: PETSc Line Search Methods

Line Search	**SNESLineSearchType**	**Options Name**
Backtracking	`SNESLINESEARCHBT`	`bt`
(damped) step	`SNESLINESEARCHBASIC`	`basic`
L2-norm Minimization	`SNESLINESEARCHL2`	`l2`
Critical point	`SNESLINESEARCHCP`	`cp`
Shell	`SNESLINESEARCHSHELL`	`shell`

Every `SNES` has a line search context of type `SNESLineSearch` that may be retrieved using

```
SNESGetLineSearch(SNES snes,SNESLineSearch *ls);.
```

There are several default options for the line searches. The order of polynomial approximation may be set with `-snes_linesearch_order` or

```
SNESLineSearchSetOrder(SNESLineSearch ls, PetscInt order);
```

for instance, 2 for quadratic or 3 for cubic. Sometimes, it may not be necessary to monitor the progress of the nonlinear iteration. In this case, `-snes_linesearch_norms` or

```
SNESLineSearchSetComputeNorms(SNESLineSearch ls,PetscBool norms);
```

may be used to turn off function, step, and solution norm computation at the end of the linesearch.

The default line search for the line search Newton method, `SNESLINESEARCHBT` involves several parameters, which are set to defaults that are reasonable for many applications. The user can override the defaults by using the following options:

- `-snes_linesearch_alpha <alpha>`
- `-snes_linesearch_maxstep <max>`
- `-snes_linesearch_minlambda <tol>`

Besides the backtracking linesearch, there are `SNESLINESEARCHL2`, which uses a polynomial secant minimization of $||F(x)||_2$, and `SNESLINESEARCHCP`, which minimizes $F(x) \cdot Y$ where Y is the search direction. These are both potentially iterative line searches, which may be used to find a better-fitted steplength in the case where a single secant search is not sufficient. The number of iterations may be set with `-snes_linesearch_max_it`. In addition, the convergence criteria of the iterative line searches may be set using function tolerances `-snes_linesearch_rtol` and `-snes_linesearch_atol`, and steplength tolerance `snes_linesearch_ltol`.

Custom line search types may either be defined using `SNESLineSearchShell`, or by creating a custom user line search type in the model of the preexisting ones and register it using

```
SNESLineSearchRegister(const char sname[],PetscErrorCode (*function)(SNESLineSearch));
↪.
```

Trust Region Methods

The trust region method in `SNES` for solving systems of nonlinear equations, `SNESNEWTONTR` (`-snes_type newtontr`), is taken from the MINPACK project [MoreSGH84]. Several parameters can be set to control the variation of the trust region size during the solution process. In particular, the user can control the initial trust region radius, computed by

$$\Delta = \Delta_0 \|F_0\|_2,$$

by setting Δ_0 via the option `-snes_tr_delta0 <delta0>`.

Nonlinear Krylov Methods

A number of nonlinear Krylov methods are provided, including Nonlinear Richardson, conjugate gradient, GMRES, and Anderson Mixing. These methods are described individually below. They are all instrumental to PETSc's nonlinear preconditioning.

Nonlinear Richardson. The nonlinear Richardson iteration merely takes the form of a line search-damped fixed-point iteration of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda \mathbf{F}(\mathbf{x}_k), \quad k = 0, 1, \ldots,$$

where the default linesearch is `SNESLINESEARCHL2`. This simple solver is mostly useful as a nonlinear smoother, or to provide line search stabilization to an inner method.

Nonlinear Conjugate Gradients. Nonlinear CG is equivalent to linear CG, but with the steplength determined by line search (`SNESLINESEARCHCP` by default). Five variants (Fletcher-Reed, Hestenes-Steifel, Polak-Ribiere-Polyak, Dai-Yuan, and Conjugate Descent) are implemented in PETSc and may be chosen using

```
SNESNCGSetType(SNES snes, SNESNCGType btype);
```

Anderson Mixing and Nonlinear GMRES Methods. Nonlinear GMRES and Anderson Mixing methods combine the last m iterates, plus a new fixed-point iteration iterate, into a residual-minimizing new iterate.

Quasi-Newton Methods

Quasi-Newton methods store iterative rank-one updates to the Jacobian instead of computing it directly. Three limited-memory quasi-Newton methods are provided, L-BFGS, which are described in Table *PETSc quasi-Newton solvers*. These all are encapsulated under `-snes_type qn` and may be changed with `snes_qn_type`. The default is L-BFGS, which provides symmetric updates to an approximate Jacobian. This iteration is similar to the line search Newton methods.

Table 2.9: PETSc quasi-Newton solvers

QN Method	SNESQNType	Options Name	Default Line Search
L-BFGS	`SNES_QN_LBFGS`	`lbfgs`	`SNESLINESEARCHCP`
"Good" Broyden	`SNES_QN_BROYDEN`	`broyden`	`SNESLINESEARCHBASIC`
"Bad" Broyden	`SNES_QN_BADBROYEN`	`badbroyden`	`SNESLINESEARCHL2`

One may also control the form of the initial Jacobian approximation with

```
SNESQNSetScaleType(SNES snes, SNESQNScaleType stype);
```

and the restart type with

```
SNESQNSetRestartType(SNES snes, SNESQNRestartType rtype);
```

The Full Approximation Scheme

The Full Approximation Scheme is a nonlinear multigrid correction. At each level, there is a recursive cycle control `SNES` instance, and either one or two nonlinear solvers as smoothers (up and down). Problems set up using the `SNES` `DMDA` interface are automatically coarsened. FAS differs slightly from `PCMG`, in that the hierarchy is constructed recursively. However, much of the interface is a one-to-one map. We describe the "get" operations here, and it can be assumed that each has a corresponding "set" operation. For instance, the number of levels in the hierarchy may be retrieved using

```
SNESFASGetLevels(SNES snes, PetscInt *levels);
```

There are four `SNESFAS` cycle types, `SNES_FAS_MULTIPLICATIVE`, `SNES_FAS_ADDITIVE`, `SNES_FAS_FULL`, and `SNES_FAS_KASKADE`. The type may be set with

```
SNESFASSetType(SNES snes,SNESFASType fastype);.
```

and the cycle type, 1 for V, 2 for W, may be set with

```
SNESFASSetCycles(SNES snes, PetscInt cycles);.
```

Much like the interface to `PCMG` described in *Multigrid Preconditioners*, there are interfaces to recover the various levels' cycles and smoothers. The level smoothers may be accessed with

```
SNESFASGetSmoother(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmootherUp(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmootherDown(SNES snes, PetscInt level, SNES *smooth);
```

and the level cycles with

```
SNESFASGetCycleSNES(SNES snes,PetscInt level,SNES *lsnes);.
```

Also akin to `PCMG`, the restriction and prolongation at a level may be acquired with

```
SNESFASGetInterpolation(SNES snes, PetscInt level, Mat *mat);
SNESFASGetRestriction(SNES snes, PetscInt level, Mat *mat);
```

In addition, FAS requires special restriction for solution-like variables, called injection. This may be set with

```
SNESFASGetInjection(SNES snes, PetscInt level, Mat *mat);.
```

The coarse solve context may be acquired with

```
SNESFASGetCoarseSolve(SNES snes, SNES *smooth);
```

Nonlinear Additive Schwarz

Nonlinear Additive Schwarz methods (NASM) take a number of local nonlinear subproblems, solves them independently in parallel, and combines those solutions into a new approximate solution.

```
SNESNASMSetSubdomains(SNES snes,PetscInt n,SNES subsnes[],VecScatter iscatter[],
↪VecScatter oscatter[],VecScatter gscatter[]);
```

allows for the user to create these local subdomains. Problems set up using the `SNES` `DMDA` interface are automatically decomposed. To begin, the type of subdomain updates to the whole solution are limited to two types borrowed from `PCASM`: `PC_ASM_BASIC`, in which the overlapping updates added. `PC_ASM_RESTRICT` updates in a nonoverlapping fashion. This may be set with

```
SNESNASMSetType(SNES snes,PCASMType type);.
```

`SNESASPIN` is a helper `SNES` type that sets up a nonlinearly preconditioned Newton's method using NASM as the preconditioner.

2.4.3 General Options

This section discusses options and routines that apply to all `SNES` solvers and problem classes. In particular, we focus on convergence tests, monitoring routines, and tools for checking derivative computations.

Convergence Tests

Convergence of the nonlinear solvers can be detected in a variety of ways; the user can even specify a customized test, as discussed below. Most of the nonlinear solvers use `SNESConvergenceTestDefault()`, however, `SNESNEWTONTR` uses a method-specific additional convergence test as well. The convergence tests involves several parameters, which are set by default to values that should be reasonable for a wide range of problems. The user can customize the parameters to the problem at hand by using some of the following routines and options.

One method of convergence testing is to declare convergence when the norm of the change in the solution between successive iterations is less than some tolerance, `stol`. Convergence can also be determined based on the norm of the function. Such a test can use either the absolute size of the norm, `atol`, or its relative decrease, `rtol`, from an initial guess. The following routine sets these parameters, which are used in many of the default `SNES` convergence tests:

```
SNESSetTolerances(SNES snes,PetscReal atol,PetscReal rtol,PetscReal stol, PetscInt␣
↪its,PetscInt fcts);
```

This routine also sets the maximum numbers of allowable nonlinear iterations, `its`, and function evaluations, `fcts`. The corresponding options database commands for setting these parameters are:

- `-snes_atol <atol>`
- `-snes_rtol <rtol>`
- `-snes_stol <stol>`
- `-snes_max_it <its>`
- `-snes_max_funcs <fcts>`

A related routine is `SNESGetTolerances()`.

Convergence tests for trust regions methods often use an additional parameter that indicates the minimum allowable trust region radius. The user can set this parameter with the option `-snes_trtol <trtol>` or with the routine

```
SNESSetTrustRegionTolerance(SNES snes,PetscReal trtol);
```

Users can set their own customized convergence tests in `SNES` by using the command

```
SNESSetConvergenceTest(SNES snes,PetscErrorCode (*test)(SNES snes,PetscInt it,
↪PetscReal xnorm, PetscReal gnorm,PetscReal f,SNESConvergedReason reason, void␣
↪*cctx),void *cctx,PetscErrorCode (*destroy)(void *cctx));
```

The final argument of the convergence test routine, `cctx`, denotes an optional user-defined context for private data. When solving systems of nonlinear equations, the arguments `xnorm`, `gnorm`, and `f` are the current iterate norm, current step norm, and function norm, respectively. `SNESConvergedReason` should be set positive for convergence and negative for divergence. See `include/petscsnes.h` for a list of values for `SNESConvergedReason`.

Convergence Monitoring

By default the `SNES` solvers run silently without displaying information about the iterations. The user can initiate monitoring with the command

```
SNESMonitorSet(SNES snes,PetscErrorCode (*mon)(SNES,PetscInt its,PetscReal norm,void*␣
↪mctx),void *mctx,PetscErrorCode (*monitordestroy)(void**));
```

The routine, `mon`, indicates a user-defined monitoring routine, where `its` and `mctx` respectively denote the iteration number and an optional user-defined context for private data for the monitor routine. The argument `norm` is the function norm.

The routine set by `SNESMonitorSet()` is called once after every successful step computation within the nonlinear solver. Hence, the user can employ this routine for any application-specific computations that should be done after the solution update. The option `-snes_monitor` activates the default `SNES` monitor routine, `SNESMonitorDefault()`, while `-snes_monitor_lg_residualnorm` draws a simple line graph of the residual norm's convergence.

One can cancel hardwired monitoring routines for `SNES` at runtime with `-snes_monitor_cancel`.

As the Newton method converges so that the residual norm is small, say 10^{-10}, many of the final digits printed with the `-snes_monitor` option are meaningless. Worse, they are different on different machines; due to different round-off rules used by, say, the IBM RS6000 and the Sun SPARC. This makes testing between different machines difficult. The option `-snes_monitor_short` causes PETSc to print fewer of the digits of the residual norm as it gets smaller; thus on most of the machines it will always print the same numbers making cross-process testing easier.

The routines

```
SNESGetSolution(SNES snes,Vec *x);
SNESGetFunction(SNES snes,Vec *r,void *ctx,int(**func)(SNES,Vec,Vec,void*));
```

return the solution vector and function vector from a `SNES` context. These routines are useful, for instance, if the convergence test requires some property of the solution or function other than those passed with routine arguments.

Checking Accuracy of Derivatives

Since hand-coding routines for Jacobian matrix evaluation can be error prone, `SNES` provides easy-to-use support for checking these matrices against finite difference versions. In the simplest form of comparison, users can employ the option `-snes_test_jacobian` to compare the matrices at several points. Although not exhaustive, this test will generally catch obvious problems. One can compare the elements of the two matrices by using the option `-snes_test_jacobian_view` , which causes the two matrices to be printed to the screen.

Another means for verifying the correctness of a code for Jacobian computation is running the problem with either the finite difference or matrix-free variant, `-snes_fd` or `-snes_mf`; see *Finite Difference Jacobian Approximations* or *Matrix-Free Methods*. If a problem converges well with these matrix approximations but not with a user-provided routine, the problem probably lies with the hand-coded matrix. See the note in *Jacobian Evaluation* about assembling your Jabobian in the "preconditioner" slot `Pmat`.

The correctness of user provided `MATSHELL` Jacobians in general can be checked with `MatShellTestMultTranspose()` and `MatShellTestMult()`.

The correctness of user provided `MATSHELL` Jacobians via `TSSetRHSJacobian()` can be checked with `TSRHSJacobianTestTranspose()` and `TSRHSJacobianTest()` that check the correction of the matrix-transpose vector product and the matrix-product. From the command line, these can be checked with

- `-ts_rhs_jacobian_test_mult_transpose`
- `-mat_shell_test_mult_transpose_view`
- `-ts_rhs_jacobian_test_mult`
- `-mat_shell_test_mult_view`

2.4.4 Inexact Newton-like Methods

Since exact solution of the linear Newton systems within (2.4) at each iteration can be costly, modifications are often introduced that significantly reduce these expenses and yet retain the rapid convergence of Newton's method. Inexact or truncated Newton techniques approximately solve the linear systems using an iterative scheme. In comparison with using direct methods for solving the Newton systems, iterative methods have the virtue of requiring little space for matrix storage and potentially saving significant computational work. Within the class of inexact Newton methods, of particular interest are Newton-Krylov methods, where the subsidiary iterative technique for solving the Newton system is chosen from the class of Krylov subspace projection methods. Note that at runtime the user can set any of the linear solver options discussed in *KSP: Linear System Solvers*, such as `-ksp_type <ksp_method>` and `-pc_type <pc_method>`, to set the Krylov subspace and preconditioner methods.

Two levels of iterations occur for the inexact techniques, where during each global or outer Newton iteration a sequence of subsidiary inner iterations of a linear solver is performed. Appropriate control of the accuracy to which the subsidiary iterative method solves the Newton system at each global iteration is critical, since these inner iterations determine the asymptotic convergence rate for inexact Newton techniques. While the Newton systems must be solved well enough to retain fast local convergence of the Newton's iterates, use of excessive inner iterations, particularly when $\|\mathbf{x}_k - \mathbf{x}_*\|$ is large, is neither necessary nor economical. Thus, the number of required inner iterations typically increases as the Newton process progresses, so that the truncated iterates approach the true Newton iterates.

A sequence of nonnegative numbers $\{\eta_k\}$ can be used to indicate the variable convergence criterion. In this case, when solving a system of nonlinear equations, the update step of the Newton process remains unchanged, and direct solution of the linear system is replaced by iteration on the system until the residuals

$$\mathbf{r}_k^{(i)} = \mathbf{F}'(\mathbf{x}_k)\Delta\mathbf{x}_k + \mathbf{F}(\mathbf{x}_k)$$

satisfy

$$\frac{\|\mathbf{r}_k^{(i)}\|}{\|\mathbf{F}(\mathbf{x}_k)\|} \leq \eta_k \leq \eta < 1.$$

Here $\mathbf{x}_0$ is an initial approximation of the solution, and $\| \cdot \|$ denotes an arbitrary norm in $\Re^n$.

By default a constant relative convergence tolerance is used for solving the subsidiary linear systems within the Newton-like methods of `SNES`. When solving a system of nonlinear equations, one can instead employ the techniques of Eisenstat and Walker [EW96] to compute η_k at each step of the nonlinear solver by using the option `-snes_ksp_ew_conv` . In addition, by adding one's own `KSP` convergence test (see *Convergence Tests*), one can easily create one's own, problem-dependent, inner convergence tests.

2.4.5 Matrix-Free Methods

The `SNES` class fully supports matrix-free methods. The matrices specified in the Jacobian evaluation routine need not be conventional matrices; instead, they can point to the data required to implement a particular matrix-free method. The matrix-free variant is allowed *only* when the linear systems are solved by an iterative method in combination with no preconditioning (`PCNONE` or `-pc_type none`), a user-provided preconditioner matrix, or a user-provided preconditioner shell (`PCSHELL`, discussed in *Preconditioners*); that is, obviously matrix-free methods cannot be used with a direct solver, approximate factorization, or other preconditioner which requires access to explicit matrix entries.

The user can create a matrix-free context for use within `SNES` with the routine

```
MatCreateSNESMF(SNES snes,Mat *mat);
```

This routine creates the data structures needed for the matrix-vector products that arise within Krylov space iterative methods [BS90] by employing the matrix type `MATSHELL`, discussed in *Matrix-Free Matrices*. The default `SNES` matrix-free approximations can also be invoked with the command `-snes_mf`. Or, one can retain the user-provided Jacobian preconditioner, but replace the user-provided Jacobian matrix with the default matrix free variant with the option `-snes_mf_operator`.

See also

```
MatCreateMFFD(Vec x, Mat *mat);
```

for users who need a matrix-free matrix but are not using `SNES`.

The user can set one parameter to control the Jacobian-vector product approximation with the command

```
MatMFFDSetFunctionError(Mat mat,PetscReal rerror);
```

The parameter `rerror` should be set to the square root of the relative error in the function evaluations, e_{rel}; the default is the square root of machine epsilon (about 10^{-8} in double precision), which assumes that the functions are evaluated to full floating-point precision accuracy. This parameter can also be set from the options database with `-snes_mf_err <err>`

In addition, `SNES` provides a way to register new routines to compute the differencing parameter (h); see the manual page for `MatMFFDSetType()` and `MatMFFDRegister()`. We currently provide two default routines accessible via `-snes_mf_type <default or wp>`. For the default approach there is one "tuning" parameter, set with

```
MatMFFDDSSetUmin(Mat mat,PetscReal umin);
```

This parameter, `umin` (or u_{min}), is a bit involved; its default is 10^{-6} . The Jacobian-vector product is approximated via the formula

$$F'(u)a \approx \frac{F(u+h*a)-F(u)}{h}$$

where h is computed via

$$h = e_{\text{rel}} \cdot \begin{cases} u^T a/\|a\|_2^2 & \text{if } |u^T a| > u_{\min}\|a\|_1 \\ u_{\min}\,\text{sign}(u^T a)\|a\|_1/\|a\|_2^2 & \text{otherwise.} \end{cases}$$

This approach is taken from Brown and Saad [BS90]. The parameter can also be set from the options database with `-snes_mf_umin <umin>`

The second approach, taken from Walker and Pernice, [PW98], computes h via

$$h = \frac{\sqrt{1+||u||}e_{rel}}{||a||}$$

This has no tunable parameters, but note that inside the nonlinear solve for the entire *linear* iterative process u does not change hence $\sqrt{1+||u||}$ need be computed only once. This information may be set with the options

```
MatMFFDWPSetComputeNormU(Mat mat,PetscBool );
```

or `-mat_mffd_compute_normu <true or false>`. This information is used to eliminate the redundant computation of these parameters, therefore reducing the number of collective operations and improving the efficiency of the application code.

It is also possible to monitor the differencing parameters h that are computed via the routines

```
MatMFFDSetHHistory(Mat,PetscScalar *,int);
MatMFFDResetHHistory(Mat,PetscScalar *,int);
MatMFFDGetH(Mat,PetscScalar *);
```

We include an explicit example of using matrix-free methods in *ex3.c*. Note that by using the option `-snes_mf` one can easily convert any `SNES` code to use a matrix-free Newton-Krylov method without a preconditioner. As shown in this example, `SNESSetFromOptions()` must be called *after* `SNESSetJacobian()` to enable runtime switching between the user-specified Jacobian and the default `SNES` matrix-free form.

Listing: `src/snes/tutorials/ex3.c`

```
static char help[] = "Newton methods to solve u'' + u^{2} = f in parallel.\n\
This example employs a user-defined monitoring routine and optionally a user-defined\
↪n\
routine to check candidate iterates produced by line search routines.\n\
The command line options include:\n\
  -pre_check_iterates : activate checking of iterates\n\
  -post_check_iterates : activate checking of iterates\n\
  -check_tol <tol>: set tolerance for iterate checking\n\
  -user_precond : activate a (trivial) user-defined preconditioner\n\n";

/*T
   Concepts: SNES^basic parallel example
   Concepts: SNES^setting a user-defined monitoring routine
   Processors: n
```

(continues on next page)

(continued from previous page)

```
T*/



/*
   Include "petscdm.h" so that we can use data management objects (DMs)
   Include "petscdmda.h" so that we can use distributed arrays (DMDAs).
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petscsys.h    - base PETSc routines
     petscvec.h    - vectors
     petscmat.h    - matrices
     petscis.h     - index sets
     petscksp.h    - Krylov subspace methods
     petscviewer.h - viewers
     petscpc.h     - preconditioners
     petscksp.h    - linear solvers
*/

#include <petscdm.h>
#include <petscdmda.h>
#include <petscsnes.h>

/*
   User-defined routines.
*/
PetscErrorCode FormJacobian(SNES,Vec,Mat,Mat,void*);
PetscErrorCode FormFunction(SNES,Vec,Vec,void*);
PetscErrorCode FormInitialGuess(Vec);
PetscErrorCode Monitor(SNES,PetscInt,PetscReal,void*);
PetscErrorCode PreCheck(SNESLineSearch,Vec,Vec,PetscBool*,void*);
PetscErrorCode PostCheck(SNESLineSearch,Vec,Vec,Vec,PetscBool*,PetscBool*,void*);
PetscErrorCode PostSetSubKSP(SNESLineSearch,Vec,Vec,Vec,PetscBool*,PetscBool*,void*);
PetscErrorCode MatrixFreePreconditioner(PC,Vec,Vec);

/*
   User-defined application context
*/
typedef struct {
  DM          da;      /* distributed array */
  Vec         F;       /* right-hand-side of PDE */
  PetscMPIInt rank;    /* rank of processor */
  PetscMPIInt size;    /* size of communicator */
  PetscReal   h;       /* mesh spacing */
  PetscBool   sjerr;   /* if or not to test jacobian domain error */
} ApplicationCtx;

/*
   User-defined context for monitoring
*/
typedef struct {
  PetscViewer viewer;
} MonitorCtx;

/*
   User-defined context for checking candidate iterates that are
```

(continues on next page)

(continued from previous page)

```
   determined by line search methods
*/
typedef struct {
  Vec            last_step;  /* previous iterate */
  PetscReal      tolerance;  /* tolerance for changes between successive iterates */
  ApplicationCtx *user;
} StepCheckCtx;

typedef struct {
  PetscInt its0; /* num of prevous outer KSP iterations */
} SetSubKSPCtx;

int main(int argc,char **argv)
{
  SNES           snes;                 /* SNES context */
  SNESLineSearch linesearch;           /* SNESLineSearch context */
  Mat            J;                    /* Jacobian matrix */
  ApplicationCtx ctx;                  /* user-defined context */
  Vec            x,r,U,F;              /* vectors */
  MonitorCtx     monP;                 /* monitoring context */
  StepCheckCtx   checkP;               /* step-checking context */
  SetSubKSPCtx   checkP1;
  PetscBool      pre_check,post_check,post_setsubksp; /* flag indicating whether we
↪'re checking candidate iterates */
  PetscScalar    xp,*FF,*UU,none = -1.0;
  PetscErrorCode ierr;
  PetscInt       its,N = 5,i,maxit,maxf,xs,xm;
  PetscReal      abstol,rtol,stol,norm;
  PetscBool      flg,viewinitial = PETSC_FALSE;


  ierr = PetscInitialize(&argc,&argv,(char*)0,help);if (ierr) return ierr;
  ierr  = MPI_Comm_rank(PETSC_COMM_WORLD,&ctx.rank);CHKERRQ(ierr);
  ierr  = MPI_Comm_size(PETSC_COMM_WORLD,&ctx.size);CHKERRQ(ierr);
  ierr  = PetscOptionsGetInt(NULL,NULL,"-n",&N,NULL);CHKERRQ(ierr);
  ctx.h = 1.0/(N-1);
  ctx.sjerr = PETSC_FALSE;
  ierr  = PetscOptionsGetBool(NULL,NULL,"-test_jacobian_domain_error",&ctx.sjerr,
↪NULL);CHKERRQ(ierr);
  ierr  = PetscOptionsGetBool(NULL,NULL,"-view_initial",&viewinitial,NULL);
↪CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = SNESCreate(PETSC_COMM_WORLD,&snes);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create vector data structures; set function evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Create distributed array (DMDA) to manage parallel grid and vectors
  */
  ierr = DMDACreate1d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,N,1,1,NULL,&ctx.da);
↪CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  ierr = DMSetFromOptions(ctx.da);CHKERRQ(ierr);
  ierr = DMSetUp(ctx.da);CHKERRQ(ierr);

  /*
     Extract global and local vectors from DMDA; then duplicate for remaining
     vectors that are the same types
  */
  ierr = DMCreateGlobalVector(ctx.da,&x);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&r);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&F);CHKERRQ(ierr); ctx.F = F;
  ierr = VecDuplicate(x,&U);CHKERRQ(ierr);

  /*
     Set function evaluation routine and vector.  Whenever the nonlinear
     solver needs to compute the nonlinear function, it will call this
     routine.
      - Note that the final routine argument is the user-defined
        context that provides application-specific data for the
        function evaluation routine.
  */
  ierr = SNESSetFunction(snes,r,FormFunction,&ctx);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create matrix data structure; set Jacobian evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = MatCreate(PETSC_COMM_WORLD,&J);CHKERRQ(ierr);
  ierr = MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,N,N);CHKERRQ(ierr);
  ierr = MatSetFromOptions(J);CHKERRQ(ierr);
  ierr = MatSeqAIJSetPreallocation(J,3,NULL);CHKERRQ(ierr);
  ierr = MatMPIAIJSetPreallocation(J,3,NULL,3,NULL);CHKERRQ(ierr);

  /*
     Set Jacobian matrix data structure and default Jacobian evaluation
     routine.  Whenever the nonlinear solver needs to compute the
     Jacobian matrix, it will call this routine.
      - Note that the final routine argument is the user-defined
        context that provides application-specific data for the
        Jacobian evaluation routine.
  */
  ierr = SNESSetJacobian(snes,J,J,FormJacobian,&ctx);CHKERRQ(ierr);

  /*
     Optionally allow user-provided preconditioner
   */
  ierr = PetscOptionsHasName(NULL,NULL,"-user_precond",&flg);CHKERRQ(ierr);
  if (flg) {
    KSP ksp;
    PC  pc;
    ierr = SNESGetKSP(snes,&ksp);CHKERRQ(ierr);
    ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
    ierr = PCSetType(pc,PCSHELL);CHKERRQ(ierr);
    ierr = PCShellSetApply(pc,MatrixFreePreconditioner);CHKERRQ(ierr);
  }

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

(continues on next page)

(continued from previous page)

```
     Customize nonlinear solver; set runtime options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Set an optional user-defined monitoring routine
  */
  ierr = PetscViewerDrawOpen(PETSC_COMM_WORLD,0,0,0,0,400,400,&monP.viewer);
↪CHKERRQ(ierr);
  ierr = SNESMonitorSet(snes,Monitor,&monP,0);CHKERRQ(ierr);

  /*
     Set names for some vectors to facilitate monitoring (optional)
  */
  ierr = PetscObjectSetName((PetscObject)x,"Approximate Solution");CHKERRQ(ierr);
  ierr = PetscObjectSetName((PetscObject)U,"Exact Solution");CHKERRQ(ierr);

  /*
     Set SNES/KSP/KSP/PC runtime options, e.g.,
         -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
  */
  ierr = SNESSetFromOptions(snes);CHKERRQ(ierr);

  /*
     Set an optional user-defined routine to check the validity of candidate
     iterates that are determined by line search methods
  */
  ierr = SNESGetLineSearch(snes, &linesearch);CHKERRQ(ierr);
  ierr = PetscOptionsHasName(NULL,NULL,"-post_check_iterates",&post_check);
↪CHKERRQ(ierr);

  if (post_check) {
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Activating post step checking routine\n");
↪CHKERRQ(ierr);
    ierr = SNESLineSearchSetPostCheck(linesearch,PostCheck,&checkP);CHKERRQ(ierr);
    ierr = VecDuplicate(x,&(checkP.last_step));CHKERRQ(ierr);

    checkP.tolerance = 1.0;
    checkP.user      = &ctx;

    ierr = PetscOptionsGetReal(NULL,NULL,"-check_tol",&checkP.tolerance,NULL);
↪CHKERRQ(ierr);
  }

  ierr = PetscOptionsHasName(NULL,NULL,"-post_setsubksp",&post_setsubksp);
↪CHKERRQ(ierr);
  if (post_setsubksp) {
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Activating post setsubksp\n");CHKERRQ(ierr);
    ierr = SNESLineSearchSetPostCheck(linesearch,PostSetSubKSP,&checkP1);
↪CHKERRQ(ierr);
  }

  ierr = PetscOptionsHasName(NULL,NULL,"-pre_check_iterates",&pre_check);
↪CHKERRQ(ierr);
  if (pre_check) {
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Activating pre step checking routine\n");
↪CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
    ierr = SNESLineSearchSetPreCheck(linesearch,PreCheck,&checkP);CHKERRQ(ierr);
  }

  /*
     Print parameters used for convergence testing (optional) ... just
     to demonstrate this routine; this information is also printed with
     the option -snes_view
  */
  ierr = SNESGetTolerances(snes,&abstol,&rtol,&stol,&maxit,&maxf);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"atol=%g, rtol=%g, stol=%g, maxit=%D, maxf=%D\n
↪",(double)abstol,(double)rtol,(double)stol,maxit,maxf);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Initialize application:
     Store right-hand-side of PDE and exact solution
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Get local grid boundaries (for 1-dimensional DMDA):
       xs, xm - starting grid index, width of local grid (no ghost points)
  */
  ierr = DMDAGetCorners(ctx.da,&xs,NULL,NULL,&xm,NULL,NULL);CHKERRQ(ierr);

  /*
     Get pointers to vector data
  */
  ierr = DMDAVecGetArray(ctx.da,F,&FF);CHKERRQ(ierr);
  ierr = DMDAVecGetArray(ctx.da,U,&UU);CHKERRQ(ierr);

  /*
     Compute local vector entries
  */
  xp = ctx.h*xs;
  for (i=xs; i<xs+xm; i++) {
    FF[i] = 6.0*xp + PetscPowScalar(xp+1.e-12,6.0); /* +1.e-12 is to prevent 0^6 */
    UU[i] = xp*xp*xp;
    xp   += ctx.h;
  }

  /*
     Restore vectors
  */
  ierr = DMDAVecRestoreArray(ctx.da,F,&FF);CHKERRQ(ierr);
  ierr = DMDAVecRestoreArray(ctx.da,U,&UU);CHKERRQ(ierr);
  if (viewinitial) {
    ierr = VecView(U,0);CHKERRQ(ierr);
    ierr = VecView(F,0);CHKERRQ(ierr);
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Evaluate initial guess; then solve nonlinear system
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Note: The user should initialize the vector, x, with the initial guess
     for the nonlinear solver prior to calling SNESSolve().  In particular,
```

(continues on next page)

(continued from previous page)

```
     to employ an initial guess of zero, the user should explicitly set
     this vector to zero by calling VecSet().
  */
  ierr = FormInitialGuess(x);CHKERRQ(ierr);
  ierr = SNESSolve(snes,NULL,x);CHKERRQ(ierr);
  ierr = SNESGetIterationNumber(snes,&its);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Number of SNES iterations = %D\n",its);
↪CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Check solution and clean up
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Check the error
  */
  ierr = VecAXPY(x,none,U);CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %g Iterations %D\n",(double)norm,
↪its);CHKERRQ(ierr);
  if (ctx.sjerr) {
    SNESType       snestype;
    ierr = SNESGetType(snes,&snestype);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"SNES Type %s\n",snestype);CHKERRQ(ierr);
  }

  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  ierr = PetscViewerDestroy(&monP.viewer);CHKERRQ(ierr);
  if (post_check) {ierr = VecDestroy(&checkP.last_step);CHKERRQ(ierr);}
  ierr = VecDestroy(&x);CHKERRQ(ierr);
  ierr = VecDestroy(&r);CHKERRQ(ierr);
  ierr = VecDestroy(&U);CHKERRQ(ierr);
  ierr = VecDestroy(&F);CHKERRQ(ierr);
  ierr = MatDestroy(&J);CHKERRQ(ierr);
  ierr = SNESDestroy(&snes);CHKERRQ(ierr);
  ierr = DMDestroy(&ctx.da);CHKERRQ(ierr);
  ierr = PetscFinalize();
  return ierr;
}

/* ------------------------------------------------------------------- */
/*
   FormInitialGuess - Computes initial guess.

   Input/Output Parameter:
.  x - the solution vector
*/
PetscErrorCode FormInitialGuess(Vec x)
{
  PetscErrorCode ierr;
  PetscScalar    pfive = .50;

  PetscFunctionBeginUser;
```

(continues on next page)

(continued from previous page)

```
  ierr = VecSet(x,pfive);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   FormFunction - Evaluates nonlinear function, F(x).

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  ctx - optional user-defined context, as set by SNESSetFunction()

   Output Parameter:
.  f - function vector

   Note:
   The user-defined context can contain any application-specific
   data needed for the function evaluation.
*/
PetscErrorCode FormFunction(SNES snes,Vec x,Vec f,void *ctx)
{
  ApplicationCtx *user = (ApplicationCtx*) ctx;
  DM             da    = user->da;
  PetscScalar    *xx,*ff,*FF,d;
  PetscErrorCode ierr;
  PetscInt       i,M,xs,xm;
  Vec            xlocal;

  PetscFunctionBeginUser;
  ierr = DMGetLocalVector(da,&xlocal);CHKERRQ(ierr);
  /*
     Scatter ghost points to local vector, using the 2-step process
        DMGlobalToLocalBegin(), DMGlobalToLocalEnd().
     By placing code between these two statements, computations can
     be done while messages are in transition.
  */
  ierr = DMGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal);CHKERRQ(ierr);
  ierr = DMGlobalToLocalEnd(da,x,INSERT_VALUES,xlocal);CHKERRQ(ierr);

  /*
     Get pointers to vector data.
       - The vector xlocal includes ghost point; the vectors x and f do
         NOT include ghost points.
       - Using DMDAVecGetArray() allows accessing the values using global ordering
  */
  ierr = DMDAVecGetArray(da,xlocal,&xx);CHKERRQ(ierr);
  ierr = DMDAVecGetArray(da,f,&ff);CHKERRQ(ierr);
  ierr = DMDAVecGetArray(da,user->F,&FF);CHKERRQ(ierr);

  /*
     Get local grid boundaries (for 1-dimensional DMDA):
       xs, xm  - starting grid index, width of local grid (no ghost points)
  */
  ierr = DMDAGetCorners(da,&xs,NULL,NULL,&xm,NULL,NULL);CHKERRQ(ierr);
  ierr = DMDAGetInfo(da,NULL,&M,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
↪NULL);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  /*
     Set function values for boundary points; define local interior grid point range:
        xsi - starting interior grid index
        xei - ending interior grid index
  */
  if (xs == 0) { /* left boundary */
    ff[0] = xx[0];
    xs++;xm--;
  }
  if (xs+xm == M) {  /* right boundary */
    ff[xs+xm-1] = xx[xs+xm-1] - 1.0;
    xm--;
  }

  /*
     Compute function over locally owned part of the grid (interior points only)
  */
  d = 1.0/(user->h*user->h);
  for (i=xs; i<xs+xm; i++) ff[i] = d*(xx[i-1] - 2.0*xx[i] + xx[i+1]) + xx[i]*xx[i] -␣
↪FF[i];

  /*
     Restore vectors
  */
  ierr = DMDAVecRestoreArray(da,xlocal,&xx);CHKERRQ(ierr);
  ierr = DMDAVecRestoreArray(da,f,&ff);CHKERRQ(ierr);
  ierr = DMDAVecRestoreArray(da,user->F,&FF);CHKERRQ(ierr);
  ierr = DMRestoreLocalVector(da,&xlocal);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   FormJacobian - Evaluates Jacobian matrix.

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  dummy - optional user-defined context (not used here)

   Output Parameters:
.  jac - Jacobian matrix
.  B - optionally different preconditioning matrix
.  flag - flag indicating matrix structure
*/
PetscErrorCode FormJacobian(SNES snes,Vec x,Mat jac,Mat B,void *ctx)
{
  ApplicationCtx *user = (ApplicationCtx*) ctx;
  PetscScalar    *xx,d,A[3];
  PetscErrorCode ierr;
  PetscInt       i,j[3],M,xs,xm;
  DM             da = user->da;

  PetscFunctionBeginUser;
  if (user->sjerr) {
```

(continues on next page)

(continued from previous page)

```
    ierr = SNESSetJacobianDomainError(snes);CHKERRQ(ierr);
    PetscFunctionReturn(0);
  }
  /*
     Get pointer to vector data
  */
  ierr = DMDAVecGetArrayRead(da,x,&xx);CHKERRQ(ierr);
  ierr = DMDAGetCorners(da,&xs,NULL,NULL,&xm,NULL,NULL);CHKERRQ(ierr);

  /*
    Get range of locally owned matrix
  */
  ierr = DMDAGetInfo(da,NULL,&M,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
↪NULL);CHKERRQ(ierr);

  /*
     Determine starting and ending local indices for interior grid points.
     Set Jacobian entries for boundary points.
  */

  if (xs == 0) {  /* left boundary */
    i = 0; A[0] = 1.0;

    ierr = MatSetValues(jac,1,&i,1,&i,A,INSERT_VALUES);CHKERRQ(ierr);
    xs++;xm--;
  }
  if (xs+xm == M) { /* right boundary */
    i    = M-1;
    A[0] = 1.0;
    ierr = MatSetValues(jac,1,&i,1,&i,A,INSERT_VALUES);CHKERRQ(ierr);
    xm--;
  }

  /*
     Interior grid points
      - Note that in this case we set all elements for a particular
        row at once.
  */
  d = 1.0/(user->h*user->h);
  for (i=xs; i<xs+xm; i++) {
    j[0] = i - 1; j[1] = i; j[2] = i + 1;
    A[0] = A[2] = d; A[1] = -2.0*d + 2.0*xx[i];
    ierr = MatSetValues(jac,1,&i,3,j,A,INSERT_VALUES);CHKERRQ(ierr);
  }

  /*
     Assemble matrix, using the 2-step process:
       MatAssemblyBegin(), MatAssemblyEnd().
     By placing code between these two statements, computations can be
     done while messages are in transition.

     Also, restore vector.
  */

  ierr = MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = DMDAVecRestoreArrayRead(da,x,&xx);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  ierr = MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   Monitor - Optional user-defined monitoring routine that views the
   current iterate with an x-window plot. Set by SNESMonitorSet().

   Input Parameters:
   snes - the SNES context
   its - iteration number
   norm - 2-norm function value (may be estimated)
   ctx - optional user-defined context for private data for the
         monitor routine, as set by SNESMonitorSet()

   Note:
   See the manpage for PetscViewerDrawOpen() for useful runtime options,
   such as -nox to deactivate all x-window output.
 */
PetscErrorCode Monitor(SNES snes,PetscInt its,PetscReal fnorm,void *ctx)
{
  PetscErrorCode ierr;
  MonitorCtx     *monP = (MonitorCtx*) ctx;
  Vec            x;

  PetscFunctionBeginUser;
  ierr = PetscPrintf(PETSC_COMM_WORLD,"iter = %D,SNES Function norm %g\n",its,
↪(double)fnorm);CHKERRQ(ierr);
  ierr = SNESGetSolution(snes,&x);CHKERRQ(ierr);
  ierr = VecView(x,monP->viewer);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   PreCheck - Optional user-defined routine that checks the validity of
   candidate steps of a line search method.  Set by SNESLineSearchSetPreCheck().

   Input Parameters:
   snes - the SNES context
   xcurrent - current solution
   y - search direction and length

   Output Parameters:
   y         - proposed step (search direction and length) (possibly changed)
   changed_y - tells if the step has changed or not
 */
PetscErrorCode PreCheck(SNESLineSearch linesearch,Vec xcurrent,Vec y, PetscBool␣
↪*changed_y, void * ctx)
{
  PetscFunctionBeginUser;
  *changed_y = PETSC_FALSE;
  PetscFunctionReturn(0);
}
```

(continues on next page)

(continued from previous page)

```
/* ------------------------------------------------------------------- */
/*
   PostCheck - Optional user-defined routine that checks the validity of
   candidate steps of a line search method.  Set by SNESLineSearchSetPostCheck().

   Input Parameters:
   snes - the SNES context
   ctx  - optional user-defined context for private data for the
          monitor routine, as set by SNESLineSearchSetPostCheck()
   xcurrent - current solution
   y - search direction and length
   x    - the new candidate iterate

   Output Parameters:
   y    - proposed step (search direction and length) (possibly changed)
   x    - current iterate (possibly modified)

 */
PetscErrorCode PostCheck(SNESLineSearch linesearch,Vec xcurrent,Vec y,Vec x,PetscBool␣
↪ *changed_y,PetscBool  *changed_x, void * ctx)
{
  PetscErrorCode ierr;
  PetscInt       i,iter,xs,xm;
  StepCheckCtx   *check;
  ApplicationCtx *user;
  PetscScalar    *xa,*xa_last,tmp;
  PetscReal      rdiff;
  DM             da;
  SNES           snes;

  PetscFunctionBeginUser;
  *changed_x = PETSC_FALSE;
  *changed_y = PETSC_FALSE;

  ierr  = SNESLineSearchGetSNES(linesearch, &snes);CHKERRQ(ierr);
  check = (StepCheckCtx*)ctx;
  user  = check->user;
  ierr  = SNESGetIterationNumber(snes,&iter);CHKERRQ(ierr);

  /* iteration 1 indicates we are working on the second iteration */
  if (iter > 0) {
    da   = user->da;
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Checking candidate step at iteration %D with␣
↪tolerance %g\n",iter,(double)check->tolerance);CHKERRQ(ierr);

    /* Access local array data */
    ierr = DMDAVecGetArray(da,check->last_step,&xa_last);CHKERRQ(ierr);
    ierr = DMDAVecGetArray(da,x,&xa);CHKERRQ(ierr);
    ierr = DMDAGetCorners(da,&xs,NULL,NULL,&xm,NULL,NULL);CHKERRQ(ierr);

    /*
       If we fail the user-defined check for validity of the candidate iterate,
       then modify the iterate as we like.  (Note that the particular modification
       below is intended simply to demonstrate how to manipulate this data, not
       as a meaningful or appropriate choice.)
```

(continues on next page)

(continued from previous page)

```
    */
    for (i=xs; i<xs+xm; i++) {
      if (!PetscAbsScalar(xa[i])) rdiff = 2*check->tolerance;
      else rdiff = PetscAbsScalar((xa[i] - xa_last[i])/xa[i]);
      if (rdiff > check->tolerance) {
        tmp        = xa[i];
        xa[i]      = .5*(xa[i] + xa_last[i]);
        *changed_x = PETSC_TRUE;
        ierr       = PetscPrintf(PETSC_COMM_WORLD,"  Altering entry %D: x=%g, x_last=
↪%g, diff=%g, x_new=%g\n",i,(double)PetscAbsScalar(tmp),(double)PetscAbsScalar(xa_
↪last[i]),(double)rdiff,(double)PetscAbsScalar(xa[i]));CHKERRQ(ierr);
      }
    }
    ierr = DMDAVecRestoreArray(da,check->last_step,&xa_last);CHKERRQ(ierr);
    ierr = DMDAVecRestoreArray(da,x,&xa);CHKERRQ(ierr);
  }
  ierr = VecCopy(x,check->last_step);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   PostSetSubKSP - Optional user-defined routine that reset SubKSP options when␣
↪hierarchical bjacobi PC is used
   e.g,
     mpiexec -n 8 ./ex3 -nox -n 10000 -ksp_type fgmres -pc_type bjacobi -pc_bjacobi_
↪blocks 4 -sub_ksp_type gmres -sub_ksp_max_it 3 -post_setsubksp -sub_ksp_rtol 1.e-16
   Set by SNESLineSearchSetPostCheck().

   Input Parameters:
   linesearch - the LineSearch context
   xcurrent - current solution
   y - search direction and length
   x    - the new candidate iterate

   Output Parameters:
   y    - proposed step (search direction and length) (possibly changed)
   x    - current iterate (possibly modified)

 */
PetscErrorCode PostSetSubKSP(SNESLineSearch linesearch,Vec xcurrent,Vec y,Vec x,
↪PetscBool  *changed_y,PetscBool  *changed_x, void * ctx)
{
  PetscErrorCode ierr;
  SetSubKSPCtx   *check;
  PetscInt       iter,its,sub_its,maxit;
  KSP            ksp,sub_ksp,*sub_ksps;
  PC             pc;
  PetscReal      ksp_ratio;
  SNES           snes;

  PetscFunctionBeginUser;
  ierr    = SNESLineSearchGetSNES(linesearch, &snes);CHKERRQ(ierr);
  check   = (SetSubKSPCtx*)ctx;
  ierr    = SNESGetIterationNumber(snes,&iter);CHKERRQ(ierr);
  ierr    = SNESGetKSP(snes,&ksp);CHKERRQ(ierr);
```

(continues on next page)

(continued from previous page)

```
  ierr    = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
  ierr    = PCBJacobiGetSubKSP(pc,NULL,NULL,&sub_ksps);CHKERRQ(ierr);
  sub_ksp = sub_ksps[0];
  ierr    = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);      /* outer KSP␣
↪iteration number */
  ierr    = KSPGetIterationNumber(sub_ksp,&sub_its);CHKERRQ(ierr); /* inner KSP␣
↪iteration number */

  if (iter) {
    ierr      = PetscPrintf(PETSC_COMM_WORLD,"    ...PostCheck snes iteration %D, ksp_
↪it %D %D, subksp_it %D\n",iter,check->its0,its,sub_its);CHKERRQ(ierr);
    ksp_ratio = ((PetscReal)(its))/check->its0;
    maxit     = (PetscInt)(ksp_ratio*sub_its + 0.5);
    if (maxit < 2) maxit = 2;
    ierr = KSPSetTolerances(sub_ksp,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT,maxit);
↪CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"    ...ksp_ratio %g, new maxit %D\n\n",
↪(double)ksp_ratio,maxit);CHKERRQ(ierr);
  }
  check->its0 = its; /* save current outer KSP iteration number */
  PetscFunctionReturn(0);
}

/* ------------------------------------------------------------------- */
/*
   MatrixFreePreconditioner - This routine demonstrates the use of a
   user-provided preconditioner.  This code implements just the null
   preconditioner, which of course is not recommended for general use.

   Input Parameters:
+  pc - preconditioner
-  x - input vector

   Output Parameter:
.  y - preconditioned vector
*/
PetscErrorCode MatrixFreePreconditioner(PC pc,Vec x,Vec y)
{
  PetscErrorCode ierr;
  ierr = VecCopy(x,y);CHKERRQ(ierr);
  return 0;
}
```

Table *Jacobian Options* summarizes the various matrix situations that `SNES` supports. In particular, different linear system matrices and preconditioning matrices are allowed, as well as both matrix-free and application-provided preconditioners. If *ex3.c* is run with the options `-snes_mf` and `-user_precond` then it uses a matrix-free application of the matrix-vector multiple and a user provided matrix free Jacobian.

Table 2.10: Jacobian Options

Matrix Use	Conventional Matrix Formats	Matrix-free versions
Jacobian Matrix	Create matrix with `MatCreate()`*. Assemble matrix with user-defined routine †	Create matrix with `MatCreateShell()`. Use `MatShellSetOperation()` to set various matrix actions, or use `MatCreateMFFD()` or `MatCreateSNESMF()`.
Preconditioning Matrix	Create matrix with `MatCreate()`*. Assemble matrix with user-defined routine †	Use `SNESGetKSP()` and `KSPGetPC()` to access the `PC`, then use `PCSetType(pc, PCSHELL)` followed by `PCShellSetApply()`.

* Use either the generic `MatCreate()` or a format-specific variant such as `MatCreateAIJ()`.

† Set user-defined matrix formation routine with `SNESSetJacobian()` or with a `DM` variant such as `DMDASNESSetJacobianLocal()`

2.4.6 Finite Difference Jacobian Approximations

PETSc provides some tools to help approximate the Jacobian matrices efficiently via finite differences. These tools are intended for use in certain situations where one is unable to compute Jacobian matrices analytically, and matrix-free methods do not work well without a preconditioner, due to very poor conditioning. The approximation requires several steps:

- First, one colors the columns of the (not yet built) Jacobian matrix, so that columns of the same color do not share any common rows.
- Next, one creates a `MatFDColoring` data structure that will be used later in actually computing the Jacobian.
- Finally, one tells the nonlinear solvers of `SNES` to use the `SNESComputeJacobianDefaultColor()` routine to compute the Jacobians.

A code fragment that demonstrates this process is given below.

```
ISColoring    iscoloring;
MatFDColoring fdcoloring;
MatColoring   coloring;

/*
  This initializes the nonzero structure of the Jacobian. This is artificial
  because clearly if we had a routine to compute the Jacobian we wouldn't
  need to use finite differences.
*/
FormJacobian(snes,x, &J, &J, &user);

/*
   Color the matrix, i.e. determine groups of columns that share no common
  rows. These columns in the Jacobian can all be computed simultaneously.
*/
MatColoringCreate(J, &coloring);
MatColoringSetType(coloring,MATCOLORINGSL);
MatColoringSetFromOptions(coloring);
MatColoringApply(coloring, &iscoloring);
MatColoringDestroy(&coloring);
```

(continues on next page)

(continued from previous page)

```
/*
   Create the data structure that SNESComputeJacobianDefaultColor() uses
   to compute the actual Jacobians via finite differences.
*/
MatFDColoringCreate(J,iscoloring, &fdcoloring);
ISColoringDestroy(&iscoloring);
MatFDColoringSetFunction(fdcoloring,(PetscErrorCode (*)(void))FormFunction, &user);
MatFDColoringSetFromOptions(fdcoloring);

/*
  Tell SNES to use the routine SNESComputeJacobianDefaultColor()
  to compute Jacobians.
*/
SNESSetJacobian(snes,J,J,SNESComputeJacobianDefaultColor,fdcoloring);
```

Of course, we are cheating a bit. If we do not have an analytic formula for computing the Jacobian, then how do we know what its nonzero structure is so that it may be colored? Determining the structure is problem dependent, but fortunately, for most structured grid problems (the class of problems for which PETSc was originally designed) if one knows the stencil used for the nonlinear function one can usually fairly easily obtain an estimate of the location of nonzeros in the matrix. This is harder in the unstructured case, but one typically knows where the nonzero entries are from the mesh topology and distribution of degrees of freedom. If using `DMPlex` (*DMPlex: Unstructured Grids in PETSc*) for unstructured meshes, the nonzero locations will be identified in `DMCreateMatrix()` and the procedure above can be used. Most external packages for unstructured meshes have similar functionality.

One need not necessarily use a `MatColoring` object to determine a coloring. For example, if a grid can be colored directly (without using the associated matrix), then that coloring can be provided to `MatFDColoringCreate()`. Note that the user must always preset the nonzero structure in the matrix regardless of which coloring routine is used.

PETSc provides the following coloring algorithms, which can be selected using `MatColoringSetType()` or via the command line argument `-mat_coloring_type`.

Algorithm	MatColoringType	-mat_coloring_type	Parallel
smallest-last [MoreSGH84]	`MATCOLORINGSL`	`sl`	No
largest-first [MoreSGH84]	`MATCOLORINGLF`	`lf`	No
incidence-degree [MoreSGH84]	`MATCOLORINGID`	`id`	No
Jones-Plassmann [JP93]	`MATCOLORINGJP`	`jp`	Yes
Greedy	`MATCOLORINGGREEDY`	`greedy`	Yes
Natural (1 color per column)	`MATCOLORINGNATURAL`	`natural`	Yes
Power (A^k followed by 1-coloring)	`MATCOLORINGPOWER`	`power`	Yes

As for the matrix-free computation of Jacobians (*Matrix-Free Methods*), two parameters affect the accuracy of the finite difference Jacobian approximation. These are set with the command

```
MatFDColoringSetParameters(MatFDColoring fdcoloring,PetscReal rerror,PetscReal umin);
```

The parameter `rerror` is the square root of the relative error in the function evaluations, e_{rel}; the default is the square root of machine epsilon (about 10^{-8} in double precision), which assumes that the functions are evaluated approximately to floating-point precision accuracy. The second parameter, `umin`, is a bit more involved; its default is $10e^{-6}$. Column i of the Jacobian matrix (denoted by $F_{:i}$) is approximated by the formula

$$F'_{:i} \approx \frac{F(u + h * dx_i) - F(u)}{h}$$

where h is computed via:

$$h = e_{\text{rel}} \cdot \begin{cases} u_i & \text{if } |u_i| > u_{\min} \\ u_{\min} \cdot \operatorname{sign}(u_i) & \text{otherwise.} \end{cases}$$

for `MATMFFD_DS` or:

$$h = e_{\text{rel}} \sqrt{(}\|u\|)$$

for `MATMFFD_WP` (default). These parameters may be set from the options database with

```
-mat_fd_coloring_err <err>
-mat_fd_coloring_umin <umin>
-mat_fd_type <htype>
```

Note that `MatColoring` type `MATCOLORINGSL`, `MATCOLORINGLF`, and `MATCOLORINGID` are sequential algorithms. `MATCOLORINGJP` and `MATCOLORINGGREEDY` are parallel algorithms, although in practice they may create more colors than the sequential algorithms. If one computes the coloring `iscoloring` reasonably with a parallel algorithm or by knowledge of the discretization, the routine `MatFDColoringCreate()` is scalable. An example of this for 2D distributed arrays is given below that uses the utility routine `DMCreateColoring()`.

```
DMCreateColoring(da,IS_COLORING_GHOSTED, &iscoloring);
MatFDColoringCreate(J,iscoloring, &fdcoloring);
MatFDColoringSetFromOptions(fdcoloring);
ISColoringDestroy( &iscoloring);
```

Note that the routine `MatFDColoringCreate()` currently is only supported for the AIJ and BAIJ matrix formats.

2.4.7 Variational Inequalities

`SNES` can also solve variational inequalities with box constraints. These are nonlinear algebraic systems with additional inequality constraints on some or all of the variables: $Lu_i \le u_i \le Hu_i$. Some or all of the lower bounds may be negative infinity (indicated to PETSc with `SNES_VI_NINF`) and some or all of the upper bounds may be infinity (indicated by `SNES_VI_INF`). The command

```
SNESVISetVariableBounds(SNES,Vec Lu,Vec Hu);
```

is used to indicate that one is solving a variational inequality. The option `-snes_vi_monitor` turns on extra monitoring of the active set associated with the bounds and `-snes_vi_type` allows selecting from several VI solvers, the default is preferred.

2.4.8 Nonlinear Preconditioning

The mathematical framework of nonlinear preconditioning is explained in detail in [BKST15]. Nonlinear preconditioning in PETSc involves the use of an inner `SNES` instance to define the step for an outer `SNES` instance. The inner instance may be extracted using

```
SNESGetNPC(SNES snes,SNES *npc);
```

and passed run-time options using the `-npc_` prefix. Nonlinear preconditioning comes in two flavors: left and right. The side may be changed using `-snes_npc_side` or `SNESSetNPCSide()`. Left nonlinear preconditioning redefines the nonlinear function as the action of the nonlinear preconditioner $\mathbf{M}$;

$$\mathbf{F}_M(x) = \mathbf{M}(\mathbf{x}, \mathbf{b}) - \mathbf{x}.$$

Right nonlinear preconditioning redefines the nonlinear function as the function on the action of the nonlinear preconditioner;

$$\mathbf{F}(\mathbf{M}(\mathbf{x},\mathbf{b})) = \mathbf{b},$$

which can be interpreted as putting the preconditioner into "striking distance" of the solution by outer acceleration.

In addition, basic patterns of solver composition are available with the `SNESType` `SNESCOMPOSITE`. This allows for two or more `SNES` instances to be combined additively or multiplicatively. By command line, a set of `SNES` types may be given by comma separated list argument to `-snes_composite_sneses`. There are additive (`SNES_COMPOSITE_ADDITIVE`), additive with optimal damping (`SNES_COMPOSITE_ADDITIVEOPTIMAL`), and multiplicative (`SNES_COMPOSITE_MULTIPLICATIVE`) variants which may be set with

```
SNESCompositeSetType(SNES,SNESCompositeType);
```

New subsolvers may be added to the composite solver with

```
SNESCompositeAddSNES(SNES,SNESType);
```

and accessed with

```
SNESCompositeGetSNES(SNES,PetscInt,SNES *);
```

2.5 TS: Scalable ODE and DAE Solvers

The `TS` library provides a framework for the scalable solution of ODEs and DAEs arising from the discretization of time-dependent PDEs.

Simple Example: Consider the PDE

$$u_t = u_{xx}$$

discretized with centered finite differences in space yielding the semi-discrete equation

$$\begin{aligned}(u_i)_t &= \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \\ u_t &= \tilde{A}u;\end{aligned}$$

or with piecewise linear finite elements approximation in space $u(x,t) \doteq \sum_i \xi_i(t)\phi_i(x)$ yielding the semi-discrete equation

$$B\xi'(t) = A\xi(t)$$

Now applying the backward Euler method results in

$$(B - dt^n A)u^{n+1} = Bu^n,$$

in which

$$u^n{}_i = \xi_i(t_n) \doteq u(x_i, t_n),$$

$$\xi'(t_{n+1}) \doteq \frac{u^{n+1}{}_i - u^n{}_i}{dt^n},$$

A is the stiffness matrix, and B is the identity for finite differences or the mass matrix for the finite element method.

The PETSc interface for solving time dependent problems assumes the problem is written in the form

$$F(t,u,\dot{u}) = G(t,u), \quad u(t_0) = u_0.$$

In general, this is a differential algebraic equation (DAE)[4]. For ODE with nontrivial mass matrices such as arise in FEM, the implicit/DAE interface significantly reduces overhead to prepare the system for algebraic solvers (`SNES`/`KSP`) by having the user assemble the correctly shifted matrix. Therefore this interface is also useful for ODE systems.

To solve an ODE or DAE one uses:

- Function $F(t,u,\dot{u})$

  ```
  TSSetIFunction(TS ts,Vec R,PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,Vec,void*),
  ↪void *funP);
  ```

  The vector `R` is an optional location to store the residual. The arguments to the function `f()` are the timestep context, current time, input state u, input time derivative $\dot{u}$, and the (optional) user-provided context `funP`. If $F(t,u,\dot{u}) = \dot{u}$ then one need not call this function.

- Function $G(t,u)$, if it is nonzero, is provided with the function

  ```
  TSSetRHSFunction(TS ts,Vec R,PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),void␣
  ↪*funP);
  ```

- Jacobian $\sigma F_{\dot{u}}(t^n,u^n,\dot{u}^n) + F_u(t^n,u^n,\dot{u}^n)$
  If using a fully implicit or semi-implicit (IMEX) method one also can provide an appropriate (approximate) Jacobian matrix of $F()$.

  ```
  TSSetIJacobian(TS ts,Mat A,Mat B,PetscErrorCode (*fjac)(TS,PetscReal,Vec,Vec,
  ↪PetscReal,Mat,Mat,void*),void *jacP);
  ```

  The arguments for the function `fjac()` are the timestep context, current time, input state u, input derivative $\dot{u}$, input shift σ, matrix A, preconditioning matrix B, and the (optional) user-provided context `jacP`.

  The Jacobian needed for the nonlinear system is, by the chain rule,

  $$\frac{dF}{du^n} = \frac{\partial F}{\partial \dot{u}}|_{u^n} \frac{\partial \dot{u}}{\partial u}|_{u^n} + \frac{\partial F}{\partial u}|_{u^n}.$$

  For any ODE integration method the approximation of $\dot{u}$ is linear in u^n hence $\frac{\partial \dot{u}}{\partial u}|_{u^n} = \sigma$, where the shift σ depends on the ODE integrator and time step but not on the function being integrated. Thus

  $$\frac{dF}{du^n} = \sigma F_{\dot{u}}(t^n,u^n,\dot{u}^n) + F_u(t^n,u^n,\dot{u}^n).$$

  This explains why the user provide Jacobian is in the given form for all integration methods. An equivalent way to derive the formula is to note that

  $$F(t^n,u^n,\dot{u}^n) = F(t^n,u^n,w+\sigma*u^n)$$

  where w is some linear combination of previous time solutions of u so that

  $$\frac{dF}{du^n} = \sigma F_{\dot{u}}(t^n,u^n,\dot{u}^n) + F_u(t^n,u^n,\dot{u}^n)$$

---

[4] If the matrix $F_{\dot{u}}(t) = \partial F/\partial \dot{u}$ is nonsingular then it is an ODE and can be transformed to the standard explicit form, although this transformation may not lead to efficient algorithms.

again by the chain rule.

For example, consider backward Euler's method applied to the ODE $F(t,u,\dot{u}) = \dot{u} - f(t,u)$ with $\dot{u} = (u^n - u^{n-1})/\delta t$ and $\frac{\partial \dot{u}}{\partial u}|_{u^n} = 1/\delta t$ resulting in

$$\frac{dF}{du^n} = \quad (1/\delta t)F_{\dot{u}} + F_u(t^n, u^n, \dot{u}^n).$$

But $F_{\dot{u}} = 1$, in this special case, resulting in the expected Jacobian $I/\delta t - f_u(t,u^n)$.

- Jacobian G_u

  If using a fully implicit method and the function $G()$ is provided, one also can provide an appropriate (approximate) Jacobian matrix of $G()$.

  ```
  TSSetRHSJacobian(TS ts,Mat A,Mat B,
  PetscErrorCode (*fjac)(TS,PetscReal,Vec,Mat,Mat,void*),void *jacP);
  ```

  The arguments for the function `fjac()` are the timestep context, current time, input state u, matrix A, preconditioning matrix B, and the (optional) user-provided context `jacP`.

Providing appropriate $F()$ and $G()$ for your problem allows for the easy runtime switching between explicit, semi-implicit (IMEX), and fully implicit methods.

2.5.1 Basic TS Options

The user first creates a `TS` object with the command

```
int TSCreate(MPI_Comm comm,TS *ts);
```

```
int TSSetProblemType(TS ts,TSProblemType problemtype);
```

The `TSProblemType` is one of `TS_LINEAR` or `TS_NONLINEAR`.

To set up `TS` for solving an ODE, one must set the "initial conditions" for the ODE with

```
TSSetSolution(TS ts, Vec initialsolution);
```

One can set the solution method with the routine

```
TSSetType(TS ts,TSType type);
```

Currently supported types are `TSEULER`, `TSRK` (Runge-Kutta), `TSBEULER`, `TSCN` (Crank-Nicolson), `TSTHETA`, `TSGLLE` (generalized linear), `TSPSEUDO`, and `TSSUNDIALS` (only if the Sundials package is installed), or the command line option `-ts_type euler,rk,beuler,cn,theta,gl,pseudo,sundials,eimex,arkimex,rosw`.

A list of available methods is given in the following table.

Table 2.11: Time integration schemes

TS Name	Reference	Class	Type	Order
euler	forward Euler	one-step	explicit	1
ssp	multistage SSP [Ket08]	Runge-Kutta	explicit	≤ 4
rk*	multiscale	Runge-Kutta	explicit	≥ 1
beuler	backward Euler	one-step	implicit	1
cn	Crank-Nicolson	one-step	implicit	2
theta*	theta-method	one-step	implicit	≤ 2
alpha	alpha-method [JWH00]	one-step	implicit	2
gl	general linear [BJW07]	multistep-multistage	implicit	≤ 3
eimex	extrapolated IMEX [CS10]	one-step	≥ 1, adaptive	
arkimex	See *IMEX Runge-Kutta schemes*	IMEX Runge-Kutta	IMEX	$1-5$
rosw	See *Rosenbrock W-schemes*	Rosenbrock-W	linearly implicit	$1-4$
glee	See *GL schemes with global error estimation*	GL with global error	explicit and implicit	$1-3$

Set the initial time with the command

```
TSSetTime(TS ts,PetscReal time);
```

One can change the timestep with the command

```
TSSetTimeStep(TS ts,PetscReal dt);
```

can determine the current timestep with the routine

```
TSGetTimeStep(TS ts,PetscReal* dt);
```

Here, "current" refers to the timestep being used to attempt to promote the solution form u^n to u^{n+1}.

One sets the total number of timesteps to run or the total time to run (whatever is first) with the commands

```
TSSetMaxSteps(TS ts,PetscInt maxsteps);
TSSetMaxTime(TS ts,PetscReal maxtime);
```

and determines the behavior near the final time with

```
TSSetExactFinalTime(TS ts,TSExactFinalTimeOption eftopt);
```

where `eftopt` is one of `TS_EXACTFINALTIME_STEPOVER`,`TS_EXACTFINALTIME_INTERPOLATE`, or `TS_EXACTFINALTIME_MATCHSTEP`. One performs the requested number of time steps with

```
TSSolve(TS ts,Vec U);
```

The solve call implicitly sets up the timestep context; this can be done explicitly with

```
TSSetUp(TS ts);
```

One destroys the context with

```
TSDestroy(TS *ts);
```

and views it with

```
TSView(TS ts,PetscViewer viewer);
```

In place of `TSSolve()`, a single step can be taken using

```
TSStep(TS ts);
```

2.5.2 DAE Formulations

You can find a discussion of DAEs in [AP98] or Scholarpedia. In PETSc, TS deals with the semi-discrete form of the equations, so that space has already been discretized. If the DAE depends explicitly on the coordinate x, then this will just appear as any other data for the equation, not as an explicit argument. Thus we have

$$F(t, u, \dot{u}) = 0$$

In this form, only fully implicit solvers are appropriate. However, specialized solvers for restricted forms of DAE are supported by PETSc. Below we consider an ODE which is augmented with algebraic constraints on the variables.

Hessenberg Index-1 DAE

This is a Semi-Explicit Index-1 DAE which has the form

$$\begin{aligned} \dot{u} &= f(t, u, z) \\ 0 &= h(t, u, z) \end{aligned}$$

where z is a new constraint variable, and the Jacobian $\frac{dh}{dz}$ is non-singular everywhere. We have suppressed the x dependence since it plays no role here. Using the non-singularity of the Jacobian and the Implicit Function Theorem, we can solve for z in terms of u. This means we could, in principle, plug $z(u)$ into the first equation to obtain a simple ODE, even if this is not the numerical process we use. Below we show that this type of DAE can be used with IMEX schemes.

Hessenberg Index-2 DAE

This DAE has the form

$$\begin{aligned} \dot{u} &= f(t, u, z) \\ 0 &= h(t, u) \end{aligned}$$

Notice that the constraint equation h is not a function of the constraint variable :math:'z'. This means that we cannot naively invert as we did in the index-1 case. Our strategy will be to convert this into an index-1 DAE using a time derivative, which loosely corresponds to the idea of index being the number of derivatives necessary to get back to an ODE. If we differentiate the constraint equation with respect to time, we can use the ODE to simplify it,

$$\begin{aligned} 0 &= \dot{h}(t, u) \\ &= \frac{dh}{du}\dot{u} + \frac{\partial h}{\partial t} \\ &= \frac{dh}{du} f(t, u, z) + \frac{\partial h}{\partial t} \end{aligned}$$

If the Jacobian $\frac{dh}{du}\frac{df}{dz}$ is non-singular, then we have precisely a semi-explicit index-1 DAE, and we can once again use the PETSc IMEX tools to solve it. A common example of an index-2 DAE is the incompressible

Navier-Stokes equations, since the continuity equation $\nabla \cdot u = 0$ does not involve the pressure. Using PETSc IMEX with the above conversion then corresponds to the Segregated Runge-Kutta method applied to this equation [OColomesB16].

2.5.3 Using Implicit-Explicit (IMEX) Methods

For "stiff" problems or those with multiple time scales $F()$ will be treated implicitly using a method suitable for stiff problems and $G()$ will be treated explicitly when using an IMEX method like TSARKIMEX. $F()$ is typically linear or weakly nonlinear while $G()$ may have very strong nonlinearities such as arise in non-oscillatory methods for hyperbolic PDE. The user provides three pieces of information, the APIs for which have been described above.

- "Slow" part $G(t, u)$ using `TSSetRHSFunction()`.
- "Stiff" part $F(t, u, \dot{u})$ using `TSSetIFunction()`.
- Jacobian $F_u + \sigma F_{\dot{u}}$ using `TSSetIJacobian()`.

The user needs to set `TSSetEquationType()` to `TS_EQ_IMPLICIT` or higher if the problem is implicit; e.g., $F(t, u, \dot{u}) = M\dot{u} - f(t, u)$, where M is not the identity matrix:

- the problem is an implicit ODE (defined implicitly through `TSSetIFunction()`) or
- a DAE is being solved.

An IMEX problem representation can be made implicit by setting `TSARKIMEXSetFullyImplicit()`.

In PETSc, DAEs and ODEs are formulated as $F(t, u, \dot{u}) = G(t, u)$, where $F()$ is meant to be integrated implicitly and $G()$ explicitly. An IMEX formulation such as $M\dot{u} = f(t, u) + g(t, u)$ requires the user to provide $M^{-1}g(t, u)$ or solve $g(t, u) - Mx = 0$ in place of $G(t, u)$. General cases such as $F(t, u, \dot{u}) = G(t, u)$ are not amenable to IMEX Runge-Kutta, but can be solved by using fully implicit methods. Some use-case examples for `TSARKIMEX` are listed in Table 2.5.3 and a list of methods with a summary of their properties is given in *IMEX Runge-Kutta schemes*.

$\dot{u} = g(t, u)$	nonstiff ODE	$F(t, u, \dot{u}) = \dot{u}$ $G(t, u) = g(t, u)$
$M\dot{u} = g(t, u)$	nonstiff ODE with mass matrix	$F(t, u, \dot{u}) = \dot{u}$ $G(t, u) = M^{-1}g(t, u)$
$\dot{u} = f(t, u)$	stiff ODE	$F(t, u, \dot{u}) = \dot{u} - f(t, u)$ $G(t, u) = 0$
$M\dot{u} = f(t, u)$	stiff ODE with mass matrix	$F(t, u, \dot{u}) = M\dot{u} - f(t, u)$ $G(t, u) = 0$
$\dot{u} = f(t, u) + g(t, u)$	stiff-nonstiff ODE	$F(t, u, \dot{u}) = \dot{u} - f(t, u)$ $G(t, u) = g(t, u)$
$M\dot{u} = f(t, u) + g(t, u)$	stiff-nonstiff ODE with mass matrix	$F(t, u, \dot{u}) = M\dot{u} - f(t, u)$ $G(t, u) = M^{-1}g(t, u)$
$\dot{u} = f(t, u, z) + g(t, u, z)$ $0 = h(t, y, z)$	semi-explicit index-1 DAE	$F(t, u, \dot{u}) = \begin{pmatrix} \dot{u} - f(t, u, z) \\ h(t, u, z) \end{pmatrix}$ $G(t, u) = g(t, u)$
$f(t, u, \dot{u}) = 0$	fully implicit ODE/DAE	$F(t, u, \dot{u}) = f(t, u, \dot{u})$ $G(t, u) = 0$; the user needs to set `TSSetEquationType()` to `TS_EQ_IMPLICIT` or higher

Table 2.12 lists of the currently available IMEX Runge-Kutta schemes. For each method, it gives the `-ts_arkimex_type` name, the reference, the total number of stages/implicit stages, the order/stage-order, the implicit stability properties (IM), stiff accuracy (SA), the existence of an embedded scheme, and dense output (DO).

Table 2.12: IMEX Runge-Kutta schemes

Name	Reference	Stages (IM)	Order (Stage)	IM	SA	Embed	DO	Remarks
a2	based on CN	2 (1)	2 (2)	A-Stable	yes	yes (1)	yes (2)	
l2	SSP2(2,2,2) [PR05]	2 (2)	2 (1)	L-Stable	yes	yes (1)	yes (2)	SSP SDIRK
ars122	ARS122 [ARS97]	2 (1)	3 (1)	A-Stable	yes	yes (1)	yes (2)	
2c	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
2d	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
2e	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
prssp2	PRS(3,3,2) [PR05]	3 (3)	3 (1)	L-Stable	yes	no	no	SSP
3	[KC03]	4 (3)	3 (2)	L-Stable	yes	yes (2)	yes (2)	SDIRK
bpr3	[BPR11]	5 (4)	3 (2)	L-Stable	yes	no	no	SDIRK
ars443	[ARS97]	5 (4)	3 (1)	L-Stable	yes	no	no	SDIRK
4	[KC03]	6 (5)	4 (2)	L-Stable	yes	yes (3)	yes	SDIRK
5	[KC03]	8 (7)	5 (2)	L-Stable	yes	yes (4)	yes (3)	SDIRK

ROSW are linearized implicit Runge-Kutta methods known as Rosenbrock W-methods. They can accommodate inexact Jacobian matrices in their formulation. A series of methods are available in PETSc are listed in Table 2.13 below. For each method, it gives the reference, the total number of stages and implicit stages, the scheme order and stage order, the implicit stability properties (IM), stiff accuracy (SA), the existence of an embedded scheme, dense output (DO), the capacity to use inexact Jacobian matrices (-W), and high order integration of differential algebraic equations (PDAE).

Table 2.13: Rosenbrock W-schemes

TS	Reference	Stages (IM)	Order (Stage)	IM	SA	Embed	DO	-W	PDAE	Remarks
theta1	classical	1(1)	1(1)	L-Stable	•	•	•	•	•	•
theta2	classical	1(1)	2(2)	A-Stable	•	•	•	•	•	•
2m	Zoltan	2(2)	2(1)	L-Stable	No	Yes(1)	Yes(2)	Yes	No	SSP
2p	Zoltan	2(2)	2(1)	L-Stable	No	Yes(1)	Yes(2)	Yes	No	SSP
ra3pw	[RA05]	3(3)	3(1)	A-Stable	No	Yes	Yes(2)	No	Yes(3)	•
ra34pw2	[RA05]	4(4)	3(1)	L-Stable	Yes	Yes	Yes(3)	Yes	Yes(3)	•
rodas3	[SVB+97]	4(4)	3(1)	L-Stable	Yes	Yes	No	No	Yes	•
sandu3	[SVB+97]	3(3)	3(1)	L-Stable	Yes	Yes	Yes(2)	No	No	•
assp3p3s1c	unpub.	3(2)	3(1)	A-Stable	No	Yes	Yes(2)	Yes	No	SSP
lassp3p4s2c	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	SSP
lassp3p4s2c	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	SSP
ark3	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	IMEX-RK

2.5.4 GLEE methods

In this section, we describe explicit and implicit time stepping methods with global error estimation that are introduced in [Con16]. The solution vector for a GLEE method is either [y, $\tilde{y}$] or [y,ε], where y is the solution, $\tilde{y}$ is the "auxiliary solution," and ε is the error. The working vector that `TSGLEE` uses is Y = [y,$\tilde{y}$], or [y,ε]. A GLEE method is defined by

- (p, r, s): (order, steps, and stages),
- γ: factor representing the global error ratio,
- A, U, B, V: method coefficients,
- S: starting method to compute the working vector from the solution (say at the beginning of time integration) so that $Y = Sy$,
- F: finalizing method to compute the solution from the working vector,$y = FY$.
- F_embed: coefficients for computing the auxiliary solution $\tilde{y}$ from the working vector ($\tilde{y} = F_\text{embed}Y$),
- F_error: coefficients to compute the estimated error vector from the working vector ($\varepsilon = F_\text{error}Y$).
- S_error: coefficients to initialize the auxiliary solution ($\tilde{y}$ or ε) from a specified error vector (ε). It is currently implemented only for $r = 2$. We have $y_\text{aux} = S_{error}[0] * \varepsilon + S_\text{error}[1] * y$, where y_aux is the 2nd component of the working vector Y.

The methods can be described in two mathematically equivalent forms: propagate two components ("$y\tilde{y}$ form") and propagating the solution and its estimated error ("$y\varepsilon$ form"). The two forms are not explicitly specified in `TSGLEE`; rather, the specific values of B, U, S, F, F_{embed}, and F_{error} characterize whether the method is in $y\tilde{y}$ or $y\varepsilon$ form.

The API used by this `TS` method includes:

- `TSGetSolutionComponents`: Get all the solution components of the working vector

  ```
  ierr = TSGetSolutionComponents(TS,int*,Vec*)
  ```

  Call with `NULL` as the last argument to get the total number of components in the working vector Y (this is r (not $r-1$)), then call to get the i-th solution component.

- `TSGetAuxSolution`: Returns the auxiliary solution $\tilde{y}$ (computed as $F_{\text{embed}}Y$)

  ```
  ierr = TSGetAuxSolution(TS,Vec*)
  ```

- `TSGetTimeError`: Returns the estimated error vector ε (computed as $F_{\text{error}}Y$ if $n=0$ or restores the error estimate at the end of the previous step if $n=-1$)

  ```
  ierr = TSGetTimeError(TS,PetscInt n,Vec*)
  ```

- `TSSetTimeError`: Initializes the auxiliary solution ($\tilde{y}$ or ε) for a specified initial error.

  ```
  ierr = TSSetTimeError(TS,Vec)
  ```

The local error is estimated as $\varepsilon(n+1)-\varepsilon(n)$. This is to be used in the error control. The error in $y\tilde{y}$ GLEE is $\varepsilon(n) = \frac{1}{1-\gamma} * (\tilde{y}(n) - y(n))$.

Note that y and $\tilde{y}$ are reported to `TSAdapt basic` (`TSADAPTBASIC`), and thus it computes the local error as $\varepsilon_{loc} = (\tilde{y} - y)$. However, the actual local error is $\varepsilon_{loc} = \varepsilon_{n+1} - \varepsilon_n = \frac{1}{1-\gamma} * [(\tilde{y} - y)_{n+1} - (\tilde{y} - y)_n]$.

Table 2.14 lists currently available GL schemes with global error estimation [Con16].

Table 2.14: GL schemes with global error estimation

TS	Reference	IM/EX	(p,r,s)	γ	Form	Notes
`TSGLEEi1`	`BE1`	IM	$(1,3,2)$	0.5	$y\varepsilon$	Based on backward Euler
`TSGLEE23`	`23`	EX	$(2,3,2)$	0	$y\varepsilon$	
`TSGLEE24`	`24`	EX	$(2,4,2)$	0	$y\tilde{y}$	
`TSGLEE25I`	`25i`	EX	$(2,5,2)$	0	$y\tilde{y}$	
`TSGLEE35`	`35`	EX	$(3,5,2)$	0	$y\tilde{y}$	
`TSGLEEEXRK2A`	`exrk2a`	EX	$(2,6,2)$	0.25	$y\varepsilon$	
`TSGLEERK32G1`	`rk32g1`	EX	$(3,8,2)$	0	$y\varepsilon$	
`TSGLEERK285EX`	`rk285ex`	EX	$(2,9,2)$	0.25	$y\varepsilon$	

2.5.5 Using fully implicit methods

To use a fully implicit method like `TSTHETA` or `TSGL`, either provide the Jacobian of $F()$ (and $G()$ if $G()$ is provided) or use a `DM` that provides a coloring so the Jacobian can be computed efficiently via finite differences.

2.5.6 Using the Explicit Runge-Kutta timestepper with variable timesteps

The explicit Euler and Runge-Kutta methods require the ODE be in the form

$$\dot{u} = G(u,t).$$

The user can either call `TSSetRHSFunction()` and/or they can call `TSSetIFunction()` (so long as the function provided to `TSSetIFunction()` is equivalent to $\dot{u} + \tilde{F}(t,u)$) but the Jacobians need not be provided.[5]

The Explicit Runge-Kutta timestepper with variable timesteps is an implementation of the standard Runge-Kutta with an embedded method. The error in each timestep is calculated using the solutions from the Runge-Kutta method and its embedded method (the 2-norm of the difference is used). The default method is the 3rd-order Bogacki-Shampine method with a 2nd-order embedded method (`TSRK3BS`). Other available methods are the 5th-order Fehlberg RK scheme with a 4th-order embedded method (`TSRK5F`), the 5th-order Dormand-Prince RK scheme with a 4th-order embedded method (`TSRK5DP`), the 5th-order Bogacki-Shampine RK scheme with a 4th-order embedded method (`TSRK5BS`, and the 6th-, 7th, and 8th-order robust Verner RK schemes with a 5th-, 6th, and 7th-order embedded method, respectively (`TSRK6VR`, `TSRK7VR`, `TSRK8VR`). Variable timesteps cannot be used with RK schemes that do not have an embedded method (`TSRK1FE` - 1st-order, 1-stage forward Euler, `TSRK2A` - 2nd-order, 2-stage RK scheme, `TSRK3` - 3rd-order, 3-stage RK scheme, `TSRK4` - 4-th order, 4-stage RK scheme).

2.5.7 Special Cases

- $\dot{u} = Au$. First compute the matrix A then call

  ```
  TSSetProblemType(ts,TS_LINEAR);
  TSSetRHSFunction(ts,NULL,TSComputeRHSFunctionLinear,NULL);
  TSSetRHSJacobian(ts,A,A,TSComputeRHSJacobianConstant,NULL);
  ```

  or

  ```
  TSSetProblemType(ts,TS_LINEAR);
  TSSetIFunction(ts,NULL,TSComputeIFunctionLinear,NULL);
  TSSetIJacobian(ts,A,A,TSComputeIJacobianConstant,NULL);
  ```

- $\dot{u} = A(t)u$. Use

  ```
  TSSetProblemType(ts,TS_LINEAR);
  TSSetRHSFunction(ts,NULL,TSComputeRHSFunctionLinear,NULL);
  TSSetRHSJacobian(ts,A,A,YourComputeRHSJacobian, &appctx);
  ```

  where `YourComputeRHSJacobian()` is a function you provide that computes A as a function of time. Or use

  ```
  TSSetProblemType(ts,TS_LINEAR);
  TSSetIFunction(ts,NULL,TSComputeIFunctionLinear,NULL);
  TSSetIJacobian(ts,A,A,YourComputeIJacobian, &appctx);
  ```

---

[5] PETSc will automatically translate the function provided to the appropriate form.

2.5.8 Monitoring and visualizing solutions

- `-ts_monitor` - prints the time and timestep at each iteration.
- `-ts_adapt_monitor` - prints information about the timestep adaption calculation at each iteration.
- `-ts_monitor_lg_timestep` - plots the size of each timestep, `TSMonitorLGTimeStep()`.
- `-ts_monitor_lg_solution` - for ODEs with only a few components (not arising from the discretization of a PDE) plots the solution as a function of time, `TSMonitorLGSolution()`.
- `-ts_monitor_lg_error` - for ODEs with only a few components plots the error as a function of time, only if `TSSetSolutionFunction()` is provided, `TSMonitorLGError()`.
- `-ts_monitor_draw_solution` - plots the solution at each iteration, `TSMonitorDrawSolution()`.
- `-ts_monitor_draw_error` - plots the error at each iteration only if `TSSetSolutionFunction()` is provided, `TSMonitorDrawSolution()`.
- `-ts_monitor_solution binary[:filename]` - saves the solution at each iteration to a binary file, `TSMonitorSolution()`.
- `-ts_monitor_solution_vtk <filename-%03D.vts>` - saves the solution at each iteration to a file in vtk format, `TSMonitorSolutionVTK()`.

2.5.9 Error control via variable time-stepping

Most of the time stepping methods avaialable in PETSc have an error estimation and error control mechanism. This mechanism is implemented by changing the step size in order to maintain user specified absolute and relative tolerances. The PETSc object responsible with error control is `TSAdapt`. The available `TSAdapt` types are listed in the following table.

Table 2.15: `TSAdapt`: available adaptors

ID	Name	Notes
TSADAPT-NONE	none	no adaptivity
TSADAPT-BASIC	basic	the default adaptor
TSADAPT-GLEE	glee	extension of the basic adaptor to treat Tol_A and Tol_R as separate criteria. It can also control global erorrs if the integrator (e.g., `TSGLEE`) provides this information

When using `TSADAPTBASIC` (the default), the user typically provides a desired absolute Tol_A or a relative Tol_R error tolerance by invoking `TSSetTolerances()` or at the command line with options `-ts_atol` and `-ts_rtol`. The error estimate is based on the local truncation error, so for every step the algorithm verifies that the estimated local truncation error satisfies the tolerances provided by the user and computes a new step size to be taken. For multistage methods, the local truncation is obtained by comparing the solution y to a lower order $\widehat{p} = p - 1$ approximation, $\widehat{y}$, where p is the order of the method and $\widehat{p}$ the order of $\widehat{y}$.

The adaptive controller at step n computes a tolerance level

$$Tol_n(i) = \quad \mathrm{Tol}_\mathrm{A}(i) + \max(y_n(i), \widehat{y}_n(i)) \mathrm{Tol}_\mathrm{R}(i) \,,$$

and forms the acceptable error level

$$\mathrm{wlte_n} = \quad \frac{1}{m} \sum_{i=1}^{m} \sqrt{\frac{\| y_n(i) - \widehat{y}_n(i) \|}{Tol(i)}} \,,$$

where the errors are computed componentwise, m is the dimension of y and `-ts_adapt_wnormtype` is `2` (default). If `-ts_adapt_wnormtype` is `infinity` (max norm), then

$$\mathrm{wlte_n} = \quad \max_{1 \dots m} \frac{\|y_n(i) - \widehat{y}_n(i)\|}{Tol(i)}.$$

The error tolerances are satisfied when $\mathrm{wlte} \leq 1.0$.

The next step size is based on this error estimate, and determined by

$$\Delta t_{\mathrm{new}}(t) = \quad \Delta t_{\mathrm{old}} \min(\alpha_{\mathrm{max}}, \max(\alpha_{\mathrm{min}}, \beta (1/\mathrm{wlte})^{\frac{1}{\mathrm{p}+1}})),$$

where α_{min} =`-ts_adapt_clip`[0] and α_{max}=`-ts_adapt_clip`[1] keep the change in Δt to within a certain factor, and $\beta < 1$ is chosen through `-ts_adapt_safety` so that there is some margin to which the tolerances are satisfied and so that the probability of rejection is decreased.

This adaptive controller works in the following way. After completing step k, if $\mathrm{wlte_{k+1}} \leq 1.0$, then the step is accepted and the next step is modified according to (*[eq:hnew]*); otherwise, the step is rejected and retaken with the step length computed in (*[eq:hnew]*).

`TSADAPTGLEE` is an extension of the basic adaptor to treat $\mathrm{Tol_A}$ and $\mathrm{Tol_R}$ as separate criteria. it can also control global errors if the integrator (e.g., `TSGLEE`) provides this information.

2.5.10 Handling of discontinuities

For problems that involve discontinuous right hand sides, one can set an "event" function $g(t, u)$ for PETSc to detect and locate the times of discontinuities (zeros of $g(t, u)$). Events can be defined through the event monitoring routine

```
TSSetEventHandler(TS ts,PetscInt nevents,PetscInt *direction,PetscBool *terminate,
↪PetscErrorCode (*eventhandler)(TS,PetscReal,Vec,PetscScalar*,void* eventP),
↪PetscErrorCode (*postevent)(TS,PetscInt,PetscInt[],PetscReal,Vec,PetscBool,void*␣
↪eventP),void *eventP);
```

Here, `nevents` denotes the number of events, `direction` sets the type of zero crossing to be detected for an event (+1 for positive zero-crossing, -1 for negative zero-crossing, and 0 for both), `terminate` conveys whether the time-stepping should continue or halt when an event is located, `eventmonitor` is a user-defined routine that specifies the event description, `postevent` is an optional user-defined routine to take specific actions following an event.

The arguments to `eventhandler()` are the timestep context, current time, input state u, array of event function value, and the (optional) user-provided context `eventP`.

The arguments to `postevent()` routine are the timestep context, number of events occured, indices of events occured, current time, input state u, a boolean flag indicating forward solve (1) or adjoint solve (0), and the (optional) user-provided context `eventP`.

The event monitoring functionality is only available with PETSc's implicit time-stepping solvers `TSTHETA`, `TSARKIMEX`, and `TSROSW`.

2.5.11 Using TChem from PETSc

TChem[6] is a package originally developed at Sandia National Laboratory that can read in CHEMKIN[7] data files and compute the right hand side function and its Jacobian for a reaction ODE system. To utilize PETSc's ODE solvers for these systems, first install PETSc with the additional `./configure` option `--download-tchem`. We currently provide two examples of its use; one for single cell reaction and one for an "artificial" one dimensional problem with periodic boundary conditions and diffusion of all species. The self-explanatory examples are the The TS tutorial extchem and The TS tutorial extchemfield.

2.5.12 Using Sundials from PETSc

Sundials is a parallel ODE solver developed by Hindmarsh et al. at LLNL. The `TS` library provides an interface to use the CVODE component of Sundials directly from PETSc. (To configure PETSc to use Sundials, see the installation guide, `docs/installation/index.htm`.)

To use the Sundials integrators, call

```
TSSetType(TS ts,TSType TSSUNDIALS);
```

or use the command line option `-ts_type` `sundials`.

Sundials' CVODE solver comes with two main integrator families, Adams and BDF (backward differentiation formula). One can select these with

```
TSSundialsSetType(TS ts,TSSundialsLmmType [SUNDIALS_ADAMS,SUNDIALS_BDF]);
```

or the command line option `-ts_sundials_type <adams,bdf>`. BDF is the default.

Sundials does not use the `SNES` library within PETSc for its nonlinear solvers, so one cannot change the nonlinear solver options via `SNES`. Rather, Sundials uses the preconditioners within the `PC` package of PETSc, which can be accessed via

```
TSSundialsGetPC(TS ts,PC *pc);
```

The user can then directly set preconditioner options; alternatively, the usual runtime options can be employed via `-pc_xxx`.

Finally, one can set the Sundials tolerances via

```
TSSundialsSetTolerance(TS ts,double abs,double rel);
```

where `abs` denotes the absolute tolerance and `rel` the relative tolerance.

Other PETSc-Sundials options include

```
TSSundialsSetGramSchmidtType(TS ts,TSSundialsGramSchmidtType type);
```

where `type` is either `SUNDIALS_MODIFIED_GS` or `SUNDIALS_UNMODIFIED_GS`. This may be set via the options data base with `-ts_sundials_gramschmidt_type <modifed,unmodified>`.

The routine

```
TSSundialsSetMaxl(TS ts,PetscInt restart);
```

sets the number of vectors in the Krylov subpspace used by GMRES. This may be set in the options database with `-ts_sundials_maxl` `maxl`.

---

[6] bitbucket.org/jedbrown/tchem

[7] en.wikipedia.org/wiki/CHEMKIN

2.6 Performing sensitivity analysis

The `TS` library provides a framework based on discrete adjoint models for sensitivity analysis for ODEs and DAEs. The ODE/DAE solution process (henceforth called the forward run) can be obtained by using either explicit or implicit solvers in `TS`, depending on the problem properties. Currently supported method types are `TSRK` (Runge-Kutta) explicit methods and `TSTHETA` implicit methods, which include `TSBEULER` and `TSCN`.

2.6.1 Using the discrete adjoint methods

Consider the ODE/DAE

$$F(t,y,\dot{y},p)=0,\quad y(t_0)=y_0(p)\quad t_0\le t\le t_F$$

and the cost function(s)

$$\Psi_i(y_0,p)=\Phi_i(y_F,p)+\int_{t_0}^{t_F} r_i(y(t),p,t)dt\quad i=1,...,n_{\text{cost}}.$$

The `TSAdjoint` routines of PETSc provide

$$\frac{\partial \Psi_i}{\partial y_0}=\lambda_i$$

and

$$\frac{\partial \Psi_i}{\partial p}=\mu_i+\lambda_i(\frac{\partial y_0}{\partial p}).$$

To perform the discrete adjoint sensitivity analysis one first sets up the `TS` object for a regular forward run but with one extra function call

```
TSSetSaveTrajectory(TS ts),
```

then calls `TSSolve()` in the usual manner.

One must create two arrays of n_{cost} vectors λ andμ (if there are no parameters p then one can use `NULL` for the μ array.) The λ vectors are the same dimension and parallel layout as the solution vector for the ODE, the mu vectors are of dimension p; when p is small usually all its elements are on the first MPI process, while the vectors have no entries on the other processes. λ_i and mu_i should be initialized with the values $d\Phi_i/dy|_{t=t_F}$ and $d\Phi_i/dp|_{t=t_F}$ respectively. Then one calls

```
TSSetCostGradients(TS ts,PetscInt numcost, Vec *lambda,Vec *mu);
```

If $F()$ is a function of p one needs to also provide the Jacobian $-F_p$ with

```
TSSetRHSJacobianP(TS ts,Mat Amat,PetscErrorCode (*fp)(TS,PetscReal,Vec,Mat,void*),
↪void *ctx)
```

The arguments for the function `fp()` are the timestep context, current time, y, and the (optional) user-provided context.

If there is an integral term in the cost function, i.e. r is nonzero, it can be transformed into another ODE that is augmented to the original ODE. To evaluate the integral, one needs to create a child `TS` objective by calling

```
TSCreateQuadratureTS(TS ts,PetscBool fwd,TS *quadts);
```

and provide the ODE RHS function (which evaluates the integrand r) with

```
TSSetRHSFunction(TS quadts,Vec R,PetscErrorCode (*rf)(TS,PetscReal,Vec,Vec,void*),
↪void *ctx)
```

Similar to the settings for the original ODE, Jacobians of the integrand can be provided with

```
TSSetRHSJacobian(TS quadts,Vec DRDU,Vec DRDU,PetscErrorCode (*drdyf)(TS,PetscReal,Vec,
↪Vec*,void*),void *ctx)
TSSetRHSJacobianP(TS quadts,Vec DRDU,Vec DRDU,PetscErrorCode (*drdyp)(TS,PetscReal,
↪Vec,Vec*,void*),void *ctx)
```

where drdyf $= dr/dy$, drdpf $= dr/dp$. Since the integral term is additive to the cost function, its gradient information will be included in λ and μ.

Lastly, one starts the backward run by calling

```
TSAdjointSolve(TS ts).
```

One can obtain the value of the integral term by calling

```
TSGetCostIntegral(TS ts,Vec *q).
```

or accessing directly the solution vector used by quadts.

The second argument of `TSCreateQuadratureTS()` allows one to choose if the integral term is evaluated in the forward run (inside `TSSolve()`) or in the backward run (inside `TSAdjointSolve()`) when `TSSetCostGradients()` and `TSSetCostIntegrand()` are called before `TSSolve()`. Note that this also allows for evaluating the integral without having to use the adjoint solvers.

To provide a better understanding of the use of the adjoint solvers, we introduce a simple example, corresponding to TS Power Grid Tutorial ex3adj. The problem is to study dynamic security of power system when there are credible contingencies such as short-circuits or loss of generators, transmission lines, or loads. The dynamic security constraints are incorporated as equality constraints in the form of discretized differential equations and inequality constraints for bounds on the trajectory. The governing ODE system is

$$\begin{aligned} \phi' &= & \omega_B(\omega - \omega_S) \\ 2H/\omega_S\,\omega' &= p_m - p_{max} sin(\phi) - D(\omega - \omega_S), & t_0 \leq t \leq t_F, \end{aligned}$$

where ϕ is the phase angle and ω is the frequency.

The initial conditions at time t_0 are

$$\begin{aligned} \phi(t_0) &= \arcsin\left(p_m/p_{max}\right), \\ w(t_0) &= 1. \end{aligned}$$

p_{max} is a positive number when the system operates normally. At an event such as fault incidence/removal, p_{max} will change to 0 temporarily and back to the original value after the fault is fixed. The objective is to maximize p_m subject to the above ODE constraints and $\phi < \phi_S$ during all times. To accommodate the inequality constraint, we want to compute the sensitivity of the cost function

$$\Psi(p_m, \phi) = -p_m + c \int_{t_0}^{t_F} \left(\max(0, \phi - \phi_S)\right)^2 dt$$

with respect to the parameter p_m. *numcost* is 1 since it is a scalar function.

For ODE solution, PETSc requires user-provided functions to evaluate the system $F(t, y, \dot{y}, p)$ (set by `TSSetIFunction()`) and its corresponding Jacobian $F_y + \sigma F_{\dot{y}}$ (set by `TSSetIJacobian()`). Note that the solution state y is $[\phi\ \omega]^T$ here. For sensitivity analysis, we need to provide a routine to compute $\mathrm{f}_p = [0\ 1]^T$ using `TSASetRHSJacobianP()`, and three routines corresponding to the integrand $r = c\left(\max(0, \phi - \phi_S)\right)^2$, $r_p = [0\ 0]^T$ and $r_y = [2c\left(\max(0, \phi - \phi_S)\right)\ 0]^T$ using `TSSetCostIntegrand()`.

In the adjoint run, λ and μ are initialized as $[0\ 0]^T$ and $[-1]$ at the final time t_F. After `TSAdjointSolve()`, the sensitivity of the cost function w.r.t. initial conditions is given by the sensitivity variable λ (at time t_0) directly. And the sensitivity of the cost function w.r.t. the parameter p_m can be computed (by users) as

$$\frac{\mathrm{d}\Psi}{\mathrm{d}p_m} = \mu(t_0) + \lambda(t_0)\frac{\mathrm{d}\left[\phi(t_0)\ \omega(t_0)\right]^T}{\mathrm{d}p_m}.$$

For explicit methods where one does not need to provide the Jacobian F_u for the forward solve one still does need it for the backward solve and thus must call

```
TSSetRHSJacobian(TS ts,Mat Amat, Mat Pmat,PetscErrorCode (*f)(TS,PetscReal,Vec,Mat,
↪Mat,void*),void *fP);
```

Examples include:

- a discrete adjoint sensitivity using explicit time stepping methods TS Tutorial ex16adj,
- a discrete adjoint sensitivity using implicit time stepping methods TS Tutorial ex20adj,
- an optimization using the discrete adjoint models of ERK TS Tutorial ex16opt_ic and *TS Tutorial ex16opt_p* <https://www.mcs.anl.gov/petsc/petsc-current/src/ts/tutorials/ex16opt_p.c.html>‘___,
- an optimization using the discrete adjoint models of Theta methods for stiff DAEs TS Tutorial ex20opt_ic and TS Tutorial ex20opt_p,
- an ODE-constrained optimization using the discrete adjoint models of Theta methods for cost function with an integral term TS Power Grid Tutorial ex3opt,
- a discrete adjoint sensitivity using `TSCN` (Crank-Nicolson) methods for DAEs with discontinuities TS Power Grid Stability Tutorial ex9busadj.c,
- a DAE-constrained optimization using the discrete adjoint models of `TSCN` (Crank-Nicolson) methods for cost function with an integral term TS Power Grid Tutorial ex9busopt.c,
- a discrete adjoint sensitivity using `TSCN` methods for a PDE problem TS Advection-Diffusion-Reaction Tutorial ex5adj.

2.6.2 Checkpointing

The discrete adjoint model requires the states (and stage values in the context of multistage timestepping methods) to evaluate the Jacobian matrices during the adjoint (backward) run. By default, PETSc stores the whole trajectory to disk as binary files, each of which contains the information for a single time step including state, time, and stage values (optional). One can also make PETSc store the trajectory to memory with the option `-ts_trajectory_type memory`. However, there might not be sufficient memory capacity especially for large-scale problems and long-time integration.

A so-called checkpointing scheme is needed to solve this problem. The scheme stores checkpoints at selective time steps and recomputes the missing information. The `revolve` library is used by PETSc `TSTrajectory` to generate an optimal checkpointing schedule that minimizes the recomputations given a limited number of available checkpoints. One can specify the number of available checkpoints with the option `-ts_trajectory_max_cps_ram [maximum number of checkpoints in RAM]`. Note that one checkpoint corresponds to one time step.

The `revolve` library also provides an optimal multistage checkpointing scheme that uses both RAM and disk for storage. This scheme is automatically chosen if one uses both the option `-ts_trajectory_max_cps_ram [maximum number of checkpoints in RAM]` and the option `-ts_trajectory_max_cps_disk [maximum number of checkpoints on disk]`.

Some other useful options are listed below.

- `-ts_trajectory_view` prints the total number of recomputations,
- `-ts_monitor` and `-ts_adjoint_monitor` allow users to monitor the progress of the adjoint work flow,
- `-ts_trajectory_type visualization` may be used to save the whole trajectory for visualization. It stores the solution and the time, but no stage values. The binary files generated can be read into MATLAB via the script `${PETSC_DIR}/share/petsc/matlab/PetscReadBinaryTrajectory.m`.

2.7 Solving Steady-State Problems with Pseudo-Timestepping

Simple Example: `TS` provides a general code for performing pseudo timestepping with a variable timestep at each physical node point. For example, instead of directly attacking the steady-state problem

$$G(u) = 0,$$

we can use pseudo-transient continuation by solving

$$u_t = G(u).$$

Using time differencing

$$u_t \doteq \frac{u^{n+1} - u^n}{dt^n}$$

with the backward Euler method, we obtain nonlinear equations at a series of pseudo-timesteps

$$\frac{1}{dt^n} B(u^{n+1} - u^n) = G(u^{n+1}).$$

For this problem the user must provide $G(u)$, the time steps dt^n and the left-hand-side matrix B (or optionally, if the timestep is position independent and B is the identity matrix, a scalar timestep), as well as optionally the Jacobian of $G(u)$.

More generally, this can be applied to implicit ODE and DAE for which the transient form is

$$F(u, \dot{u}) = 0.$$

For solving steady-state problems with pseudo-timestepping one proceeds as follows.

- Provide the function `G(u)` with the routine

  ```
  TSSetRHSFunction(TS ts,Vec r,PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),void␣
  ↪*fP);
  ```

  The arguments to the function `f()` are the timestep context, the current time, the input for the function, the output for the function and the (optional) user-provided context variable `fP`.

- Provide the (approximate) Jacobian matrix of `G(u)` and a function to compute it at each Newton iteration. This is done with the command

```
TSSetRHSJacobian(TS ts,Mat Amat, Mat Pmat,PetscErrorCode (*f)(TS,PetscReal,Vec,
↪Mat,Mat,void*),void *fP);
```

The arguments for the function `f()` are the timestep context, the current time, the location where the Jacobian is to be computed, the (approximate) Jacobian matrix, an alternative approximate Jacobian matrix used to construct the preconditioner, and the optional user-provided context, passed in as `fP`. The user must provide the Jacobian as a matrix; thus, if using a matrix-free approach, one must create a `MATSHELL` matrix.

In addition, the user must provide a routine that computes the pseudo-timestep. This is slightly different depending on if one is using a constant timestep over the entire grid, or it varies with location.

- For location-independent pseudo-timestepping, one uses the routine

```
TSPseudoSetTimeStep(TS ts,PetscInt(*dt)(TS,PetscReal*,void*),void* dtctx);
```

The function `dt` is a user-provided function that computes the next pseudo-timestep. As a default one can use `TSPseudoTimeStepDefault(TS,PetscReal*,void*)` for `dt`. This routine updates the pseudo-timestep with one of two strategies: the default

$$dt^n = dt_{\text{increment}} * dt^{n-1} * \frac{||F(u^{n-1})||}{||F(u^n)||}$$

or, the alternative,

$$dt^n = dt_{\text{increment}} * dt^0 * \frac{||F(u^0)||}{||F(u^n)||}$$

which can be set with the call

```
TSPseudoIncrementDtFromInitialDt(TS ts);
```

or the option `-ts_pseudo_increment_dt_from_initial_dt`. The value $dt_{\text{increment}}$ is by default 1.1, but can be reset with the call

```
TSPseudoSetTimeStepIncrement(TS ts,PetscReal inc);
```

or the option `-ts_pseudo_increment <inc>`.

- For location-dependent pseudo-timestepping, the interface function has not yet been created.

2.8 High Level Support for Multigrid with KSPSetDM() and SNESSetDM()

This chapter needs to be written. For now, see the manual pages (and linked examples) for `KSPSetDM()` and `SNESSetDM()`.

Smoothing on each level of the hierarchy is handled by a `KSP` held by the `PCMG`, or in the nonlinear case, a `SNES` held by `SNESFAS`. The `DM` for each level is associated with the smoother using `KSPSetDM()` and `SNESSetDM()`.

The linear operators which carry out interpolation and restriction (usually of type `MATMAIJ`) are held by the `PCMG`/`SNESFAS`, and generated automatically by the `DM` using information about the discretization. Below we briefly discuss the different operations:

Interpolation transfers a function from the coarse space to the fine space. We would like this process to be accurate for the functions resolved by the coarse grid, in particular the approximate solution computed there. By default, we create these matrices using local interpolation of the fine grid dual basis functions in the coarse basis. However, an adaptive procedure can optimize the coefficients of the interpolator to reproduce pairs of coarse/fine functions which should approximate the lowest modes of the generalized eigenproblem

$$Ax = \lambda Mx$$

where A is the system matrix and M is the smoother. Note that for defect-correction MG, the interpolated solution from the coarse space need not be as accurate as the fine solution, for the same reason that updates in iterative refinement can be less accurate. However, in FAS or in the final interpolation step for each level of Full Multigrid, we must have interpolation as accurate as the fine solution since we are moving the entire solution itself.

Injection should accurately transfer the fine solution to the coarse grid. Accuracy here means that the action of a coarse dual function on either should produce approximately the same result. In the structured grid case, this means that we just use the same values on coarse points. This can result in aliasing.

Restriction is intended to transfer the fine residual to the coarse space. Here we use averaging (often the transpose of the interpolation operation) to damp out the fine space contributions. Thus, it is less accurate than injection, but avoids aliasing of the high modes.

2.8.1 Adaptive Interpolation

For a multigrid cycle, the interpolator P is intended to accurately reproduce "smooth" functions from the coarse space in the fine space, keeping the energy of the interpolant about the same. For the Laplacian on a structured mesh, it is easy to determine what these low-frequency functions are. They are the Fourier modes. However an arbitrary operator A will have different coarse modes that we want to resolve accurately on the fine grid, so that our coarse solve produces a good guess for the fine problem. How do we make sure that our interpolator P can do this?

We first must decide what we mean by accurate interpolation of some functions. Suppose we know the continuum function f that we care about, and we are only interested in a finite element description of discrete functions. Then the coarse function representing f is given by

$$f^C = \sum_i f_i^C \phi_i^C,$$

and similarly the fine grid form is

$$f^F = \sum_i f_i^F \phi_i^F.$$

Now we would like the interpolant of the coarse representer to the fine grid to be as close as possible to the fine representer in a least squares sense, meaning we want to solve the minimization problem

$$\min_P \|f^F - Pf^C\|_2$$

Now we can express P as a matrix by looking at the matrix elements $P_{ij} = \phi_i^F P \phi_j^C$. Then we have

$$\begin{aligned} &\phi_i^F f^F - \phi_i^F P f^C \\ =&f_i^F - \sum_j P_{ij} f_j^C \end{aligned}$$

so that our discrete optimization problem is

$$\min_{P_{ij}} \|f_i^F - \sum_j P_{ij} f_j^C\|_2$$

and we will treat each row of the interpolator as a separate optimization problem. We could allow an arbitrary sparsity pattern, or try to determine adaptively, as is done in sparse approximate inverse preconditioning. However, we know the supports of the basis functions in finite elements, and thus the naive sparsity pattern from local interpolation can be used.

We note here that the BAMG framework of Brannick, et. al. [BBKL11] does not use fine and coarse functions spaces, but rather a fine point/coarse point division which we will not employ here. Our general PETSc routine should work for both since the input would be the checking set (fine basis coefficients or fine space points) and the approximation set (coarse basis coefficients in the support or coarse points in the sparsity pattern).

We can easily solve the above problem using QR factorization. However, there are many smooth functions from the coarse space that we want interpolated accurately, and a single f would not constrain the values P_{ij} well. Therefore, we will use several functions $\{f_k\}$ in our minimization,

$$\begin{aligned}
&\min_{P_{ij}} \sum_k w_k \| f_i^{F,k} - \sum_j P_{ij} f_j^{C,k} \|_2 \\
=&\min_{P_{ij}} \sum_k \| \sqrt{w_k} f_i^{F,k} - \sqrt{w_k} \sum_j P_{ij} f_j^{C,k} \|_2 \\
=&\min_{P_{ij}} \| W^{1/2} \mathbf{f}_i^F - W^{1/2} \mathbf{f}^C p_i \|_2
\end{aligned}$$

where

$$\begin{aligned}
W &= \begin{pmatrix} w_0 & & \\ & \ddots & \\ & & w_K \end{pmatrix} \\
\mathbf{f}_i^F &= \begin{pmatrix} f_i^{F,0} \\ \vdots \\ f_i^{F,K} \end{pmatrix} \\
\mathbf{f}^C &= \begin{pmatrix} f_0^{C,0} & \cdots & f_n^{C,0} \\ \vdots & \ddots & \vdots \\ f_0^{C,K} & \cdots & f_n^{C,K} \end{pmatrix} \\
p_i &= \begin{pmatrix} P_{i0} \\ \vdots \\ P_{in} \end{pmatrix}
\end{aligned}$$

or alternatively

$$\begin{aligned}
{}_{kk} &= w_k \\
[f_i^F]_k &= f_i^{F,k} \\
[f^C]_{kj} &= f_j^{C,k} \\
[p_i]_j &= P_{ij}
\end{aligned}$$

We thus have a standard least-squares problem

$$\min_{P_{ij}} \|b - Ax\|_2$$

where

$$\begin{aligned}
A &= W^{1/2} f^C \\
b &= W^{1/2} f_i^F \\
x &= p_i
\end{aligned}$$

which can be solved using LAPACK.

We will typically perform this optimization on a multigrid level l when the change in eigenvalue from level $l+1$ is relatively large, meaning

$$\frac{|\lambda_l - \lambda_{l+1}|}{|\lambda_l|}.$$

This indicates that the generalized eigenvector associated with that eigenvalue was not adequately represented by P^l_{l+1}, and the interpolator should be recomputed.

2.9 DMPlex: Unstructured Grids in PETSc

This chapter introduces the `DMPLEX` subclass of `DM`, which allows the user to handle unstructured grids using the generic `DM` interface for hierarchy and multi-physics. `DMPlex` was created to remedy a huge problem in all current PDE simulation codes, namely that the discretization was so closely tied to the data layout and solver that switching discretizations in the same code was not possible. Not only does this preclude the kind of comparison that is necessary for scientific investigation, but it makes library (as opposed to monolithic application) development impossible.

2.9.1 Representing Unstructured Grids

The main advantage of `DMPlex` in representing topology is that it treats all the different pieces of a mesh, e.g. cells, faces, edges, and vertices, in exactly the same way. This allows the interface to be very small and simple, while remaining flexible and general. This also allows "dimension independent programming", which means that the same algorithm can be used unchanged for meshes of different shapes and dimensions.

All pieces of the mesh are treated as *points*, which are identified by `PetscInt`s. A mesh is built by relating points to other points, in particular specifying a "covering" relation among the points. For example, an edge is defined by being covered by two vertices, and a triangle can be defined by being covered by three edges (or even by three vertices). In fact, this structure has been known for a long time. It is a Hasse Diagram Hasse Diagram, which is a Directed Acyclic Graph (DAG) representing a cell complex using the covering relation. The graph edges represent the relation, which also encodes a partially ordered set (poset).

For example, we can encode the doublet mesh as in Fig. 2.5,

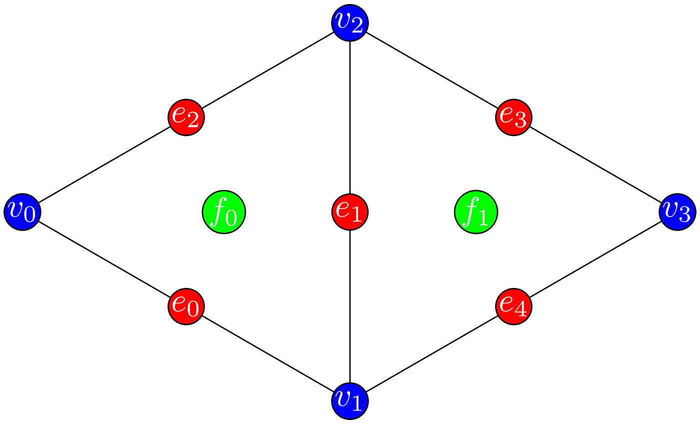


Fig. 2.5: A 2D doublet mesh, two triangles sharing an edge.

which can also be represented as the DAG in Fig. 2.6.

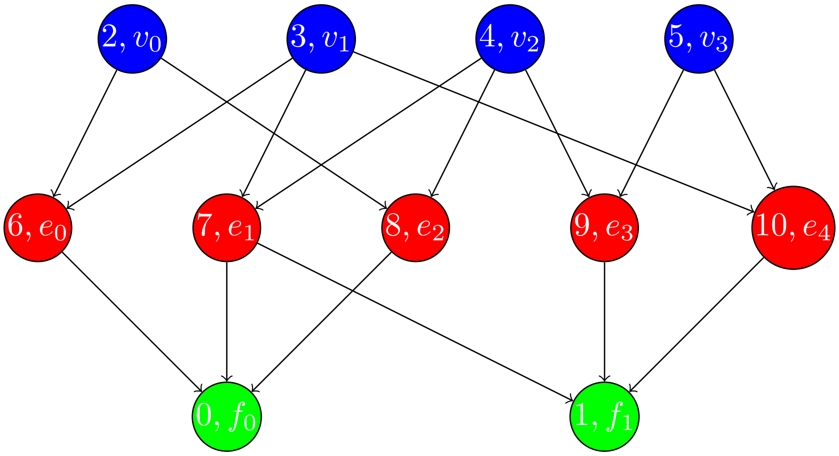


Fig. 2.6: The Hasse diagram for our 2D doublet mesh, expressed as a DAG.

To use the PETSc API, we first consecutively number the mesh pieces. The PETSc convention in 3 dimensions is to number first cells, then vertices, then faces, and then edges. In 2 dimensions the convention is to number faces, vertices, and then edges. The user is free to violate these conventions. In terms of the labels in Fig. 2.5, these numberings are

$$f_0 \mapsto \mathtt{0}, f_1 \mapsto \mathtt{1}, \quad v_0 \mapsto \mathtt{2}, v_1 \mapsto \mathtt{3}, v_2 \mapsto \mathtt{4}, v_3 \mapsto \mathtt{5}, \quad e_0 \mapsto \mathtt{6}, e_1 \mapsto \mathtt{7}, e_2 \mapsto \mathtt{8}, e_3 \mapsto \mathtt{9}, e_4 \mapsto \mathtt{10}$$

First, we declare the set of points present in a mesh,

```
DMPlexSetChart(dm, 0, 11);
```

Note that a *chart* here corresponds to a semi-closed interval (e.g $[0, 11) = \{0, 1, \ldots, 10\}$) specifying the range of indices we'd like to use to define points on the current rank. We then define the covering relation, which we call the *cone*, which are also the in-edges in the DAG. In order to preallocate correctly, we first setup sizes,

```
DMPlexSetConeSize(dm, 0, 3);
DMPlexSetConeSize(dm, 1, 3);
DMPlexSetConeSize(dm, 6, 2);
DMPlexSetConeSize(dm, 7, 2);
DMPlexSetConeSize(dm, 8, 2);
DMPlexSetConeSize(dm, 9, 2);
DMPlexSetConeSize(dm, 10, 2);
DMSetUp(dm);
```

and then point values,

```
DMPlexSetCone(dm, 0, [6, 7, 8]);
DMPlexSetCone(dm, 1, [7, 9, 10]);
DMPlexSetCone(dm, 6, [2, 3]);
DMPlexSetCone(dm, 7, [3, 4]);
DMPlexSetCone(dm, 8, [4, 2]);
DMPlexSetCone(dm, 9, [4, 5]);
DMPlexSetCone(dm, 10, [5, 3]);
```

There is also an API for the dual relation, using `DMPlexSetSupportSize()` and `DMPlexSetSupport()`, but this can be calculated automatically by calling

```
DMPlexSymmetrize(dm);
```

In order to support efficient queries, we also want to construct fast search structures and indices for the different types of points, which is done using

```
DMPlexStratify(dm);
```

2.9.2 Data on Unstructured Grids

The strongest links between solvers and discretizations are

- the layout of data over the mesh,
- problem partitioning, and
- ordering of unknowns.

To enable modularity, we encode the operations above in simple data structures that can be understood by the linear algebra engine in PETSc without any reference to the mesh (topology) or discretization (analysis).

Data Layout

Data is associated with a mesh using the `PetscSection` object. A `PetscSection` can be thought of as a generalization of `PetscLayout`, in the same way that a fiber bundle is a generalization of the normal Euclidean basis used in linear algebra. With `PetscLayout`, we associate a unit vector (e_i) with every point in the space, and just divide up points between processes. Using `PetscSection`, we can associate a set of dofs, a small space $\{e_k\}$, with every point, and though our points must be contiguous like `PetscLayout`, they can be in any range $[\mathrm{pStart}, \mathrm{pEnd})$.

The sequence for setting up any `PetscSection` is the following:

1. Specify the chart,
2. Specify the number of dofs per point, and
3. Set up the `PetscSection`.

For example, using the mesh from Fig. 2.5, we can lay out data for a continuous Galerkin P_3 finite element method,

```
PetscInt pStart, pEnd, cStart, cEnd, c, vStart, vEnd, v, eStart, eEnd, e;

DMPlexGetChart(dm, &pStart, &pEnd);
DMPlexGetHeightStratum(dm, 0, &cStart, &cEnd);   /* cells */
DMPlexGetHeightStratum(dm, 1, &eStart, &eEnd);   /* edges */
DMPlexGetHeightStratum(dm, 2, &vStart, &vEnd);   /* vertices, equivalent to␣
↪DMPlexGetDepthStratum(dm, 0, &vStart, &vEnd); */
PetscSectionSetChart(s, pStart, pEnd);
for(c = cStart; c < cEnd; ++c)
    PetscSectionSetDof(s, c, 1);
for(v = vStart; v < vEnd; ++v)
    PetscSectionSetDof(s, v, 1);
for(e = eStart; e < eEnd; ++e)
    PetscSectionSetDof(s, e, 2);
PetscSectionSetUp(s);
```

DMPlexGetHeightStratum() returns all the points of the requested height in the DAG. Since this problem is in two dimensions the edges are at height 1 and the vertices at height 2 (the cells are always at height 0). One can also use `DMPlexGetDepthStratum()` to use the depth in the DAG to select the points. `DMPlexGetDepth(,&depth)` routines the depth of the DAG, hence `DMPlexGetDepthStratum(dm, depth-1-h,)` returns the same values as `DMPlexGetHeightStratum(dm,h,)`.

For P3 elements there is one degree of freedom at each vertex, 2 along each edge (resulting in a total of 4 degrees of freedom alone each edge including the vertices, thus being able to reproduce a cubic function) and 1 degree of freedom within the cell (the bubble function which is zero along all edges).

Now a PETSc local vector can be created manually using this layout,

```
PetscSectionGetStorageSize(s, &n);
VecSetSizes(localVec, n, PETSC_DETERMINE);
VecSetFromOptions(localVec);
```

though it is usually easier to use the `DM` directly, which also provides global vectors,

```
DMSetLocalSection(dm, s);
DMGetLocalVector(dm, &localVec);
DMGetGlobalVector(dm, &globalVec);
```

Partitioning and Ordering

In exactly the same way as in `MatPartitioning` or with `MatGetOrdering()`, the results of a partition using `DMPlexPartition` or reordering using `DMPlexPermute` are encoded in an `IS`. However, the graph is not the adjacency graph of the problem Jacobian, but the mesh itself. Once the mesh is partitioned and reordered, the data layout from a `PetscSection` can be used to automatically derive a problem partitioning/ordering.

Influence of Variables on One Another

The Jacobian of a problem is intended to represent the influence of some variable on other variables in the problem. Very often, this influence pattern is determined jointly by the computational mesh and discretization. `DMCreateMatrix` must compute this pattern when it automatically creates the properly preallocated Jacobian matrix. In `DMDA` the influence pattern, or what we will call variable *adjacency*, depends only on the stencil since the topology is Cartesian and the discretization is implicitly finite difference. In `DMPlex`, we allow the user to specify the adjacency topologically, while maintaining good defaults.

The pattern is controlled by two flags. The first flag, `useCone`, indicates whether variables couple first to their boundary and then to neighboring entities, or the reverse. For example, in finite elements, the variables couple to the set of neighboring cells containing the mesh point, and we set the flag to `useCone = PETSC_FALSE`. By constrast, in finite volumes, cell variables first couple to the cell boundary, and then to the neighbors, so we set the flag to `useCone = PETSC_TRUE`. The second flag, `useClosure`, indicates whether we consider the transitive closure of the neighbor relation above, or just a single level. For example, in finite elements, the entire boundary of any cell couples to the interior, and we set the flag to `useClosure = PETSC_TRUE`. By contrast, in most finite volume methods, cells couple only across faces, and not through vertices, so we set the flag to `useClosure = PETSC_FALSE`. However, the power of this method is its flexibility. If we wanted a finite volume method that coupled all cells around a vertex, we could easily prescribe that by changing to `useClosure = PETSC_TRUE`.

2.9.3 Evaluating Residuals

The evaluation of a residual or Jacobian, for most discretizations has the following general form:

- Traverse the mesh, picking out pieces (which in general overlap),
- Extract some values from the solution vector, associated with this piece,
- Calculate some values for the piece, and
- Insert these values into the residual vector

`DMPlex` separates these different concerns by passing sets of points, which are just `PetscInt`s, from mesh traversal routines to data extraction routines and back. In this way, the `PetscSection` which structures the data inside a `Vec` does not need to know anything about the mesh inside a `DMPlex`.

The most common mesh traversal is the transitive closure of a point, which is exactly the transitive closure of a point in the DAG using the covering relation. In other words, the transitive closure consists of all points that cover the given point (generally a cell) plus all points that cover those points, etc. So in 2d the transitive closure for a cell consists of edges and vertices while in 3d it consists of faces, edges, and vertices. Note that this closure can be calculated orienting the arrows in either direction. For example, in a finite element calculation, we calculate an integral over each element, and then sum up the contributions to the basis function coefficients. The closure of the element can be expressed discretely as the transitive closure of the element point in the mesh DAG, where each point also has an orientation. Then we can retrieve the data using `PetscSection` methods,

```
PetscScalar *a;
PetscInt     numPoints, *points = NULL, p;

VecGetArray(u,&a);
DMPlexGetTransitiveClosure(dm,cell,PETSC_TRUE,&numPoints,&points);
for (p = 0; p <= numPoints*2; p += 2) {
  PetscInt dof, off, d;

  PetscSectionGetDof(section, points[p], &dof);
  PetscSectionGetOffset(section, points[p], &off);
  for (d = 0; d <= dof; ++d) {
    myfunc(a[off+d]);
  }
}
DMPlexRestoreTransitiveClosure(dm, cell, PETSC_TRUE, &numPoints, &points);
VecRestoreArray(u, &a);
```

This operation is so common that we have built a convenience method around it which returns the values in a contiguous array, correctly taking into account the orientations of various mesh points:

```
const PetscScalar *values;
PetscInt           csize;

DMPlexVecGetClosure(dm, section, u, cell, &csize, &values);
/* Do integral in quadrature loop */
DMPlexVecRestoreClosure(dm, section, u, cell, &csize, &values);
DMPlexVecSetClosure(dm, section, residual, cell, &r, ADD_VALUES);
```

A simple example of this kind of calculation is in `DMPlexComputeL2Diff_Plex()` (source). Note that there is no restriction on the type of cell or dimension of the mesh in the code above, so it will work for polyhedral cells, hybrid meshes, and meshes of any dimension, without change. We can also reverse the covering relation, so that the code works for finite volume methods where we want the data from neighboring cells for each face:

```
PetscScalar *a;
PetscInt     points[2*2], numPoints, p, dofA, offA, dofB, offB;

VecGetArray(u,  &a);
DMPlexGetTransitiveClosure(dm, cell, PETSC_FALSE, &numPoints, &points);
assert(numPoints == 2);
PetscSectionGetDof(section, points[0*2], &dofA);
PetscSectionGetDof(section, points[1*2], &dofB);
assert(dofA == dofB);
PetscSectionGetOffset(section, points[0*2], &offA);
PetscSectionGetOffset(section, points[1*2], &offB);
myfunc(a[offA], a[offB]);
VecRestoreArray(u, &a);
```

This kind of calculation is used in TS Tutorial ex11.

2.9.4 Networks

Built on top of `DMPlex`, the `DMNetwork` subclass provides abstractions for representing general unstructured networks such as communication networks, power grid, computer networks, transportation networks, electrical circuits, graphs, and others.

Application flow

The general flow of an application code using `DMNetwork` is as follows:

1. Create a network object

   ```
   DMNetworkCreate(MPI_Comm comm, DM *dm);
   ```

2. Create components and register them with the network. A “component” is specific application data at a vertex/edge of the network required for its residual evaluation. For example, components could be resistor, inductor data for circuit applications, edge weights for graph problems, generator/transmission line data for power grids. Components are registered by calling

   ```
   DMNetworkRegisterComponent(DM dm, const char *name, size_t size, PetscInt␣
   ↪*compkey);
   ```

   Here, `name` is the component name, `size` is the size of component data type, and `compkey` is an integer key that can be used for setting/getting the component at a vertex or an edge. DMNetwork currently allows upto 16 components to be registered for a network.

3. A DMNetwork can consist of one or more *physical* subnetworks. When multiple physical subnetworks are used one can (optionally) provide *coupling information between subnetworks* which consist only of edges connecting the vertices of the physical subnetworks. The topological sizes of the network are set by calling

   ```
   DMNetworkSetSizes(DM dm, PetscInt Nsubnet, PetscInt nV[], PetscInt nE[], PetscInt␣
   ↪NsubnetCouple, PetscInt nec[]);
   ```

   Here, `Nsubnet` is the number of subnetworks, `nV` and `nE` is the number of vertices and edges for each subnetwork, `NsubnetCouple` is the number of pairs of subnetworks that are coupled, and `nec` is the number of edges coupling each subnetwork pair. DMNetwork assumes coupling between the subnetworks through coupling edges. For a single network, set `Nsubnet` $= 1$, `NsubnetCouple` $= 0$, and `nec` = NULL. Note that the coupling between subnetworks is still an experimental feature and under development.

4. The next step is to set up the connectivity for the network. This is done by specifying the connectivity within each subnetwork (`edgelist`) and between subnetworks (`edgelistCouple`).

   ```
   DMNetworkSetEdgeList(DM dm, PetscInt *edgelist[], PetscInt *edgelistCouple[]);
   ```

   Each element of `edgelist` is an integer array of size 2*nE[i] containing the edge connectivity for the i-th subnetwork. Each element in `edgelistCouple` has four entries - from subnetwork number (net.id), from subnetwork vertex number (vertex.id), to subnetwork number (net.id), to subetwork vertex number (vertex.id).

   As an example, consider a network comprising of 2 subnetworks that are coupled. The topological information for the network is as follows:

   subnetwork 0: v0 — v1 — v2 — v3

   subnetwork 1: v1 — v2 — v0

   coupling between subnetworks: subnetwork 1: v2 — subnetwork 0: v0

   The `edgelist` and `edgelistCouple` for this network are

   edgelist[0] = {0,1,1,2,2,3}

   edgelist[1] = {1,2,2,0}

   edgelistCouple[0] = {1,2,0,0}.

5. The next step is to have DMNetwork to create a bare layout (graph) of the network by calling

   ```
   DMNetworkLayoutSetUp(DM dm);
   ```

6. After completing the previous steps, the network graph is set up, but no physics is associated yet. This is done by adding the components and setting the number of variables for the vertices and edges.

   A component is added to a vertex/edge by calling

   ```
   DMNetworkAddComponent(DM dm, PetscInt p, PetscInt compkey, void* compdata);
   ```

   where `p` is the network vertex/edge point in the range obtained by either DMNetworkGetEdgeRange or DMNetworkGetVertexRange, `compkey` is the component key returned when registering the component (DMNetworkRegisterComponent), and `compdata` holds the data for the component. DMNetwork supports setting multiple components (max. 36) at a vertex/edge.

   DMNetwork currently assumes the component data to be stored in a contiguous chunk of memory. As such, it does not do any packing/unpacking before/after the component data gets distributed. Any such serialization (packing/unpacking) should be done by the application.

   The number of variables at each vertex/edge are set by

   ```
   DMNetworkSetNumVariables(DM dm, PetscInt p, PetscInt nvar);
   ```

   or

   ```
   DMNetworkAddNumVariables(DM dm, PetscInt p, PetscInt nvar);
   ```

   Alternatively, the number of variables can be set for a component directly. This allows much finer control, specifically for vertices/edges that have multiple components set on them.

```
DMNetworkSetComponentNumVariables(DM dm, PetscInt p, PetscInt compnum, PetscInt␣
↪nvar);
```

7. Set up network internal data structures.

```
DMSetUp(DM dm);
```

8. Distribute the network (also moves components attached with vertices/edges) to multiple processors.

```
DMNetworkDistribute(DM dm, const char partitioner[], PetscInt overlap, DM␣
↪*distDM);
```

9. Associate the `DM` with a PETSc solver:

```
KSPSetDM(KSP ksp, DM dm) or SNESSetDM(SNES snes, DM dm) or TSSetDM(TS ts, DM dm).
```

Utility functions

`DMNetwork` provides several utility functions for operations on the network. The mostly commonly used functions are: obtaining iterators for vertices/edges,

```
DMNetworkGetEdgeRange(DM dm, PetscInt *eStart, PetscInt *eEnd);
```

```
DMNetworkGetVertexRange(DM dm, PetscInt *vStart, PetscInt *vEnd);
```

```
DMNetworkGetSubnetworkInfo(DM dm, PetscInt netid, PetscInt *nv, PetscInt *ne, const␣
↪PetscInt **vtx, const PetscInt **edge);
```

Checking the "ghost" status of a vertex,

```
DMNetworkIsGhostVertex(DM dm, PetscInt p, PetscBool *isghost);
```

and retrieving local/global indices of vertex/edge variables for inserting elements in vectors/matrices.

```
DMNetworkGetVariableOffset(DM dm, PetscInt p, PetscInt *offset);
```

```
DMNetworkGetVariableGlobalOffset(DM dm, PetscInt p, PetscInt *offsetg);
```

If the number of variables are set at the component level, then their local/global offsets can be retrieved via

```
DMNetworkGetComponentVariableOffset(DM dm, PetscInt p, PetscInt compnum, PetscInt␣
↪*offset);
```

```
DMNetworkGetComponentVariableGlobalOffset(DM dm, PetscInt p, PetscInt compnum,␣
↪PetscInt *offsetg);
```

In network applications, one frequently needs to find the supporting edges for a vertex or the connecting vertices covering an edge. These can be obtained by the following two routines.

```
DMNetworkGetConnectedVertices(DM dm, PetscInt edge, const PetscInt *vertices[]);
```

```
DMNetworkGetSupportingEdges(DM dm, PetscInt vertex, PetscInt *nedges, const PetscInt␣
↪*edges[]);
```

Retrieving components

The components set at a vertex/edge can be accessed by

```
DMNetworkGetComponent(DM dm, PetscInt p, PetscInt compnum, PetscInt *compkey, void**␣
↪component);
```

`compkey` is the key set by `DMNetworkRegisterComponent`. An example of accessing and retrieving the components at vertices is:

```
PetscInt Start, End, numcomps,key,v,compnum;
void *component;

DMNetworkGetVertexRange(dm, &Start, &End);
for (v=Start; v  < End; v++) {
  DMNetworkGetNumComponents(dm,v, &numcomps);
  for (compnum=0; compnum < numcomps;compnum++) {
    DMNetworkGetComponent(dm,v,compnum, &key, &component);
    compdata = (UserCompDataType)(component);
  }
}
```

The above example does not explicitly make use the component key. It is used when different component types are set at different vertices. In this case, the compkey is used to differentiate the component type.

CHAPTER

THREE

ADDITIONAL INFORMATION

3.1 PETSc for Fortran Users

Most of the functionality of PETSc can be obtained by people who program purely in Fortran.

3.1.1 C vs. Fortran Interfaces

Only a few differences exist between the C and Fortran PETSc interfaces, are due to Fortran syntax differences. All Fortran routines have the same names as the corresponding C versions, and PETSc command line options are fully supported. The routine arguments follow the usual Fortran conventions; the user need not worry about passing pointers or values. The calling sequences for the Fortran version are in most cases identical to the C version, except for the error checking variable discussed in *Error Checking* and a few routines listed in *Routines with Different Fortran Interfaces*.

Fortran Include Files

The Fortran include files for PETSc are located in the directory `${PETSC_DIR}/include/petsc/finclude` and should be used via statements such as the following:

```
#include <petsc/finclude/petscXXX.h>
```

for example,

```
#include <petsc/finclude/petscksp.h>
```

You must also use the appropriate Fortran module which is done with

```
use petscXXX
```

for example,

```
use petscksp
```

Error Checking

In the Fortran version, each PETSc routine has as its final argument an integer error variable, in contrast to the C convention of providing the error variable as the routine's return value. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For example, the Fortran and C variants of `KSPSolve()` are given, respectively, below, where `ierr` denotes the error variable:

```
call KSPSolve(ksp,b,x,ierr) ! Fortran
ierr = KSPSolve(ksp,b,x);   /* C */
```

Fortran programmers can check these error codes with `CHKERRQ(ierr)`, which terminates all processes when an error is encountered. Likewise, one can set error codes within Fortran programs by using `SETERRQ(comm,p,' ',ierr)`, which again terminates all processes upon detection of an error. Note that complete error tracebacks with `CHKERRQ()` and `SETERRQ()`, as described in *Simple PETSc Examples* for C routines, are *not* directly supported for Fortran routines; however, Fortran programmers can easily use the error codes in writing their own tracebacks. For example, one could use code such as the following:

```
call KSPSolve(ksp,b,x,ierr)
if (ierr .ne. 0) then
   print*, 'Error in routine ...'
   return
end if
```

Calling Fortran Routines from C (and C Routines from Fortran)

Different machines have different methods of naming Fortran routines called from C (or C routines called from Fortran). Most Fortran compilers change all the capital letters in Fortran routines to lowercase. On some machines, the Fortran compiler appends an underscore to the end of each Fortran routine name; for example, the Fortran routine `Dabsc()` would be called from C with `dabsc_()`. Other machines change all the letters in Fortran routine names to capitals.

PETSc provides two macros (defined in C/C++) to help write portable code that mixes C/C++ and Fortran. They are `PETSC_HAVE_FORTRAN_UNDERSCORE` and `PETSC_HAVE_FORTRAN_CAPS` , which are defined in the file `${PETSC_DIR}/${PETSC_ARCH}/include/petscconf.h`. The macros are used, for example, as follows:

```
#if defined(PETSC_HAVE_FORTRAN_CAPS)
#define dabsc_ DMDABSC
#elif !defined(PETSC_HAVE_FORTRAN_UNDERSCORE)
#define dabsc_ dabsc
#endif
.....
dabsc_( &n,x,y); /* call the Fortran function */
```

Passing Null Pointers

In several PETSc C functions, one has the option of passing a NULL (0) argument (for example, the fifth argument of `MatCreateSeqAIJ()`). From Fortran, users *must* pass `PETSC_NULL_XXX` to indicate a null argument (where `XXX` is `INTEGER`, `DOUBLE`, `CHARACTER`, or `SCALAR` depending on the type of argument required); passing 0 from Fortran will crash the code. Note that the C convention of passing NULL (or 0) *cannot* be used. For example, when no options prefix is desired in the routine `PetscOptionsGetInt()`, one must use the following command in Fortran:

```
call PetscOptionsGetInt(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER,PETSC_NULL_CHARACTER,
↪'-name',N,flg,ierr)
```

This Fortran requirement is inconsistent with C, where the user can employ `NULL` for all null arguments.

Duplicating Multiple Vectors

The Fortran interface to `VecDuplicateVecs()` differs slightly from the C/C++ variant because Fortran does not allow conventional arrays to be returned in routine arguments. To create `n` vectors of the same format as an existing vector, the user must declare a vector array, `v_new` of size `n`. Then, after `VecDuplicateVecs()` has been called, `v_new` will contain (pointers to) the new PETSc vector objects. When finished with the vectors, the user should destroy them by calling `VecDestroyVecs()`. For example, the following code fragment duplicates `v_old` to form two new vectors, `v_new(1)` and `v_new(2)`.

```
Vec          v_old, v_new(2)
PetscInt     ierr
PetscScalar  alpha
....
call VecDuplicateVecs(v_old,2,v_new,ierr)
alpha = 4.3
call VecSet(v_new(1),alpha,ierr)
alpha = 6.0
call VecSet(v_new(2),alpha,ierr)
....
call VecDestroyVecs(2, &v_new,ierr)
```

Matrix, Vector and IS Indices

All matrices, vectors and `IS` in PETSc use zero-based indexing, regardless of whether C or Fortran is being used. The interface routines, such as `MatSetValues()` and `VecSetValues()`, always use zero indexing. See *Basic Matrix Operations* for further details.

Setting Routines

When a function pointer is passed as an argument to a PETSc function, such as the test in `KSPSetConvergenceTest()`, it is assumed that this pointer references a routine written in the same language as the PETSc interface function that was called. For instance, if `KSPSetConvergenceTest()` is called from C, the test argument is assumed to be a C function. Likewise, if it is called from Fortran, the test is assumed to be written in Fortran.

Compiling and Linking Fortran Programs

See *Writing Application Codes with PETSc*.

Routines with Different Fortran Interfaces

The following Fortran routines differ slightly from their C counterparts; see the manual pages and previous discussion in this chapter for details:

```
PetscInitialize(char *filename,int ierr)
PetscError(MPI_COMM,int err,char *message,int ierr)
VecGetArray(), MatDenseGetArray()
ISGetIndices(),
VecDuplicateVecs(), VecDestroyVecs()
PetscOptionsGetString()
```

The following functions are not supported in Fortran:

```
PetscFClose(), PetscFOpen(), PetscFPrintf(), PetscPrintf()
PetscPopErrorHandler(), PetscPushErrorHandler()
PetscInfo()
PetscSetDebugger()
VecGetArrays(), VecRestoreArrays()
PetscViewerASCIIGetPointer(), PetscViewerBinaryGetDescriptor()
PetscViewerStringOpen(), PetscViewerStringSPrintf()
PetscOptionsGetStringArray()
```

PETSc includes some support for direct use of Fortran90 pointers. Current routines include:

```
VecGetArrayF90(), VecRestoreArrayF90()
VecGetArrayReadF90(), VecRestoreArrayReadF90()
VecDuplicateVecsF90(), VecDestroyVecsF90()
DMDAVecGetArrayF90(), DMDAVecGetArrayReadF90(), ISLocalToGlobalMappingGetIndicesF90()
MatDenseGetArrayF90(), MatDenseRestoreArrayF90()
ISGetIndicesF90(), ISRestoreIndicesF90()
```

See the manual pages for details and pointers to example programs.

3.1.2 Sample Fortran Programs

Sample programs that illustrate the PETSc interface for Fortran are given below, corresponding to Vec Test ex19f, Vec Tutorial ex4f, Draw Test ex5f, and SNES Tutorial ex1f, respectively. We also refer Fortran programmers to the C examples listed throughout the manual, since PETSc usage within the two languages differs only slightly.

Listing: `src/vec/vec/tests/ex19f.F`

```
!
!
      program main
#include <petsc/finclude/petscvec.h>
      use petscvec
      implicit none
!
!  This example demonstrates basic use of the PETSc Fortran interface
!  to vectors.
!
       PetscInt  n
       PetscErrorCode ierr
```

(continues on next page)

(continued from previous page)

```
       PetscBool  flg
       PetscScalar      one,two,three,dot
       PetscReal        norm,rdot
       Vec              x,y,w
       PetscOptions     options

       n     = 20
       one   = 1.0
       two   = 2.0
       three = 3.0

       call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
       if (ierr .ne. 0) then
         print*,'Unable to initialize PETSc'
         stop
       endif
       call PetscOptionsCreate(options,ierr)
       call PetscOptionsGetInt(options,PETSC_NULL_CHARACTER,                  &
     &                        '-n',n,flg,ierr)
       call PetscOptionsDestroy(options,ierr)

! Create a vector, then duplicate it
       call VecCreate(PETSC_COMM_WORLD,x,ierr)
       call VecSetSizes(x,PETSC_DECIDE,n,ierr)
       call VecSetFromOptions(x,ierr)
       call VecDuplicate(x,y,ierr)
       call VecDuplicate(x,w,ierr)

       call VecSet(x,one,ierr)
       call VecSet(y,two,ierr)

       call VecDot(x,y,dot,ierr)
       rdot = PetscRealPart(dot)
       write(6,100) rdot
  100  format('Result of inner product ',f10.4)

       call VecScale(x,two,ierr)
       call VecNorm(x,NORM_2,norm,ierr)
       write(6,110) norm
  110  format('Result of scaling ',f10.4)

       call VecCopy(x,w,ierr)
       call VecNorm(w,NORM_2,norm,ierr)
       write(6,120) norm
  120  format('Result of copy ',f10.4)

       call VecAXPY(y,three,x,ierr)
       call VecNorm(y,NORM_2,norm,ierr)
       write(6,130) norm
  130  format('Result of axpy ',f10.4)

       call VecDestroy(x,ierr)
       call VecDestroy(y,ierr)
       call VecDestroy(w,ierr)
       call PetscFinalize(ierr)
       end
```

(continues on next page)

(continued from previous page)

```
!/*TEST
!
!     test:
!
!TEST*/
```

Listing: **`src/vec/vec/tutorials/ex4f.F`**

```
!
!
!  Description:  Illustrates the use of VecSetValues() to set
!  multiple values at once; demonstrates VecGetArray().
!
!/*T
!   Concepts: vectors^assembling;
!   Concepts: vectors^arrays of vectors;
!   Processors: 1
!T*/
! -----------------------------------------------------------------------

      program main
#include <petsc/finclude/petscvec.h>
      use petscvec
      implicit none

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Macro definitions
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Macros to make clearer the process of setting values in vectors and
!  getting values from vectors.
!
!   - The element xx_a(ib) is element ib+1 in the vector x
!   - Here we add 1 to the base array index to facilitate the use of
!     conventional Fortran 1-based array indexing.
!
#define xx_a(ib)  xx_v(xx_i + (ib))
#define yy_a(ib)  yy_v(yy_i + (ib))

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

       PetscScalar xwork(6)
       PetscScalar xx_v(1),yy_v(1)
       PetscInt     i,n,loc(6),isix
       PetscErrorCode ierr
       PetscOffset xx_i,yy_i
       Vec         x,y

       call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
       if (ierr .ne. 0) then
```

(continues on next page)

(continued from previous page)

```
          print*,'PetscInitialize failed'
          stop
        endif
        n = 6
        isix = 6

!  Create initial vector and duplicate it

        call VecCreateSeq(PETSC_COMM_SELF,n,x,ierr)
        call VecDuplicate(x,y,ierr)

!  Fill work arrays with vector entries and locations.  Note that
!  the vector indices are 0-based in PETSc (for both Fortran and
!  C vectors)

        do 10 i=1,n
           loc(i) = i-1
           xwork(i) = 10.0*real(i)
  10    continue

!  Set vector values.  Note that we set multiple entries at once.
!  Of course, usually one would create a work array that is the
!  natural size for a particular problem (not one that is as long
!  as the full vector).

        call VecSetValues(x,isix,loc,xwork,INSERT_VALUES,ierr)

!  Assemble vector

        call VecAssemblyBegin(x,ierr)
        call VecAssemblyEnd(x,ierr)

!  View vector
        call PetscObjectSetName(x, 'initial vector:',ierr)
        call VecView(x,PETSC_VIEWER_STDOUT_SELF,ierr)
        call VecCopy(x,y,ierr)

!  Get a pointer to vector data.
!    - For default PETSc vectors, VecGetArray() returns a pointer to
!      the data array.  Otherwise, the routine is implementation dependent.
!    - You MUST call VecRestoreArray() when you no longer need access to
!      the array.
!    - Note that the Fortran interface to VecGetArray() differs from the
!      C version.  See the users manual for details.

        call VecGetArray(x,xx_v,xx_i,ierr)
        call VecGetArray(y,yy_v,yy_i,ierr)

!  Modify vector data

        do 30 i=1,n
           xx_a(i) = 100.0*real(i)
           yy_a(i) = 1000.0*real(i)
  30    continue

!  Restore vectors
```

(continues on next page)

(continued from previous page)

```
      call VecRestoreArray(x,xx_v,xx_i,ierr)
      call VecRestoreArray(y,yy_v,yy_i,ierr)

!  View vectors
      call PetscObjectSetName(x, 'new vector 1:',ierr)
      call VecView(x,PETSC_VIEWER_STDOUT_SELF,ierr)

      call PetscObjectSetName(y, 'new vector 2:',ierr)
      call VecView(y,PETSC_VIEWER_STDOUT_SELF,ierr)

!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.

      call VecDestroy(x,ierr)
      call VecDestroy(y,ierr)
      call PetscFinalize(ierr)
      end


!/*TEST
!
!     test:
!
!TEST*/
```

Listing: **`src/sys/classes/draw/tests/ex5f.F`**

```
!
!
      program main
#include <petsc/finclude/petscsys.h>
#include <petsc/finclude/petscdraw.h>
      use petscsys
      implicit none
!
!  This example demonstrates basic use of the Fortran interface for
!  PetscDraw routines.
!
      PetscDraw         draw
      PetscDrawLG       lg
      PetscDrawAxis     axis
      PetscErrorCode    ierr
      PetscBool         flg
      integer           x,y,width,height
      PetscScalar       xd,yd
      PetscReal         ten
      PetscInt          i,n,w,h
      PetscInt          one

      n      = 15
      x      = 0
      y      = 0
      w      = 400
```

(continues on next page)

(continued from previous page)

```
      h      = 300
      ten    = 10.0
      one    = 1

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      if (ierr .ne. 0) then
         print*,'Unable to initialize PETSc'
         stop
      endif

!  GetInt requires a PetscInt so have to do this ugly setting
      call PetscOptionsGetInt(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER,         &
     &                        '-width',w, flg,ierr)
      width = int(w)
      call PetscOptionsGetInt(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER,         &
     &                        '-height',h,flg,ierr)
      height = int(h)
      call PetscOptionsGetInt(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER,         &
     &                        '-n',n,flg,ierr)

      call PetscDrawCreate(PETSC_COMM_WORLD,PETSC_NULL_CHARACTER,        &
     &              PETSC_NULL_CHARACTER,x,y,width,height,draw,ierr)
      call PetscDrawSetFromOptions(draw,ierr)

      call PetscDrawLGCreate(draw,one,lg,ierr)
      call PetscDrawLGGetAxis(lg,axis,ierr)
      call PetscDrawAxisSetColors(axis,PETSC_DRAW_BLACK,PETSC_DRAW_RED, &
     &     PETSC_DRAW_BLUE,ierr)
      call PetscDrawAxisSetLabels(axis,'toplabel','xlabel','ylabel',    &
     &     ierr)

      do 10, i=0,n-1
        xd = real(i) - 5.0
        yd = xd*xd
        call PetscDrawLGAddPoint(lg,xd,yd,ierr)
 10   continue

      call PetscDrawLGSetUseMarkers(lg,PETSC_TRUE,ierr)
      call PetscDrawLGDraw(lg,ierr)

      call PetscSleep(ten,ierr)

      call PetscDrawLGDestroy(lg,ierr)
      call PetscDrawDestroy(draw,ierr)
      call PetscFinalize(ierr)
      end

!/*TEST
!
!   build:
!     requires: x
!
!   test:
!     output_file: output/ex1_1.out
!
!TEST*/
```

Listing: `src/snes/tutorials/ex1f.F90`

```
!
!
!  Description: Uses the Newton method to solve a two-variable system.
!
!!/*T
!  Concepts: SNES^basic uniprocessor example
!  Processors: 1
!T*/

      program main
#include <petsc/finclude/petsc.h>
      use petsc
      implicit none

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Variable declarations
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     snes        - nonlinear solver
!     ksp        - linear solver
!     pc          - preconditioner context
!     ksp         - Krylov subspace method context
!     x, r        - solution, residual vectors
!     J           - Jacobian matrix
!     its         - iterations for convergence
!
      SNES     snes
      PC       pc
      KSP      ksp
      Vec      x,r
      Mat      J
      SNESLineSearch linesearch
      PetscErrorCode  ierr
      PetscInt its,i2,i20
      PetscMPIInt size,rank
      PetscScalar   pfive
      PetscReal   tol
      PetscBool   setls
#if defined(PETSC_USE_LOG)
      PetscViewer viewer
#endif
      double precision threshold,oldthreshold

!  Note: Any user-defined Fortran routines (such as FormJacobian)
!  MUST be declared as external.

      external FormFunction, FormJacobian, MyLineSearch

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Macro definitions
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Macros to make clearer the process of setting values in vectors and
!  getting values from vectors.  These vectors are used in the routines
```

(continues on next page)

(continued from previous page)

```
!  FormFunction() and FormJacobian().
!   - The element lx_a(ib) is element ib in the vector x
!
#define lx_a(ib) lx_v(lx_i + (ib))
#define lf_a(ib) lf_v(lf_i + (ib))
!
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      if (ierr .ne. 0) then
        print*,'Unable to initialize PETSc'
        stop
      endif
      call PetscLogNestedBegin(ierr);CHKERRA(ierr)
      threshold = 1.0
      call PetscLogSetThreshold(threshold,oldthreshold,ierr)
! dummy test of logging a reduction
#if defined(PETSC_USE_LOG)
      ierr = PetscAReduce()
#endif
      call MPI_Comm_size(PETSC_COMM_WORLD,size,ierr)
      call MPI_Comm_rank(PETSC_COMM_WORLD,rank,ierr)
      if (size .ne. 1) then; SETERRA(PETSC_COMM_SELF,PETSC_ERR_WRONG_MPI_SIZE,
↪'Uniprocessor example'); endif

      i2  = 2
      i20 = 20
! - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - -
!  Create nonlinear solver context
! - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - -

      call SNESCreate(PETSC_COMM_WORLD,snes,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Create matrix and vector data structures; set corresponding routines
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Create vectors for solution and nonlinear function

      call VecCreateSeq(PETSC_COMM_SELF,i2,x,ierr)
      call VecDuplicate(x,r,ierr)

!  Create Jacobian matrix data structure

      call MatCreate(PETSC_COMM_SELF,J,ierr)
      call MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,i2,i2,ierr)
      call MatSetFromOptions(J,ierr)
      call MatSetUp(J,ierr)

!  Set function evaluation routine and vector

      call SNESSetFunction(snes,r,FormFunction,0,ierr)

!  Set Jacobian matrix data structure and Jacobian evaluation routine
```

(continues on next page)

(continued from previous page)

```
      call SNESSetJacobian(snes,J,J,FormJacobian,0,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Customize nonlinear solver; set runtime options
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Set linear solver defaults for this problem. By extracting the
!  KSP, KSP, and PC contexts from the SNES context, we can then
!  directly call any KSP, KSP, and PC routines to set various options.

      call SNESGetKSP(snes,ksp,ierr)
      call KSPGetPC(ksp,pc,ierr)
      call PCSetType(pc,PCNONE,ierr)
      tol = 1.e-4
      call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_REAL,                  &
     &                      PETSC_DEFAULT_REAL,i20,ierr)

!  Set SNES/KSP/KSP/PC runtime options, e.g.,
!      -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
!  These options will override those specified above as long as
!  SNESSetFromOptions() is called _after_ any other customization
!  routines.


      call SNESSetFromOptions(snes,ierr)

      call PetscOptionsHasName(PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER,   &
     &                         '-setls',setls,ierr)

      if (setls) then
        call SNESGetLineSearch(snes, linesearch, ierr)
        call SNESLineSearchSetType(linesearch, 'shell', ierr)
        call SNESLineSearchShellSetUserFunc(linesearch, MyLineSearch,   &
     &                                      0, ierr)
      endif

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Evaluate initial guess; then solve nonlinear system
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Note: The user should initialize the vector, x, with the initial guess
!  for the nonlinear solver prior to calling SNESSolve().  In particular,
!  to employ an initial guess of zero, the user should explicitly set
!  this vector to zero by calling VecSet().

      pfive = 0.5
      call VecSet(x,pfive,ierr)
      call SNESSolve(snes,PETSC_NULL_VEC,x,ierr)

!  View solver converged reason; we could instead use the option -snes_converged_
↪reason
      call SNESConvergedReasonView(snes,PETSC_VIEWER_STDOUT_WORLD,ierr)

      call SNESGetIterationNumber(snes,its,ierr);
      if (rank .eq. 0) then
```

(continues on next page)

(continued from previous page)

```
          write(6,100) its
       endif
  100 format('Number of SNES iterations = ',i5)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call VecDestroy(x,ierr)
      call VecDestroy(r,ierr)
      call MatDestroy(J,ierr)
      call SNESDestroy(snes,ierr)
#if defined(PETSC_USE_LOG)
      call PetscViewerASCIIOpen(PETSC_COMM_WORLD,'filename.xml',viewer,ierr)
      call PetscViewerPushFormat(viewer,PETSC_VIEWER_ASCII_XML,ierr)
      call PetscLogView(viewer,ierr)
      call PetscViewerDestroy(viewer,ierr)
#endif
      call PetscFinalize(ierr)
      end
!
! ---------------------------------------------------------------------
!
!  FormFunction - Evaluates nonlinear function, F(x).
!
!  Input Parameters:
!  snes - the SNES context
!  x - input vector
!  dummy - optional user-defined context (not used here)
!
!  Output Parameter:
!  f - function vector
!
      subroutine FormFunction(snes,x,f,dummy,ierr)
      use petscsnes
      implicit none

      SNES     snes
      Vec      x,f
      PetscErrorCode ierr
      integer dummy(*)

!  Declarations for use with local arrays

      PetscScalar  lx_v(2),lf_v(2)
      PetscOffset  lx_i,lf_i

!  Get pointers to vector data.
!    - For default PETSc vectors, VecGetArray() returns a pointer to
!      the data array.  Otherwise, the routine is implementation dependent.
!    - You MUST call VecRestoreArray() when you no longer need access to
!      the array.
!    - Note that the Fortran interface to VecGetArray() differs from the
!      C version.  See the Fortran chapter of the users manual for details.
```

(continues on next page)

(continued from previous page)

```
      call VecGetArrayRead(x,lx_v,lx_i,ierr)
      call VecGetArray(f,lf_v,lf_i,ierr)

!  Compute function

      lf_a(1) = lx_a(1)*lx_a(1)                                       &
     &          + lx_a(1)*lx_a(2) - 3.0
      lf_a(2) = lx_a(1)*lx_a(2)                                       &
     &          + lx_a(2)*lx_a(2) - 6.0

!  Restore vectors

      call VecRestoreArrayRead(x,lx_v,lx_i,ierr)
      call VecRestoreArray(f,lf_v,lf_i,ierr)

      return
      end

! ---------------------------------------------------------------------
!
!  FormJacobian - Evaluates Jacobian matrix.
!
!  Input Parameters:
!  snes - the SNES context
!  x - input vector
!  dummy - optional user-defined context (not used here)
!
!  Output Parameters:
!  A - Jacobian matrix
!  B - optionally different preconditioning matrix
!  flag - flag indicating matrix structure
!
      subroutine FormJacobian(snes,X,jac,B,dummy,ierr)
      use petscsnes
      implicit none

      SNES         snes
      Vec          X
      Mat          jac,B
      PetscScalar  A(4)
      PetscErrorCode ierr
      PetscInt idx(2),i2
      integer dummy(*)

!  Declarations for use with local arrays

      PetscScalar lx_v(2)
      PetscOffset lx_i

!  Get pointer to vector data

      i2 = 2
      call VecGetArrayRead(x,lx_v,lx_i,ierr)

!  Compute Jacobian entries and insert into matrix.
!   - Since this is such a small problem, we set all entries for
```

(continues on next page)

(continued from previous page)

```
!     the matrix at once.
!   - Note that MatSetValues() uses 0-based row and column numbers
!     in Fortran as well as in C (as set here in the array idx).

      idx(1) = 0
      idx(2) = 1
      A(1) = 2.0*lx_a(1) + lx_a(2)
      A(2) = lx_a(1)
      A(3) = lx_a(2)
      A(4) = lx_a(1) + 2.0*lx_a(2)
      call MatSetValues(B,i2,idx,i2,idx,A,INSERT_VALUES,ierr)

!  Restore vector

      call VecRestoreArrayRead(x,lx_v,lx_i,ierr)

!  Assemble matrix

      call MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY,ierr)
      call MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY,ierr)
      if (B .ne. jac) then
        call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
        call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)
      endif

      return
      end


      subroutine MyLineSearch(linesearch, lctx, ierr)
      use petscsnes
      implicit none

      SNESLineSearch    linesearch
      SNES              snes
      integer           lctx
      Vec               x, f,g, y, w
      PetscReal         ynorm,gnorm,xnorm
      PetscBool         flag
      PetscErrorCode    ierr

      PetscScalar       mone

      mone = -1.0
      call SNESLineSearchGetSNES(linesearch, snes, ierr)
      call SNESLineSearchGetVecs(linesearch, x, f, y, w, g, ierr)
      call VecNorm(y,NORM_2,ynorm,ierr)
      call VecAXPY(x,mone,y,ierr)
      call SNESComputeFunction(snes,x,f,ierr)
      call VecNorm(f,NORM_2,gnorm,ierr)
      call VecNorm(x,NORM_2,xnorm,ierr)
      call VecNorm(y,NORM_2,ynorm,ierr)
      call SNESLineSearchSetNorms(linesearch, xnorm, gnorm, ynorm,      &
     & ierr)
      flag = PETSC_FALSE
      return
```

(continues on next page)

(continued from previous page)

```
      end

!/*TEST
!
!   test:
!      args: -ksp_gmres_cgs_refinement_type refine_always -snes_monitor_short
!      requires: !single
!
!TEST*/
```

Array Arguments

This material is no longer relevent since one should use `VecGetArrayF90()` and the other routines that utilize Fortran pointers, instead of the code below, but it is included for historical reasons and because many of the Fortran examples still utilize the old approach.

Since Fortran 77 does not allow arrays to be returned in routine arguments, all PETSc routines that return arrays, such as `VecGetArray()`, `MatDenseGetArray()`, and `ISGetIndices()`, are defined slightly differently in Fortran than in C. Instead of returning the array itself, these routines accept as input a user-specified array of dimension one and return an integer index to the actual array used for data storage within PETSc. The Fortran interface for several routines is as follows:

```
PetscScalar    xx_v(1), aa_v(1)
PetscErrorCode ierr
PetscInt       ss_v(1), dd_v(1), nloc
PetscOffset    ss_i, xx_i, aa_i, dd_i
Vec            x
Mat            A
IS             s
DM             d

call VecGetArray(x,xx_v,xx_i,ierr)
call MatDenseGetArray(A,aa_v,aa_i,ierr)
call ISGetIndices(s,ss_v,ss_i,ierr)
```

To access array elements directly, both the user-specified array and the integer index *must* then be used together. For example, the following Fortran program fragment illustrates directly setting the values of a vector array instead of using `VecSetValues()`. Note the (optional) use of the preprocessor `#define` statement to enable array manipulations in the conventional Fortran manner.

```
#define xx_a(ib)  xx_v(xx_i + (ib))

   double precision xx_v(1)
   PetscOffset      xx_i
   PetscErrorCode   ierr
   PetscInt         i, n
   Vec              x
   call VecGetArray(x,xx_v,xx_i,ierr)
   call VecGetLocalSize(x,n,ierr)
   do 10, i=1,n
     xx_a(i) = 3*i + 1
10 continue
   call VecRestoreArray(x,xx_v,xx_i,ierr)
```

The Vec ex4f Tutorial listed above contains an example of using `VecGetArray()` within a Fortran routine.

Since in this case the array is accessed directly from Fortran, indexing begins with 1, not 0 (unless the array is declared as `xx_v(0:1)`). This is different from the use of `VecSetValues()` where, indexing always starts with 0.

Note: If using `VecGetArray()`, `MatDenseGetArray()`, or `ISGetIndices()`, from Fortran, the user *must not* compile the Fortran code with options to check for “array entries out of bounds” (e.g., on the IBM RS/6000 this is done with the `-C` compiler option, so never use the `-C` option with this).

3.2 Using MATLAB with PETSc

There are three basic ways to use MATLAB with PETSc:

1. (*Dumping Data for MATLAB*) dumping files to be read into MATLAB,
2. (*Sending Data to an Interactive MATLAB Session*) automatically sending data from a running PETSc program to a MATLAB process where you may interactively type MATLAB commands (or run scripts), and
3. (*Using the MATLAB Compute Engine*) automatically sending data back and forth between PETSc and MATLAB where MATLAB commands are issued not interactively but from a script or the PETSc program (this uses the MATLAB Engine).

3.2.1 Dumping Data for MATLAB

Dumping ASCII MATLAB data

One can dump PETSc matrices and vectors to the screen in an ASCII format that MATLAB can read in directly. This is done with the command line options `-vec_view ::ascii_matlab` or `-mat_view ::ascii_matlab`. To write a a file, use `-vec_view :filename.m:ascii_matlab` or `-mat_view :filename.m:ascii_matlab`.

This causes the PETSc program to print the vectors and matrices every time `VecAssemblyEnd()` or `MatAssemblyEnd()` are called. To provide finer control over when and what vectors and matrices are dumped one can use the `VecView()` and `MatView()` functions with a viewer type of ASCII (see `PetscViewerASCIIOpen()`, `PETSC_VIEWER_STDOUT_WORLD`, `PETSC_VIEWER_STDOUT_SELF`, or `PETSC_VIEWER_STDOUT_(MPI_Comm)`). Before calling the viewer set the output type with, for example,

```
PetscViewerPushFormat(PETSC_VIEWER_STDOUT_WORLD,PETSC_VIEWER_ASCII_MATLAB);
VecView(A,PETSC_VIEWER_STDOUT_WORLD);
PetscViewerPopFormat(PETSC_VIEWER_STDOUT_WORLD);
```

The name of each PETSc variable printed for MATLAB may be set with

```
PetscObjectSetName((PetscObject)A,"name");
```

If no name is specified, the object is given a default name using `PetscObjectName()`.

Dumping Binary Data for MATLAB

One can also read PETSc binary files (see *Viewers: Looking at PETSc Objects*) directly into MATLAB via the scripts available in `$PETSC_DIR/share/matlab`. This requires less disk space and is recommended for all but the smallest data sizes. One can also use

```
PetscViewerPushFormat(viewer,PETSC_VIEWER_BINARY_MATLAB)
```

to dump both a PETSc binary file and a corresponding `.info` file which `PetscReadBinaryMatlab.m` will use to format the binary file in more complex cases, such as using a `DMDA`. For an example, see DM Tutorial ex7. In MATLAB (R2015b), one may then generate a useful structure. For example:

```
setenv('PETSC_DIR','~/petsc');
setenv('PETSC_ARCH','arch-darwin-double-debug');
addpath('~/petsc/share/petsc/matlab');
gridData=PetscReadBinaryMatlab('output_file');
```

3.2.2 Sending Data to an Interactive MATLAB Session

One creates a viewer to MATLAB via

```
PetscViewerSocketOpen(MPI_Comm,char *machine,int port,PetscViewer *v);
```

(`port` is usually set to `PETSC_DEFAULT`; use `NULL` for the machine if the MATLAB interactive session is running on the same machine as the PETSc program) and then sends matrices or vectors via

```
VecView(Vec A,v);
MatView(Mat B,v);
```

See *Viewers: Looking at PETSc Objects* for more on PETSc viewers. One may start the MATLAB program manually or use the PETSc command `PetscStartMatlab(MPI_Comm,char *machine,char *script,FILE **fp)`; where `machine` and `script` may be `NULL`. It is also possible to start your PETSc program from MATLAB via `launch()`.

To receive the objects in MATLAB, make sure that `${PETSC_DIR}/${PETSC_ARCH}/lib/petsc/matlab` and `${PETSC_DIR}/share/petsc/matlab` are in the MATLAB path. Use `p = PetscOpenSocket();` (or `p = PetscOpenSocket(portnum)` if you provided a port number in your call to `PetscViewerSocketOpen()`), and then `a = PetscBinaryRead(p);` returns the object passed from PETSc. `PetscBinaryRead()` may be called any number of times. Each call should correspond on the PETSc side with viewing a single vector or matrix. `close()` closes the connection from MATLAB. On the PETSc side, one should destroy the viewer object with `PetscViewerDestroy()`.

For an example, which includes sending data back to PETSc, see Vec Tutorial ex42 and the associated `.m` file.

3.2.3 Using the MATLAB Compute Engine

One creates access to the MATLAB engine via

```
PetscMatlabEngineCreate(MPI_Comm comm,char *machine,PetscMatlabEngine *e);
```

where `machine` is the name of the machine hosting MATLAB (`NULL` may be used for localhost). One can send objects to MATLAB via

```
PetscMatlabEnginePut(PetscMatlabEngine e,PetscObject obj);
```

One can get objects via

```
PetscMatlabEngineGet(PetscMatlabEngine e,PetscObject obj);
```

Similarly, one can send arrays via

```
PetscMatlabEnginePutArray(PetscMatlabEngine e,int m,int n,PetscScalar *array,char␣
↪*name);
```

and get them back via

```
PetscMatlabEngineGetArray(PetscMatlabEngine e,int m,int n,PetscScalar *array,char␣
↪*name);
```

One cannot use MATLAB interactively in this mode but one can send MATLAB commands via

```
PetscMatlabEngineEvaluate(PetscMatlabEngine,"format",...);
```

where `format` has the usual `printf()` format. For example,

```
PetscMatlabEngineEvaluate(PetscMatlabEngine,"x = \%g *y + z;",avalue);
```

The name of each PETSc variable passed to MATLAB may be set with

```
PetscObjectSetName((PetscObject)A,"name");
```

Text responses can be returned from MATLAB via

```
PetscMatlabEngineGetOutput(PetscMatlabEngine,char **);
```

or

```
PetscMatlabEnginedPrintOutput(PetscMatlabEngine,FILE*).
```

There is a short-cut to starting the MATLAB engine with `PETSC_MATLAB_ENGINE_(MPI_Comm)`.

3.3 Profiling

PETSc includes a consistent, lightweight scheme to allow the profiling of application programs. The PETSc routines automatically log performance data if certain options are specified at runtime. The user can also log information about application codes for a complete picture of performance. In addition, as described in *Interpreting -log_view Output: The Basics*, PETSc provides a mechanism for printing informative messages about computations. *Basic Profiling Information* introduces the various profiling options in PETSc, while the remainder of the chapter focuses on details such as monitoring application codes and tips for accurate profiling.

3.3.1 Basic Profiling Information

If an application code and the PETSc libraries have been configured with `--with-log=1`, the default, then various kinds of profiling of code between calls to `PetscInitialize()` and `PetscFinalize()` can be activated at runtime. The profiling options include the following:

- `-log_view` - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, `-log_view` is intended as the primary means of monitoring the performance of PETSc codes.
- `-info [infofile]` - Prints verbose information about code to stdout or an optional file. This option provides details about algorithms, data structures, etc. Since the overhead of printing such output slows a code, this option should not be used when evaluating a program's performance.
- `-log_trace [logfile]` - Traces the beginning and ending of all PETSc events. This option, which can be used in conjunction with `-info`, is useful to see where a program is hanging without running in the debugger.

As discussed in *Using -log_mpe with Jumpshot*, additional profiling can be done with MPE.

Interpreting `-log_view` Output: The Basics

As shown in the listing in *Profiling Programs*, the option `-log_view` activates printing of profile data to standard output at the conclusion of a program. Profiling data can also be printed at any time within a program by calling `PetscLogView()`.

We print performance data for each routine, organized by PETSc libraries, followed by any user-defined events (discussed in *Profiling Application Codes*). For each routine, the output data include the maximum time and floating point operation (flop) rate over all processes. Information about parallel performance is also included, as discussed in the following section.

For the purpose of PETSc floating point operation counting, we define one *flop* as one operation of any of the following types: multiplication, division, addition, or subtraction. For example, one `VecAXPY()` operation, which computes $y = \alpha x + y$ for vectors of length N, requires $2N$ flop (consisting of N additions and N multiplications). Bear in mind that flop rates present only a limited view of performance, since memory loads and stores are the real performance barrier.

For simplicity, the remainder of this discussion focuses on interpreting profile data for the `KSP` library, which provides the linear solvers at the heart of the PETSc package. Recall the hierarchical organization of the PETSc library, as shown in *Numerical Libraries in PETSc*. Each `KSP` solver is composed of a `PC` (preconditioner) and a `KSP` (Krylov subspace) part, which are in turn built on top of the `Mat` (matrix) and `Vec` (vector) modules. Thus, operations in the `KSP` module are composed of lower-level operations in these packages. Note also that the nonlinear solvers library, `SNES`, is built on top of the `KSP` module, and the timestepping library, `TS`, is in turn built on top of `SNES`.

We briefly discuss interpretation of the sample output in the listing, which was generated by solving a linear system on one process using restarted GMRES and ILU preconditioning. The linear solvers in `KSP` consist of two basic phases, `KSPSetUp()` and `KSPSolve()`, each of which consists of a variety of actions, depending on the particular solution technique. For the case of using the `PCILU` preconditioner and `KSPGMRES` Krylov subspace method, the breakdown of PETSc routines is listed below. As indicated by the levels of indentation, the operations in `KSPSetUp()` include all of the operations within `PCSetUp()`, which in turn include `MatILUFactor()`, and so on.

- `KSPSetUp` - Set up linear solver
  - `PCSetUp` - Set up preconditioner
    - `MatILUFactor` - Factor preconditioning matrix
      - `MatILUFactorSymbolic` - Symbolic factorization phase
      - `MatLUFactorNumeric` - Numeric factorization phase
- `KSPSolve` - Solve linear system
  - `PCApply` - Apply preconditioner
    - `MatSolve` - Forward/backward triangular solves
  - `KSPGMRESOrthog` - Orthogonalization in GMRES
    - `VecDot` or `VecMDot` - Inner products
  - `MatMult` - Matrix-vector product
  - `MatMultAdd` - Matrix-vector product + vector addition
    - `VecScale`, `VecNorm`, `VecAXPY`, `VecCopy`, …

The summaries printed via `-log_view` reflect this routine hierarchy. For example, the performance summaries for a particular high-level routine such as `KSPSolve()` include all of the operations accumulated in the lower-level components that make up the routine.

Admittedly, we do not currently present the output with `-log_view` so that the hierarchy of PETSc operations is completely clear, primarily because we have not determined a clean and uniform way to do so throughout the library. Improvements may follow. However, for a particular problem, the user should generally have an idea of the basic operations that are required for its implementation (e.g., which operations are performed when using GMRES and ILU, as described above), so that interpreting the `-log_view` data should be relatively straightforward.

Interpreting `-log_view` Output: Parallel Performance

We next discuss performance summaries for parallel programs, as shown within the listings below , which present the combined output generated by the `-log_view` option. The program that generated this data is KSP Tutorial ex10. The code loads a matrix and right-hand-side vector from a binary file and then solves the resulting linear system; the program then repeats this process for a second linear system. This particular case was run on four processors of an Intel x86_64 Linux cluster, using restarted GMRES and the block Jacobi preconditioner, where each block was solved with ILU. The two input files `medium` and `arco6` can be downloaded from this FTP link.

The first listing presents an overall performance summary, including times, floating-point operations, computational rates, and message-passing activity (such as the number and size of messages sent and collective operations). Summaries for various user-defined stages of monitoring (as discussed in *Profiling Multiple Sections of Code*) are also given. Information about the various phases of computation then follow (as shown separately here in the second listing). Finally, a summary of memory usage and object creation and destruction is presented.

```
mpiexec -n 4 ./ex10 -f0 medium -f1 arco6 -ksp_gmres_classicalgramschmidt -log_view -
↪mat_type baij \
            -matload_block_size 3 -pc_type bjacobi -options_left

Number of iterations = 19
Residual norm 1.088292e-05
Number of iterations = 59
Residual norm 3.871022e-02
---------------------------------------------- PETSc Performance Summary: ------------
↪----------------------------------

./ex10 on a intel-bdw-opt named beboplogin4 with 4 processors, by jczhang Mon Apr 23␣
↪13:36:54 2018
Using Petsc Development GIT revision: v3.9-163-gbe3efd42  GIT Date: 2018-04-16␣
↪10:45:40 -0500

                         Max       Max/Min        Avg      Total
Time (sec):           1.849e-01      1.00002   1.849e-01
Objects:              1.060e+02      1.00000   1.060e+02
Flop:                 2.361e+08      1.00684   2.353e+08  9.413e+08
Flop/sec:            1.277e+09      1.00685   1.273e+09  5.091e+09
MPI Messages:         2.360e+02      1.34857   2.061e+02  8.245e+02
MPI Message Lengths:  1.256e+07      2.24620   4.071e+04  3.357e+07
MPI Reductions:       2.160e+02      1.00000

Summary of Stages:   ----- Time ------  ----- Flop -----  --- Messages ---  --␣
↪Message Lengths --  -- Reductions --
                        Avg     %Total     Avg     %Total    counts   %Total     Avg    ␣
↪      %Total   counts   %Total
 0:      Main Stage: 5.9897e-04   0.3%  0.0000e+00   0.0%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 1:   Load System 0: 2.9113e-03   1.6%  0.0000e+00   0.0%  3.550e+01   4.3%  5.
↪984e+02        0.1%  2.200e+01  10.2%
 2:      KSPSetUp 0: 7.7349e-04   0.4%  9.9360e+03   0.0%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 3:      KSPSolve 0: 1.7690e-03   1.0%  2.9673e+05   0.0%  1.520e+02  18.4%  1.
↪800e+02        0.1%  3.900e+01  18.1%
 4:   Load System 1: 1.0056e-01  54.4%  0.0000e+00   0.0%  3.700e+01   4.5%  5.
↪657e+05       62.4%  2.200e+01  10.2%
 5:      KSPSetUp 1: 5.6883e-03   3.1%  2.1205e+07   2.3%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 6:      KSPSolve 1: 7.2578e-02  39.3%  9.1979e+08  97.7%  6.000e+02  72.8%  2.
↪098e+04       37.5%  1.200e+02  55.6%

--------------------------------------------------------------------------------------
↪----------------------------------

.... [Summary of various phases, see part II below] ...

--------------------------------------------------------------------------------------
↪----------------------------------

Memory usage is given in bytes:

Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
...
```

(continues on next page)

(continued from previous page)

```
--- Event Stage 3: KSPSolve 0

              Matrix     0              4        23024     0.
              Vector    20             30        60048     0.
           Index Set     0              3         2568     0.
         Vec Scatter     0              1         1264     0.
       Krylov Solver     0              2        19592     0.
      Preconditioner     0              2         1912     0.
```

We next focus on the summaries for the various phases of the computation, as given in the table within the following listing. The summary for each phase presents the maximum times and flop rates over all processes, as well as the ratio of maximum to minimum times and flop rates for all processes. A ratio of approximately 1 indicates that computations within a given phase are well balanced among the processes; as the ratio increases, the balance becomes increasingly poor. Also, the total computational rate (in units of MFlop/sec) is given for each phase in the final column of the phase summary table.

$$\text{Total Mflop/sec} = 10^{-6} * (\text{sum of flop over all processors})/(\text{max time over all processors})$$

Note: Total computational rates < 1 MFlop are listed as 0 in this column of the phase summary table. Additional statistics for each phase include the total number of messages sent, the average message length, and the number of global reductions.

```
mpiexec -n 4 ./ex10 -f0 medium -f1 arco6 -ksp_gmres_classicalgramschmidt -log_view -
↪mat_type baij \
            -matload_block_size 3 -pc_type bjacobi -options_left

---------------------------------------------- PETSc Performance Summary: -----------
↪----------------------------------
.... [Overall summary, see part I] ...

Phase summary info:
   Count: number of times phase was executed
   Time and Flop/sec: Max - maximum over all processors
                       Ratio - ratio of maximum to minimum over all processors
   Mess: number of messages sent
   AvgLen: average message length
   Reduct: number of global reductions
   Global: entire computation
   Stage: optional user-defined stages of a computation. Set stages with␣
↪PetscLogStagePush() and PetscLogStagePop().
      %T - percent time in this phase         %F - percent flop in this phase
      %M - percent messages in this phase     %L - percent message lengths in this␣
↪phase
      %R - percent reductions in this phase
   Total Mflop/s: 10^6 * (sum of flop over all processors)/(max time over all␣
↪processors)
-------------------------------------------------------------------------------------
↪----------------------------------
Phase              Count      Time (sec)       Flop/sec                          ---␣
↪Global ---   --- Stage ----   Total
                            Max    Ratio      Max    Ratio  Mess AvgLen  Reduct  %T
↪%F %M %L %R  %T %F %M %L %R Mflop/s
-------------------------------------------------------------------------------------
↪----------------------------------
...
```

(continues on next page)

(continued from previous page)

```
--- Event Stage 5: KSPSetUp 1

MatLUFactorNum         1 1.0 3.6440e-03 1.1 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪2  0  0  0  62100  0  0  0  5819
MatILUFactorSym        1 1.0 1.7111e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  1 ␣
↪0  0  0  0  26  0  0  0  0     0
MatGetRowIJ            1 1.0 1.1921e-06 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
MatGetOrdering         1 1.0 3.0041e-05 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   1  0  0  0  0     0
KSPSetUp               2 1.0 6.6495e-04 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 2.0e+00  0 ␣
↪0  0  0  1   9  0  0  0100     0
PCSetUp                2 1.0 5.4271e-03 1.2 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪2  0  0  0  90100  0  0  0  3907
PCSetUpOnBlocks        1 1.0 5.3999e-03 1.2 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪2  0  0  0  90100  0  0  0  3927

--- Event Stage 6: KSPSolve 1

MatMult               60 1.0 2.4068e-02 1.1 6.54e+07 1.0 6.0e+02 2.1e+04 0.0e+00 12␣
↪27 73 37  0  32 28100100  0 10731
MatSolve              61 1.0 1.9177e-02 1.0 5.99e+07 1.0 0.0e+00 0.0e+00 0.0e+00 10␣
↪25  0  0  0  26 26  0  0  0 12491
VecMDot               59 1.0 1.4741e-02 1.3 4.86e+07 1.0 0.0e+00 0.0e+00 5.9e+01  7␣
↪21  0  0 27  18 21  0  0 49 13189
VecNorm               61 1.0 3.0417e-03 1.4 3.29e+06 1.0 0.0e+00 0.0e+00 6.1e+01  1 ␣
↪1  0  0 28   4  1  0  0 51  4332
VecScale              61 1.0 9.9802e-04 1.0 1.65e+06 1.0 0.0e+00 0.0e+00 0.0e+00  1 ␣
↪1  0  0  0   1  1  0  0  0  6602
VecCopy                2 1.0 5.9128e-05 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
VecSet                64 1.0 8.0323e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   1  0  0  0  0     0
VecAXPY                3 1.0 7.4387e-05 1.1 1.62e+05 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0  8712
VecMAXPY              61 1.0 8.8558e-03 1.1 5.18e+07 1.0 0.0e+00 0.0e+00 0.0e+00  5␣
↪22  0  0  0  12 23  0  0  0 23393
VecScatterBegin       60 1.0 9.6416e-04 1.8 0.00e+00 0.0 6.0e+02 2.1e+04 0.0e+00  0 ␣
↪0 73 37  0   1  0100100  0     0
VecScatterEnd         60 1.0 6.1543e-03 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪0  0  0  0   8  0  0  0  0     0
VecNormalize          61 1.0 4.2675e-03 1.3 4.94e+06 1.0 0.0e+00 0.0e+00 6.1e+01  2 ␣
↪2  0  0 28   5  2  0  0 51  4632
KSPGMRESOrthog        59 1.0 2.2627e-02 1.1 9.72e+07 1.0 0.0e+00 0.0e+00 5.9e+01 11␣
↪41  0  0 27  29 42  0  0 49 17185
KSPSolve               1 1.0 7.2577e-02 1.0 2.31e+08 1.0 6.0e+02 2.1e+04 1.2e+02 39␣
↪98 73 37 56  99100100100100 12673
PCSetUpOnBlocks        1 1.0 9.5367e-07 0.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
PCApply               61 1.0 2.0427e-02 1.0 5.99e+07 1.0 0.0e+00 0.0e+00 0.0e+00 11␣
↪25  0  0  0  28 26  0  0  0 11726
-------------------------------------------------------------------------------------
↪---------------------------------
.... [Conclusion of overall summary, see part I] ...
```

As discussed in the preceding section, the performance summaries for higher-level PETSc routines include the statistics for the lower levels of which they are made up. For example, the communication within matrix-vector products `MatMult()` consists of vector scatter operations, as given by the routines `VecScatterBegin()` and `VecScatterEnd()`.

The final data presented are the percentages of the various statistics (time (`%T`), flop/sec (`%F`), messages(`%M`), average message length (`%L`), and reductions (`%R`)) for each event relative to the total computation and to any user-defined stages (discussed in *Profiling Multiple Sections of Code*). These statistics can aid in optimizing performance, since they indicate the sections of code that could benefit from various kinds of tuning. *Hints for Performance Tuning* gives suggestions about achieving good performance with PETSc codes.

Using `-log_mpe` with Jumpshot

It is also possible to use the *Jumpshot* package [HL91] to visualize PETSc events. This package comes with the MPE software, which is part of the MPICH [Getal] implementation of MPI. The option

```
-log_mpe [logfile]
```

creates a logfile of events appropriate for viewing with *Jumpshot*. The user can either use the default logging file or specify a name via `logfile`. Events can be deactivated as described in *Restricting Event Logging*.

The user can also log MPI events. To do this, simply consider the PETSc application as any MPI application, and follow the MPI implementation's instructions for logging MPI calls. For example, when using MPICH, this merely required adding `-llmpich` to the library list *before* `-lmpich`.

3.3.2 Profiling Application Codes

PETSc automatically logs object creation, times, and floating-point counts for the library routines. Users can easily supplement this information by monitoring their application codes as well. The basic steps involved in logging a user-defined portion of code, called an *event*, are shown in the code fragment below:

```
PetscLogEvent  USER_EVENT;
PetscClassId   classid;
PetscLogDouble user_event_flops;

PetscClassIdRegister("class name",&classid);
PetscLogEventRegister("User event name",classid,&USER_EVENT);
PetscLogEventBegin(USER_EVENT,0,0,0,0);
/* code segment to monitor */
PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

One must register the event by calling `PetscLogEventRegister()`, which assigns a unique integer to identify the event for profiling purposes:

```
PetscLogEventRegister(const char string[],PetscClassId classid,PetscLogEvent *e);
```

Here `string` is a user-defined event name, and `color` is an optional user-defined event color (for use with *Jumpshot* logging; see *Using -log_mpe with Jumpshot*); one should see the manual page for details. The argument returned in `e` should then be passed to the `PetscLogEventBegin()` and `PetscLogEventEnd()` routines.

Events are logged by using the pair

```
PetscLogEventBegin(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject␣
↪o4);
PetscLogEventEnd(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject␣
↪o4);
```

The four objects are the PETSc objects that are most closely associated with the event. For instance, in a matrix-vector product they would be the matrix and the two vectors. These objects can be omitted by specifying 0 for `o1` - `o4`. The code between these two routine calls will be automatically timed and logged as part of the specified event.

The user can log the number of floating-point operations for this segment of code by calling

```
PetscLogFlops(number of flop for this code segment);
```

between the calls to `PetscLogEventBegin()` and `PetscLogEventEnd()`. This value will automatically be added to the global flop counter for the entire program.

3.3.3 Profiling Multiple Sections of Code

By default, the profiling produces a single set of statistics for all code between the `PetscInitialize()` and `PetscFinalize()` calls within a program. One can independently monitor up to ten stages of code by switching among the various stages with the commands

```
PetscLogStagePush(PetscLogStage stage);
PetscLogStagePop();
```

where `stage` is an integer (0-9); see the manual pages for details. The command

```
PetscLogStageRegister(const char *name,PetscLogStage *stage)
```

allows one to associate a name with a stage; these names are printed whenever summaries are generated with `-log_view` or `PetscLogView()`. The following code fragment uses three profiling stages within an program.

```
PetscInitialize(int *argc,char ***args,0,0);
/* stage 0 of code here */
PetscLogStageRegister("Stage 0 of Code", &stagenum0);
for (i=0; i<ntimes; i++) {
    PetscLogStageRegister("Stage 1 of Code", &stagenum1);
    PetscLogStagePush(stagenum1);
    /* stage 1 of code here */
    PetscLogStagePop();
    PetscLogStageRegister("Stage 2 of Code", &stagenum2);
    PetscLogStagePush(stagenum2);
    /* stage 2 of code here */
    PetscLogStagePop();
}
PetscFinalize();
```

The listings above Figures *[fig_exparprof]* and show output generated by `-log_view` for a program that employs several profiling stages. In particular, this program is subdivided into six stages except the Main stage: loading a matrix and right-hand-side vector from a binary file, setting up the preconditioner, and solving the linear system; this sequence is then repeated for a second linear system. For simplicity, the second listing contains output only for stages 5 and 6 (linear solve of the second system), which comprise the part of this computation of most interest to us in terms of performance monitoring. This code organization (solving a small linear system followed by a larger system) enables generation of more accurate profiling statistics for the second system by overcoming the often considerable overhead of paging, as discussed in *Accurate Profiling and Paging Overheads*.

3.3.4 Restricting Event Logging

By default, all PETSc operations are logged. To enable or disable the PETSc logging of individual events, one uses the commands

```
PetscLogEventActivate(int event);
PetscLogEventDeactivate(int event);
```

The `event` may be either a predefined PETSc event (as listed in the file `${PETSC_DIR}/include/petsclog.h`) or one obtained with `PetscLogEventRegister()` (as described in *Profiling Application Codes*).

PETSc also provides routines that deactivate (or activate) logging for entire components of the library. Currently, the components that support such logging (de)activation are `Mat` (matrices), `Vec` (vectors), `KSP` (linear solvers, including `KSP` and `PC`), and `SNES` (nonlinear solvers):

```
PetscLogEventDeactivateClass(MAT_CLASSID);
PetscLogEventDeactivateClass(KSP_CLASSID); /* includes PC and KSP */
PetscLogEventDeactivateClass(VEC_CLASSID);
PetscLogEventDeactivateClass(SNES_CLASSID);
```

and

```
PetscLogEventActivateClass(MAT_CLASSID);
PetscLogEventActivateClass(KSP_CLASSID);   /* includes PC and KSP */
PetscLogEventActivateClass(VEC_CLASSID);
PetscLogEventActivateClass(SNES_CLASSID);
```

Recall that the option `-log_all` produces extensive profile data, which can be a challenge for PETScView to handle due to the memory limitations of Tcl/Tk. Thus, one should generally use `-log_all` when running programs with a relatively small number of events or when disabling some of the events that occur many times in a code (e.g., `VecSetValues()`, `MatSetValues()`).

3.3.5 Interpreting `-log_info` Output: Informative Messages

Users can activate the printing of verbose information about algorithms, data structures, etc. to the screen by using the option `-info` or by calling `PetscInfoAllow(PETSC_TRUE)`. Such logging, which is used throughout the PETSc libraries, can aid the user in understanding algorithms and tuning program performance. For example, as discussed in *Sparse Matrices*, `-info` activates the printing of information about memory allocation during matrix assembly.

Application programmers can employ this logging as well, by using the routine

```
PetscInfo(void* obj,char *message,...)
```

where `obj` is the PETSc object associated most closely with the logging statement, `message`. For example, in the line search Newton methods, we use a statement such as

```
PetscInfo(snes,"Cubically determined step, lambda %g\n",lambda);
```

One can selectively turn off informative messages about any of the basic PETSc objects (e.g., `Mat`, `SNES`) with the command

```
PetscInfoDeactivateClass(int object_classid)
```

where `object_classid` is one of `MAT_CLASSID`, `SNES_CLASSID`, etc. Messages can be reactivated with the command

```
PetscInfoActivateClass(int object_classid)
```

Such deactivation can be useful when one wishes to view information about higher-level PETSc libraries (e.g., `TS` and `SNES`) without seeing all lower level data as well (e.g., `Mat`). One can deactivate events at runtime for matrix and linear solver libraries via `-info [no_mat, no_ksp]`.

3.3.6 Time

PETSc application programmers can access the wall clock time directly with the command

```
PetscLogDouble time;
PetscTime(&time);CHKERRQ(ierr);
```

which returns the current time in seconds since the epoch, and is commonly implemented with `MPI_Wtime`. A floating point number is returned in order to express fractions of a second. In addition, as discussed in *Profiling Application Codes*, PETSc can automatically profile user-defined segments of code.

3.3.7 Saving Output to a File

All output from PETSc programs (including informative messages, profiling information, and convergence data) can be saved to a file by using the command line option `-history [filename]`. If no file name is specified, the output is stored in the file `${HOME}/.petschistory`. Note that this option only saves output printed with the `PetscPrintf()` and `PetscFPrintf()` commands, not the standard `printf()` and `fprintf()` statements.

3.3.8 Accurate Profiling and Paging Overheads

One factor that often plays a significant role in profiling a code is paging by the operating system. Generally, when running a program, only a few pages required to start it are loaded into memory rather than the entire executable. When the execution proceeds to code segments that are not in memory, a pagefault occurs, prompting the required pages to be loaded from the disk (a very slow process). This activity distorts the results significantly. (The paging effects are noticeable in the log files generated by `-log_mpe`, which is described in *Using -log_mpe with Jumpshot*.)

To eliminate the effects of paging when profiling the performance of a program, we have found an effective procedure is to run the *exact same code* on a small dummy problem before running it on the actual problem of interest. We thus ensure that all code required by a solver is loaded into memory during solution of the small problem. When the code proceeds to the actual (larger) problem of interest, all required pages have already been loaded into main memory, so that the performance numbers are not distorted.

When this procedure is used in conjunction with the user-defined stages of profiling described in *Profiling Multiple Sections of Code*, we can focus easily on the problem of interest. For example, we used this technique in the program KSP Tutorial ex10 to generate the timings within the listings above In this case, the profiled code of interest (solving the linear system for the larger problem) occurs within event stages 5 and 6. *Interpreting -log_view Output: Parallel Performance* provides details about interpreting such profiling data.

In particular, the macros

```
PetscPreLoadBegin(PetscBool flag,char* stagename)
PetscPreLoadStage(char *stagename)
```

and

```
PetscPreLoadEnd()
```

can be used to easily convert a regular PETSc program to one that uses preloading. The command line options `-preload true` and `-preload false` may be used to turn on and off preloading at run time for PETSc programs that use these macros.

3.4 Hints for Performance Tuning

This chapter provides hints on how to get to achieve best performance with PETSc, particularly on distributed-memory machines with multiple CPU sockets per node. We focus on machine-related performance optimization here; algorithmic aspects like preconditioner selection are not the focus of this section.

3.4.1 Maximizing Memory Bandwidth

Most operations in PETSc deal with large datasets (typically vectors and sparse matrices) and perform relatively few arithmetic operations for each byte loaded or stored from global memory. Therefore, the *arithmetic intensity* expressed as the ratio of floating point operations to the number of bytes loaded and stored is usually well below unity for typical PETSc operations. On the other hand, modern CPUs are able to execute on the order of 10 floating point operations for each byte loaded or stored. As a consequence, almost all PETSc operations are limited by the rate at which data can be loaded or stored (*memory bandwidth limited*) rather than by the rate of floating point operations.

This section discusses ways to maximize the memory bandwidth achieved by applications based on PETSc. Where appropriate, we include benchmark results in order to provide quantitative results on typical performance gains one can achieve through parallelization, both on a single compute node and across nodes. In particular, we start with the answer to the common question of why performance generally does not increase 20-fold with a 20-core CPU.

Memory Bandwidth vs. Processes

Consider the addition of two large vectors, with the result written to a third vector. Because there are no dependencies across the different entries of each vector, the operation is embarrasingly parallel.

As :numref‘fig_stream_intel‘ shows, the performance gains due to parallelization on different multi- and many-core CPUs quickly saturates. The reason is that only a fraction of the total number of CPU cores is required to saturate the memory channels. For example, a dual-socket system equipped with Haswell 12-core Xeon CPUs achieves more than 80 percent of achievable peak memory bandwidth with only four processes per socket (8 total), cf. Fig. 3.1. Consequently, running with more than 8 MPI ranks on such a system will not increase performance substantially. For the same reason, PETSc-based applications usually do not benefit from hyper-threading.

PETSc provides a simple way to measure memory bandwidth for different numbers of processes via the target `make streams` executed from `$PETSC_DIR`. The output provides an overview of the possible speedup one can obtain on the given machine (not necessarily a shared memory system). For example, the following is the most relevant output obtained on a dual-socket system equipped with two six-core-CPUs with hyperthreading:

```
np  speedup
1 1.0
2 1.58
3 2.19
```

(continues on next page)

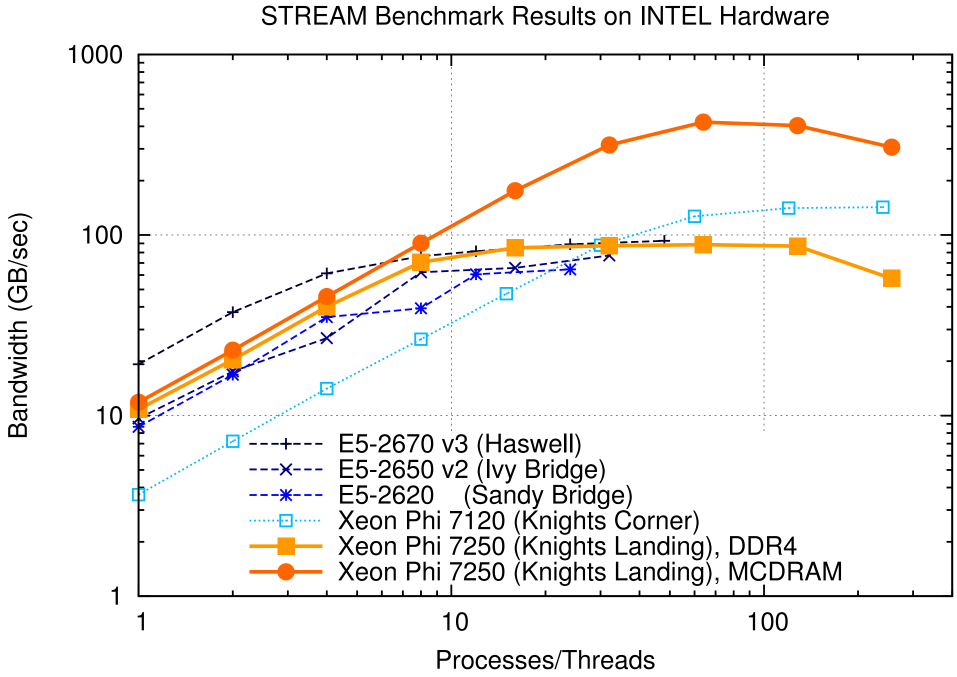


Fig. 3.1: Memory bandwidth obtained on Intel hardware (dual socket except KNL) over the number of processes used. One can get close to peak memory bandwidth with only a few processes.

(continued from previous page)

```
4 2.42
5 2.63
6 2.69
...
21 3.82
22 3.49
23 3.79
24 3.71
Estimation of possible speedup of MPI programs based on Streams benchmark.
It appears you have 1 node(s)
```

On this machine, one should expect a speed-up of typical memory bandwidth-bound PETSc applications of at most 4x when running multiple MPI ranks on the node. Most of the gains are already obtained when running with only 4-6 ranks. Because a smaller number of MPI ranks usually implies better preconditioners and better performance for smaller problems, the best performance for PETSc applications may be obtained with fewer ranks than there are physical CPU cores available.

Following the results from the above run of `make streams`, we recommend to use additional nodes instead of placing additional MPI ranks on the nodes. In particular, weak scaling (i.e. constant load per process, increasing the number of processes) and strong scaling (i.e. constant total work, increasing the number of processes) studies should keep the number of processes per node constant.

Non-Uniform Memory Access (NUMA) and Process Placement

CPUs in nodes with more than one CPU socket are internally connected via a high-speed fabric, cf. Fig. 3.2, to enable data exchange as well as cache coherency. Because main memory on modern systems is connected via the integrated memory controllers on each CPU, memory is accessed in a non-uniform way: A process running on one socket has direct access to the memory channels of the respective CPU, whereas requests for memory attached to a different CPU socket need to go through the high-speed fabric. Consequently, best aggregate memory bandwidth on the node is obtained when the memory controllers on each CPU are fully saturated. However, full saturation of memory channels is only possible if the data is distributed across the different memory channels.

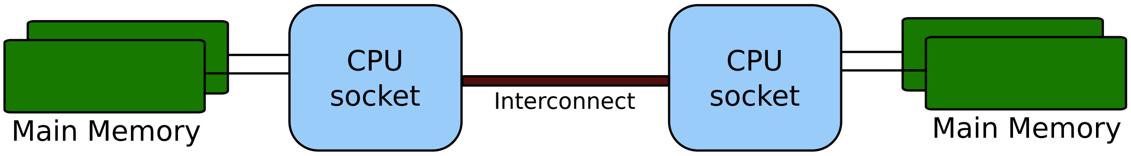


Fig. 3.2: Schematic of a two-socket NUMA system. Processes should be spread across both CPUs to obtain full bandwidth.

Data in memory on modern machines is allocated by the operating system based on a first-touch policy. That is, memory is not allocated at the point of issuing `malloc()`, but at the point when the respective memory segment is actually touched (read or write). Upon first-touch, memory is allocated on the memory channel associated with the respective CPU the process is running on. Only if all memory on the respective CPU is already in use (either allocated or as IO cache), memory available through other sockets is considered.

Maximum memory bandwidth can be achieved by ensuring that processes are spread over all sockets in the respective node. For example, the recommended placement of a 8-way parallel run on a four-socket machine is to assign two processes to each CPU socket. To do so, one needs to know the enumeration of cores and pass the requested information to `mpirun`. Consider the hardware topology information returned by `lstopo` (part of the hwloc package) for the following two-socket machine, in which each CPU consists of six cores and supports hyperthreading:

```
Machine (126GB total)
  NUMANode L#0 (P#0 63GB)
    Package L#0 + L3 L#0 (15MB)
      L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#12)
      L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#13)
      L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
        PU L#4 (P#2)
        PU L#5 (P#14)
      L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
        PU L#6 (P#3)
        PU L#7 (P#15)
      L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
        PU L#8 (P#4)
        PU L#9 (P#16)
      L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
        PU L#10 (P#5)
        PU L#11 (P#17)
  NUMANode L#1 (P#1 63GB)
    Package L#1 + L3 L#1 (15MB)
```

(continues on next page)

(continued from previous page)

```
      L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6
        PU L#12 (P#6)
        PU L#13 (P#18)
      L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7
        PU L#14 (P#7)
        PU L#15 (P#19)
      L2 L#8 (256KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8
        PU L#16 (P#8)
        PU L#17 (P#20)
      L2 L#9 (256KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9
        PU L#18 (P#9)
        PU L#19 (P#21)
      L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10
        PU L#20 (P#10)
        PU L#21 (P#22)
      L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11
        PU L#22 (P#11)
        PU L#23 (P#23)
```

The relevant physical processor IDs are shown in parentheses prefixed by `P#`. Here, IDs 0 and 12 share the same physical core and have a common L2 cache. IDs 0, 12, 1, 13, 2, 14, 3, 15, 4, 16, 5, 17 share the same socket and have a common L3 cache.

A good placement for a run with six processes is to locate three processes on the first socket and three processes on the second socket. Unfortunately, mechanisms for process placement vary across MPI implementations, so make sure to consult the manual of your MPI implementation. The following discussion is based on how processor placement is done with MPICH and OpenMPI, where one needs to pass `--bind-to core --map-by socket` to `mpirun`:

```
$> mpirun -n 6 --bind-to core --map-by socket ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 000000100000000000100000
process 2 binding: 010000000000010000000000
process 3 binding: 000000010000000000010000
process 4 binding: 001000000000001000000000
process 5 binding: 000000001000000000001000
Triad:        45403.1949   Rate (MB/s)
```

In this configuration, process 0 is bound to the first physical core on the first socket (with IDs 0 and 12), process 1 is bound to the first core on the second socket (IDs 6 and 18), and similarly for the remaining processes. The achieved bandwidth of 45 GB/sec is close to the practical peak of about 50 GB/sec available on the machine. If, however, all MPI processes are located on the same socket, memory bandwidth drops significantly:

```
$> mpirun -n 6 --bind-to core --map-by core ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 010000000000010000000000
process 2 binding: 001000000000001000000000
process 3 binding: 000100000000000100000000
process 4 binding: 000010000000000010000000
process 5 binding: 000001000000000001000000
Triad:        25510.7507   Rate (MB/s)
```

All processes are now mapped to cores on the same socket. As a result, only the first memory channel is fully saturated at 25.5 GB/sec.

One must not assume that `mpirun` uses good defaults. To demonstrate, compare the full output of `make streams` from *Memory Bandwidth vs. Processes* on the left with the results on the right obtained by passing `--bind-to core --map-by socket`:

```
$> make streams
np  speedup
1 1.0
2 1.58
3 2.19
4 2.42
5 2.63
6 2.69
7 2.31
8 2.42
9 2.37
10 2.65
11 2.3
12 2.53
13 2.43
14 2.63
15 2.74
16 2.7
17 3.28
18 3.66
19 3.95
20 3.07
21 3.82
22 3.49
23 3.79
24 3.71
```

```
$> make streams MPI_BINDING="--bind-to core --map-by socket"
np  speedup
1 1.0
2 1.59
3 2.66
4 3.5
5 3.56
6 4.23
7 3.95
8 4.39
9 4.09
10 4.46
11 4.15
12 4.42
13 3.71
14 3.83
15 4.08
16 4.22
17 4.18
18 4.31
19 4.22
20 4.28
21 4.25
22 4.23
23 4.28
```

(continues on next page)

(continued from previous page)

```
24 4.22
```

For the non-optimized version on the left, the speedup obtained when using any number of processes between 3 and 13 is essentially constant up to fluctuations, indicating that all processes were by default executed on the same socket. Only with 14 or more processes, the speedup number increases again. In contrast, the results of `make streams` with proper processor placement shown on the right resulted in slightly higher overall parallel speedup (identical baselines), in smaller performance fluctuations, and more than 90 percent of peak bandwidth with only six processes.

Machines with job submission systems such as SLURM usually provide similar mechanisms for processor placements through options specified in job submission scripts. Please consult the respective manuals.

Additional Process Placement Considerations and Details

For a typical, memory bandwidth-limited PETSc application, the primary consideration in placing MPI processes is ensuring that processes are evenly distributed among sockets, and hence using all available memory channels. Increasingly complex processor designs and cache hierarchies, however, mean that performance may also be sensitive to how processes are bound to the resources within each socket. Performance on the two processor machine in the preceding example may be relatively insensitive to such placement decisions, because one L3 cache is shared by all cores within a NUMA domain, and each core has its own L2 and L1 caches. However, processors that are less "flat", with more complex hierarchies, may be more sensitive. In many AMD Opterons or the second-generation "Knights Landing" Intel Xeon Phi, for instance, L2 caches are shared between two cores. On these processors, placing consecutive MPI ranks on cores that share the same L2 cache may benefit performance if the two ranks communicate frequently with each other, because the latency between cores sharing an L2 cache may be roughly half that of two cores not sharing one. There may be benefit, however, in placing consecutive ranks on cores that do not share an L2 cache, because (if there are fewer MPI ranks than cores) this increases the total L2 cache capacity and bandwidth available to the application. There is a trade-off to be considered between placing processes close together (in terms of shared resources) to optimize for efficient communication and synchronization vs. farther apart to maximize available resources (memory channels, caches, I/O channels, etc.), and the best strategy will depend on the application and the software and hardware stack.

Different process placement strategies can affect performance at least as much as some commonly explored settings, such as compiler optimization levels. Unfortunately, exploration of this space is complicated by two factors: First, processor and core numberings may be completely arbitrary, changing with BIOS version, etc., and second—as already noted—there is no standard mechanism used by MPI implementations (or job schedulers) to specify process affinity. To overcome the first issue, we recommend using the `lstopo` utility of the Portable Hardware Locality (`hwloc`) software package (which can be installed by configuring PETSc with `–download-hwloc`) to understand the processor topology of your machine. We cannot fully address the second issue—consult the documenation for your MPI implementation and/or job scheduler—but we offer some general observations on understanding placement options:

- An MPI implementation may support a notion of *domains* in which a process may be pinned. A domain may simply correspond to a single core; however, the MPI implementation may allow a deal of flexibility in specifying domains that encompass multiple cores, span sockets, etc. Some implementations, such as Intel MPI, provide means to specify whether domains should be "compact"—composed of cores sharing resources such as caches—or "scatter"-ed, with little resource sharing (possibly even spanning sockets).

- Separate from the specification of domains, MPI implementations often support different *orderings* in which MPI ranks should be bound to these domains. Intel MPI, for instance, supports "compact" ordering to place consecutive ranks close in terms of shared resources, "scatter" to place them far apart, and "bunch" to map proportionally to sockets while placing ranks as close together as possible within the sockets.
- An MPI implemenation that supports process pinning should offer some way to view the rank assignments. Use this output in conjunction with the topology obtained via `lstopo` or a similar tool to determine if the placements correspond to something you believe is reasonable for your application. Do not assume that the MPI implementation is doing something sensible by default!

3.4.2 Performance Pitfalls and Advice

This section looks into a potpourri of performance pitfalls encountered by users in the past. Many of these pitfalls require a deeper understanding of the system and experience to detect. The purpose of this section is to summarize and share our experience so that these pitfalls can be avoided in the future.

Debug vs. Optimized Builds

PETSc's `./configure` defaults to building PETSc with debug mode enabled. Any code development should be done in this mode, because it provides handy debugging facilities such as accurate stack traces, memory leak checks, or memory corruption checks. Note that PETSc has no reliable way of knowing whether a particular run is a production or debug run. In the case that a user requests profiling information via `-log_view`, a debug build of PETSc issues the following warning:

```
##########################################################
#                                                        #
#                          WARNING!!!                    #
#                                                        #
#   This code was compiled with a debugging option,      #
#   To get timing results run ./configure                #
#   using --with-debugging=no, the performance will      #
#   be generally two or three times faster.              #
#                                                        #
##########################################################
```

Conversely, one way of checking whether a particular build of PETSc has debugging enabled is to inspect the output of `-log_view`.

Debug mode will generally be most useful for code development if appropriate compiler options are set to faciliate debugging. The compiler should be instructed to generate binaries with debug symbols (command line option `-g` for most compilers), and the optimization level chosen should either completely disable optimizations (`-O0` for most compilers) or enable only optimizations that do not interfere with debugging (GCC, for instance, supports a `-Og` optimization level that does this).

Only once the new code is thoroughly tested and ready for production, one should disable debugging facilities by passing `--with-debugging=no` to

`./configure`. One should also ensure that an appropriate compiler optimization level is set. Note that some compilers (e.g., Intel) default to fairly comprehensive optimization levels, while others (e.g., GCC) default to no optimization at all. The best optimization flags will depend on your code, the compiler, and the target architecture, but we offer a few guidelines for finding those that will offer the best performance:

- Most compilers have a number of optimization levels (with level n usually specified via `-On`) that provide a quick way to enable sets of several optimization flags. We suggest trying the higher optimization levels (the highest level is not guaranteed to produce the fastest executable, so some experimentation may be merited). With most recent processors now supporting some form of SIMD or vector instructions, it is important to choose a level that enables the compiler's auto-vectorizer; many compilers do not enable auto-vectorization at lower optimization levels (e.g., GCC does not enable it below `-O3` and the Intel compiler does not enable it below `-O2`).

- For processors supporting newer vector instruction sets, such as Intel AVX2 and AVX-512, it is also important to direct the compiler to generate code that targets these processors (.e.g., `-march=knl` to generate AVX-512 binaries targeting the Intel "Knights Landing" Xeon Phi processor); otherwise, the executables built will not utilize the newer instructions sets and will not take advantage of the vector processing units.
- Beyond choosing the optimization levels, some value-unsafe optimizations (such as using reciprocals of values instead of dividing by those values, or allowing re-association of operands in a series of calculations) for floating point calculations may yield significant performance gains. Compilers often provide flags (e.g., `-ffast-math` in GCC) to enable a set of these optimizations, and they may be turned on when using options for very aggressive optimization (`-fast` or `-Ofast` in many compilers). These are worth exploring to maximize performance, but, if employed, it important to verify that these do not cause erroneous results with your code, since calculations may violate the IEEE standard for floating-point arithmetic.

Profiling

Users should not spend time optimizing a code until after having determined where it spends the bulk of its time on realistically sized problems. As discussed in detail in *Profiling*, the PETSc routines automatically log performance data if certain runtime options are specified.

To obtain a summary of where and how much time is spent in different sections of the code, use one of the following options:

- Run the code with the option `-log_view` to print a performance summary for various phases of the code.
- Run the code with the option `-log_mpe [logfilename]`, which creates a logfile of events suitable for viewing with Jumpshot (part of MPICH).

Then, focus on the sections where most of the time is spent. If you provided your own callback routines, e.g. for residual evaluations, search the profiling output for routines such as `SNESFunctionEval` or `SNESJacobianEval`. If their relative time is significant (say, more than 30 percent), consider optimizing these routines first. Generic instructions on how to optimize your callback functions are difficult; you may start by reading performance optimization guides for your system's hardware.

Aggregation

Performing operations on chunks of data rather than a single element at a time can significantly enhance performance because of cache reuse or lower data motion. Typical examples are:

- Insert several (many) elements of a matrix or vector at once, rather than looping and inserting a single value at a time. In order to access elements in of vector repeatedly, employ `VecGetArray()` to allow direct manipulation of the vector elements.
- When possible, use `VecMDot()` rather than a series of calls to `VecDot()`.
- If you require a sequence of matrix-vector products with the same matrix, consider packing your vectors into a single matrix and use matrix-matrix multiplications.
- Users should employ a reasonable number of `PetscMalloc()` calls in their codes. Hundreds or thousands of memory allocations may be appropriate; however, if tens of thousands are being used, then reducing the number of `PetscMalloc()` calls may be warranted. For example, reusing space

or allocating large chunks and dividing it into pieces can produce a significant savings in allocation overhead. *Data Structure Reuse* gives details.

Aggressive aggregation of data may result in inflexible datastructures and code that is hard to maintain. We advise users to keep these competing goals in mind and not blindly optimize for performance only.

Memory Allocation for Sparse Matrix Assembly

Since the process of dynamic memory allocation for sparse matrices is inherently very expensive, accurate preallocation of memory is crucial for efficient sparse matrix assembly. One should use the matrix creation routines for particular data structures, such as `MatCreateSeqAIJ()` and `MatCreateAIJ()` for compressed, sparse row formats, instead of the generic `MatCreate()` routine. For problems with multiple degrees of freedom per node, the block, compressed, sparse row formats, created by `MatCreateSeqBAIJ()` and `MatCreateBAIJ()`, can significantly enhance performance. *Sparse Matrices* includes extensive details and examples regarding preallocation.

Memory Allocation for Sparse Matrix Factorization

When symbolically factoring an AIJ matrix, PETSc has to guess how much fill there will be. Careful use of the fill parameter in the `MatILUInfo` structure when calling `MatLUFactorSymbolic()` or `MatILUFactorSymbolic()` can reduce greatly the number of mallocs and copies required, and thus greatly improve the performance of the factorization. One way to determine a good value for the fill parameter is to run a program with the option `-info`. The symbolic factorization phase will then print information such as

```
Info:MatILUFactorSymbolic_AIJ:Realloc 12 Fill ratio:given 1 needed 2.16423
```

This indicates that the user should have used a fill estimate factor of about 2.17 (instead of 1) to prevent the 12 required mallocs and copies. The command line option

```
-pc_ilu_fill 2.17
```

will cause PETSc to preallocate the correct amount of space for incomplete (ILU) factorization. The corresponding option for direct (LU) factorization is `-pc_factor_fill <fill_amount>`.

Detecting Memory Allocation Problems

PETSc provides a number of tools to aid in detection of problems with memory allocation, including leaks and use of uninitialized space. We briefly describe these below.

- The PETSc memory allocation (which collects statistics and performs error checking), is employed by default for codes compiled in a debug-mode (configured with `--with-debugging=1`). PETSc memory allocation can be activated for optimized-mode (configured with `--with-debugging=0`) using the option `-malloc`. The option `-malloc=0` forces the use of conventional memory allocation when debugging is enabled. When running timing tests, one should build libraries in optimized mode.
- When the PETSc memory allocation routines are used, the option `-malloc_dump` will print a list of unfreed memory at the conclusion of a program. If all memory has been freed, only a message stating the maximum allocated space will be printed. However, if some memory remains unfreed, this information will be printed. Note that the option `-malloc_dump` merely activates a call to `PetscMallocDump()` during `PetscFinalize()` the user can also call `PetscMallocDump()` elsewhere in a program.
- Another useful option for use with PETSc memory allocation routines is `-malloc_view`, which activates logging of all calls to malloc and reports memory usage, including all Fortran arrays. This option

provides a more complete picture than `-malloc_dump` for codes that employ Fortran with hardwired arrays. The option `-malloc_view` activates logging by calling `PetscMallocViewSet()` in `PetscInitialize()` and then prints the log by calling `PetscMallocView()` in `PetscFinalize()`. The user can also call these routines elsewhere in a program. When finer granularity is desired, the user should call `PetscMallocGetCurrentUsage()` and `PetscMallocGetMaximumUsage()` for memory allocated by PETSc, or `PetscMemoryGetCurrentUsage()` and `PetscMemoryGetMaximumUsage()` for the total memory used by the program. Note that `PetscMemorySetGetMaximumUsage()` must be called before `PetscMemoryGetMaximumUsage()` (typically at the beginning of the program).

Data Structure Reuse

Data structures should be reused whenever possible. For example, if a code often creates new matrices or vectors, there often may be a way to reuse some of them. Very significant performance improvements can be achieved by reusing matrix data structures with the same nonzero pattern. If a code creates thousands of matrix or vector objects, performance will be degraded. For example, when solving a nonlinear problem or timestepping, reusing the matrices and their nonzero structure for many steps when appropriate can make the code run significantly faster.

A simple technique for saving work vectors, matrices, etc. is employing a user-defined context. In C and C++ such a context is merely a structure in which various objects can be stashed; in Fortran a user context can be an integer array that contains both parameters and pointers to PETSc objects. See SNES Tutorial ex5 and SNES Tutorial ex5f for examples of user-defined application contexts in C and Fortran, respectively.

Numerical Experiments

PETSc users should run a variety of tests. For example, there are a large number of options for the linear and nonlinear equation solvers in PETSc, and different choices can make a *very* big difference in convergence rates and execution times. PETSc employs defaults that are generally reasonable for a wide range of problems, but clearly these defaults cannot be best for all cases. Users should experiment with many combinations to determine what is best for a given problem and customize the solvers accordingly.

- Use the options `-snes_view`, `-ksp_view`, etc. (or the routines `KSPView()`, `SNESView()`, etc.) to view the options that have been used for a particular solver.
- Run the code with the option `-help` for a list of the available runtime commands.
- Use the option `-info` to print details about the solvers' operation.
- Use the PETSc monitoring discussed in *Profiling* to evaluate the performance of various numerical methods.

Tips for Efficient Use of Linear Solvers

As discussed in *KSP: Linear System Solvers*, the default linear solvers are

- uniprocess: GMRES(30) with ILU(0) preconditioning
- multiprocess: GMRES(30) with block Jacobi preconditioning, where there is 1 block per process, and each block is solved with ILU(0)

One should experiment to determine alternatives that may be better for various applications. Recall that one can specify the `KSP` methods and preconditioners at runtime via the options:

```
-ksp_type <ksp_name> -pc_type <pc_name>
```

One can also specify a variety of runtime customizations for the solvers, as discussed throughout the manual.

In particular, note that the default restart parameter for GMRES is 30, which may be too small for some large-scale problems. One can alter this parameter with the option `-ksp_gmres_restar <restart>` or by calling `KSPGMRESSetRestart()`. *Krylov Methods* gives information on setting alternative GMRES orthogonalization routines, which may provide much better parallel performance.

For elliptic problems one often obtains good performance and scalability with multigrid solvers. Consult *Algebraic Multigrid (AMG) Preconditioners* for available options. Our experience is that GAMG works particularly well for elasticity problems, whereas hypre does well for scalar problems.

System-Related Problems

The performance of a code can be affected by a variety of factors, including the cache behavior, other users on the machine, etc. Below we briefly describe some common problems and possibilities for overcoming them.

- **Problem too large for physical memory size**: When timing a program, one should always leave at least a ten percent margin between the total memory a process is using and the physical size of the machine's memory. One way to estimate the amount of memory used by given process is with the UNIX `getrusage` system routine. Also, the PETSc option `-log_view` prints the amount of memory used by the basic PETSc objects, thus providing a lower bound on the memory used. Another useful option is `-malloc_view` which reports all memory, including any Fortran arrays in an application code.
- **Effects of other users**: If other users are running jobs on the same physical processor nodes on which a program is being profiled, the timing results are essentially meaningless.
- **Overhead of timing routines on certain machines**: On certain machines, even calling the system clock in order to time routines is slow; this skews all of the flop rates and timing results. The file `$PETSC_DIR/src/benchmarks/PetscTime.c` (source) contains a simple test problem that will approximate the amount of time required to get the current time in a running program. On good systems it will on the order of 10^{-6} seconds or less.
- **Problem too large for good cache performance**: Certain machines with lower memory bandwidths (slow memory access) attempt to compensate by having a very large cache. Thus, if a significant portion of an application fits within the cache, the program will achieve very good performance; if the code is too large, the performance can degrade markedly. To analyze whether this situation affects a particular code, one can try plotting the total flop rate as a function of problem size. If the flop rate decreases rapidly at some point, then the problem may likely be too large for the cache size.
- **Inconsistent timings**: Inconsistent timings are likely due to other users on the machine, thrashing (using more virtual memory than available physical memory), or paging in of the initial executable. *Accurate Profiling and Paging Overheads* provides information on overcoming paging overhead when profiling a code. We have found on all systems that if you follow all the advise above your timings will be consistent within a variation of less than five percent.

3.5 Other PETSc Features

3.5.1 PETSc on a process subset

Users who wish to employ PETSc routines on only a subset of processes within a larger parallel job, or who wish to use a “master” process to coordinate the work of “slave” PETSc processes, should specify an alternative communicator for `PETSC_COMM_WORLD` by directly setting its value, for example to an existing `MPI_COMM_WORLD`,

```
PETSC_COMM_WORLD=MPI_COMM_WORLD; /* To use an existing MPI_COMM_WORLD */
```

before calling `PetscInitialize()`, but, obviously, after calling `MPI_Init()`.

3.5.2 Runtime Options

Allowing the user to modify parameters and options easily at runtime is very desirable for many applications. PETSc provides a simple mechanism to enable such customization. To print a list of available options for a given program, simply specify the option `-help` at runtime, e.g.,

```
mpiexec -n 1 ./ex1 -help
```

Note that all runtime options correspond to particular PETSc routines that can be explicitly called from within a program to set compile-time defaults. For many applications it is natural to use a combination of compile-time and runtime choices. For example, when solving a linear system, one could explicitly specify use of the Krylov subspace technique BiCGStab by calling

```
KSPSetType(ksp,KSPBCGS);
```

One could then override this choice at runtime with the option

```
-ksp_type tfqmr
```

to select the Transpose-Free QMR algorithm. (See *KSP: Linear System Solvers* for details.)

The remainder of this section discusses details of runtime options.

The Options Database

Each PETSc process maintains a database of option names and values (stored as text strings). This database is generated with the command `PetscInitialize()`, which is listed below in its C/C++ and Fortran variants, respectively:

```
PetscInitialize(int *argc,char ***args,const char *file,const char *help); /* C */
```

```
call PetscInitialize(character file,integer ierr) ! Fortran
```

The arguments `argc` and `args` (in the C/C++ version only) are the addresses of usual command line arguments, while the `file` is a name of a file that can contain additional options. By default this file is called `.petscrc` in the user’s home directory. The user can also specify options via the environmental variable `PETSC_OPTIONS`. The options are processed in the following order:

1. file
2. environmental variable

3. command line

Thus, the command line options supersede the environmental variable options, which in turn supersede the options file.

The file format for specifying options is

```
-optionname possible_value
-anotheroptionname possible_value
...
```

All of the option names must begin with a dash (-) and have no intervening spaces. Note that the option values cannot have intervening spaces either, and tab characters cannot be used between the option names and values. The user can employ any naming convention. For uniformity throughout PETSc, we employ the format `-[prefix_]package_option` (for instance, `-ksp_type`, `-mat_view ::info`, or `-mg_levels_ksp_type`).

Users can specify an alias for any option name (to avoid typing the sometimes lengthy default name) by adding an alias to the `.petscrc` file in the format

```
alias -newname -oldname
```

For example,

```
alias -kspt -ksp_type
alias -sd -start_in_debugger
```

Comments can be placed in the `.petscrc` file by using `#` in the first column of a line.

Options Prefixes

Options prefixes allow specific objects to be controlled from the options database. For instance, `PCMG` gives prefixes to its nested `KSP` objects; one may control the coarse grid solver by adding the `mg_coarse` prefix, for example `-mg_coarse_ksp_type preonly`. One may also use `KSPSetOptionsPrefix()`,`DMSetOptionsPrefix()` , `SNESSetOptionsPrefix()`, `TSSetOptionsPrefix()`, and similar functions to assign custom prefixes, useful for applications with multiple or nested solvers.

User-Defined PetscOptions

Any subroutine in a PETSc program can add entries to the database with the command

```
PetscOptionsSetValue(PetscOptions options,char *name,char *value);
```

though this is rarely done. To locate options in the database, one should use the commands

```
PetscOptionsHasName(PetscOptions options,char *pre,char *name,PetscBool *flg);
PetscOptionsGetInt(PetscOptions options,char *pre,char *name,PetscInt *value,
↪PetscBool *flg);
PetscOptionsGetReal(PetscOptions options,char *pre,char *name,PetscReal *value,
↪PetscBool *flg);
PetscOptionsGetString(PetscOptions options,char *pre,char *name,char *value,int␣
↪maxlen,PetscBool  *flg);
PetscOptionsGetStringArray(PetscOptions options,char *pre,char *name,char **values,
↪PetscInt *nmax,PetscBool *flg);
```

(continues on next page)

(continued from previous page)

```
PetscOptionsGetIntArray(PetscOptions options,char *pre,char *name,int *value,PetscInt␣
↪*nmax,PetscBool *flg);
PetscOptionsGetRealArray(PetscOptions options,char *pre,char *name,PetscReal *value,␣
↪PetscInt *nmax,PetscBool *flg);
```

All of these routines set `flg=PETSC_TRUE` if the corresponding option was found, `flg=PETSC_FALSE` if it was not found. The optional argument `pre` indicates that the true name of the option is the given name (with the dash "-" removed) prepended by the prefix `pre`. Usually `pre` should be set to `NULL` (or `PETSC_NULL_CHARACTER` for Fortran); its purpose is to allow someone to rename all the options in a package without knowing the names of the individual options. For example, when using block Jacobi preconditioning, the `KSP` and `PC` methods used on the individual blocks can be controlled via the options `-sub_ksp_type` and `-sub_pc_type`.

Keeping Track of Options

One useful means of keeping track of user-specified runtime options is use of `-options_view`, which prints to `stdout` during `PetscFinalize()` a table of all runtime options that the user has specified. A related option is `-options_left`, which prints the options table and indicates any options that have *not* been requested upon a call to `PetscFinalize()`. This feature is useful to check whether an option has been activated for a particular PETSc object (such as a solver or matrix format), or whether an option name may have been accidentally misspelled.

3.5.3 Viewers: Looking at PETSc Objects

PETSc employs a consistent scheme for examining, printing, and saving objects through commands of the form

```
XXXView(XXX obj,PetscViewer viewer);
```

Here `obj` is any PETSc object of type `XXX`, where `XXX` is `Mat`, `Vec`, `SNES`, etc. There are several predefined viewers.

- Passing in a zero (`0`) for the viewer causes the object to be printed to the screen; this is useful when viewing an object in a debugger but should be avoided in source code.
- `PETSC_VIEWER_STDOUT_SELF` and `PETSC_VIEWER_STDOUT_WORLD` causes the object to be printed to the screen.
- `PETSC_VIEWER_DRAW_SELF` `PETSC_VIEWER_DRAW_WORLD` causes the object to be drawn in a default X window.
- Passing in a viewer obtained by `PetscViewerDrawOpen()` causes the object to be displayed graphically. See *Graphics* for more on PETSc's graphics support.
- To save an object to a file in ASCII format, the user creates the viewer object with the command `PetscViewerASCIIOpen(MPI_Comm comm, char* file, PetscViewer *viewer)`. This object is analogous to `PETSC_VIEWER_STDOUT_SELF` (for a communicator of `MPI_COMM_SELF`) and `PETSC_VIEWER_STDOUT_WORLD` (for a parallel communicator).
- To save an object to a file in binary format, the user creates the viewer object with the command `PetscViewerBinaryOpen(MPI_Comm comm,char* file,PetscViewerBinaryType type, PetscViewer *viewer)`. Details of binary I/O are discussed below.

- Vector and matrix objects can be passed to a running MATLAB process with a viewer created by `PetscViewerSocketOpen(MPI_Comm comm,char *machine,int port,PetscViewer *viewer)`. For more, see *Sending Data to an Interactive MATLAB Session*.

The user can control the format of ASCII printed objects with viewers created by `PetscViewerASCIIOpen()` by calling

```
PetscViewerPushFormat(PetscViewer viewer,PetscViewerFormat format);
```

Formats include `PETSC_VIEWER_DEFAULT`, `PETSC_VIEWER_ASCII_MATLAB`, and `PETSC_VIEWER_ASCII_IMPL`. The implementation-specific format, `PETSC_VIEWER_ASCII_IMPL`, displays the object in the most natural way for a particular implementation.

The routines

```
PetscViewerPushFormat(PetscViewer viewer,PetscViewerFormat format);
PetscViewerPopFormat(PetscViewer viewer);
```

allow one to temporarily change the format of a viewer.

As discussed above, one can output PETSc objects in binary format by first opening a binary viewer with `PetscViewerBinaryOpen()` and then using `MatView()`, `VecView()`, etc. The corresponding routines for input of a binary object have the form `XXXLoad()`. In particular, matrix and vector binary input is handled by the following routines:

```
MatLoad(PetscViewer viewer,MatType outtype,Mat *newmat);
VecLoad(PetscViewer viewer,VecType outtype,Vec *newvec);
```

These routines generate parallel matrices and vectors if the viewer's communicator has more than one process. The particular matrix and vector formats are determined from the options database; see the manual pages for details.

One can provide additional information about matrix data for matrices stored on disk by providing an optional file `matrixfilename.info`, where `matrixfilename` is the name of the file containing the matrix. The format of the optional file is the same as the `.petscrc` file and can (currently) contain the following:

```
-matload_block_size <bs>
```

The block size indicates the size of blocks to use if the matrix is read into a block oriented data structure (for example, `MATMPIBAIJ`). The diagonal information `s1,s2,s3,...` indicates which (block) diagonals in the matrix have nonzero values.

Viewing From Options

Command-line options provide a particularly convenient way to view PETSc objects. All options of the form `-xxx_view` accept colon(`:`)-separated compound arguments which specify a viewer type, format, and/or destination (e.g. file name or socket) if appropriate. For example, to quickly export a binary file containing a matrix, one may use `-mat_view binary:matrix.out`, or to output to a MATLAB-compatible ASCII file, one may use `-mat_view ascii:matrix.m:ascii_matlab`. See the `PetscOptionsGetViewer()` man page for full details, as well as the `XXXViewFromOptions()` man pages (for instance, `PetscDrawSetFromOptions()`) for many other convenient command-line options.

Using Viewers to Check Load Imbalance

The PetscViewer format `PETSC_VIEWER_LOAD_BALANCE` will cause certain objects to display simple measures of their imbalance. For example

```
-n 4 ./ex32 -ksp_view_mat ::load_balance
```

will display

```
Nonzeros: Min 162  avg 168  max 174
```

indicating that one process has 162 nonzero entries in the matrix, the average number of nonzeros per process is 168 and the maximum number of nonzeros is 174. Similar for vectors one can see the load balancing with, for example,

```
-n 4 ./ex32 -ksp_view_rhs ::load_balance
```

The measurements of load balancing can also be done within the program with calls to the appropriate object viewer with the viewer format `PETSC_VIEWER_LOAD_BALANCE`.

3.5.4 Using SAWs with PETSc

The Scientific Application Web server, SAWs[8], allows one to monitor running PETSc applications from a browser. `./configure` PETSc with the additional option `--download-saws`. Options to use SAWs include

- `-saws_options` - allows setting values in the PETSc options database via the browser (works only on one process).
- `-stack_view saws` - allows monitoring the current stack frame that PETSc is in; refresh to see the new location.
- `-snes_monitor_saws, -ksp_monitor_saws` - monitor the solvers' iterations from the web browser.

For each of these you need to point your browser to `http://hostname:8080`, for example `http://localhost:8080`. Options that control behavior of SAWs include

- `-saws_log filename` - log all SAWs actions in a file.
- `-saws_https certfile` - use HTTPS instead of HTTP with a certificate.
- `-saws_port_auto_select` - have SAWs pick a port number instead of using 8080.
- `-saws_port port` - use `port` instead of 8080.
- `-saws_root rootdirectory` - local directory to which the SAWs browser will have read access.
- `-saws_local` - use the local file system to obtain the SAWS javascript files (they much be in `rootdirectory/js`).

Also see the manual pages for `PetscSAWsBlock`, `PetscObjectSAWsTakeAccess`, `PetscObjectSAWsGrantAccess`, `PetscObjectSAWsSetBlock`, `PetscStackSAWsGrantAccess` `PetscStackSAWsTakeAccess`, `KSPMonitorSAWs`, and `SNESMonitorSAWs`.

---

[8] Saws wiki on Bitbucket

3.5.5 Debugging

PETSc programs may be debugged using one of the two options below.

- `-start_in_debugger [noxterm,dbx,xxgdb,xdb,xldb,lldb] [-display name]` - start all processes in debugger
- `-on_error_attach_debugger [noxterm,dbx,xxgdb,xdb,xldb,lldb] [-display name]` - start debugger only on encountering an error

Note that, in general, debugging MPI programs cannot be done in the usual manner of starting the programming in the debugger (because then it cannot set up the MPI communication and remote processes).

By default the GNU debugger `gdb` is used when `-start_in_debugger` or `-on_error_attach_debugger` is specified. To employ either `xxgdb` or the common UNIX debugger `dbx`, one uses command line options as indicated above. On HP-UX machines the debugger `xdb` should be used instead of `dbx`; on RS/6000 machines the `xldb` debugger is supported as well. On OS X systems with XCode tools, `lldb` is available. By default, the debugger will be started in a new xterm (to enable running separate debuggers on each process), unless the option `noxterm` is used. In order to handle the MPI startup phase, the debugger command `cont` should be used to continue execution of the program within the debugger. Rerunning the program through the debugger requires terminating the first job and restarting the processor(s); the usual `run` option in the debugger will not correctly handle the MPI startup and should not be used. Not all debuggers work on all machines, the user may have to experiment to find one that works correctly.

You can select a subset of the processes to be debugged (the rest just run without the debugger) with the option

```
-debugger_ranks rank1,rank2,...
```

where you simply list the ranks you want the debugger to run with.

3.5.6 Error Handling

Errors are handled through the routine `PetscError()`. This routine checks a stack of error handlers and calls the one on the top. If the stack is empty, it selects `PetscTraceBackErrorHandler()`, which tries to print a traceback. A new error handler can be put on the stack with

```
PetscPushErrorHandler(PetscErrorCode (*HandlerFunction)(int line,char *dir,char *file,
↪char *message,int number,void*),void *HandlerContext)
```

The arguments to `HandlerFunction()` are the line number where the error occurred, the file in which the error was detected, the corresponding directory, the error message, the error integer, and the `HandlerContext.` The routine

```
PetscPopErrorHandler()
```

removes the last error handler and discards it.

PETSc provides two additional error handlers besides `PetscTraceBackErrorHandler()`:

```
PetscAbortErrorHandler()
PetscAttachErrorHandler()
```

The function `PetscAbortErrorHandler()` calls abort on encountering an error, while `PetscAttachErrorHandler()` attaches a debugger to the running process if an error is detected. At runtime, these error handlers can be set with the options `-on_error_abort` or `-on_error_attach_debugger [noxterm, dbx, xxgdb, xldb] [-display DISPLAY]`.

All PETSc calls can be traced (useful for determining where a program is hanging without running in the debugger) with the option

```
-log_trace [filename]
```

where `filename` is optional. By default the traces are printed to the screen. This can also be set with the command `PetscLogTraceBegin(FILE*)`.

It is also possible to trap signals by using the command

```
PetscPushSignalHandler( PetscErrorCode (*Handler)(int,void *),void *ctx);
```

The default handler `PetscSignalHandlerDefault()` calls `PetscError()` and then terminates. In general, a signal in PETSc indicates a catastrophic failure. Any error handler that the user provides should try to clean up only before exiting. By default all PETSc programs use the default signal handler, although the user can turn this off at runtime with the option `-no_signal_handler` .

There is a separate signal handler for floating-point exceptions. The option `-fp_trap` turns on the floating-point trap at runtime, and the routine

```
PetscSetFPTrap(PetscFPTrap flag);
```

can be used in-line. A `flag` of `PETSC_FP_TRAP_ON` indicates that floating-point exceptions should be trapped, while a value of `PETSC_FP_TRAP_OFF` (the default) indicates that they should be ignored. Note that on certain machines, in particular the IBM RS/6000, trapping is very expensive.

A small set of macros is used to make the error handling lightweight. These macros are used throughout the PETSc libraries and can be employed by the application programmer as well. When an error is first detected, one should set it by calling

```
SETERRQ(MPI_Comm comm,PetscErrorCode flag,,char *message);
```

The user should check the return codes for all PETSc routines (and possibly user-defined routines as well) with

```
ierr = PetscRoutine(...);CHKERRQ(PetscErrorCode ierr);
```

Likewise, all memory allocations should be checked with

```
ierr = PetscMalloc1(n, &ptr);CHKERRQ(ierr);
```

If this procedure is followed throughout all of the user's libraries and codes, any error will by default generate a clean traceback of the location of the error.

Note that the macro `PETSC_FUNCTION_NAME` is used to keep track of routine names during error tracebacks. Users need not worry about this macro in their application codes; however, users can take advantage of this feature if desired by setting this macro before each user-defined routine that may call `SETERRQ()`, `CHKERRQ()`. A simple example of usage is given below.

```
PetscErrorCode MyRoutine1() {
    /* Declarations Here */
    PetscFunctionBeginUser;
    /* code here */
    PetscFunctionReturn(0);
}
```

3.5.7 Numbers

PETSc supports the use of complex numbers in application programs written in C, C++, and Fortran. To do so, we employ either the C99 `complex` type or the C++ versions of the PETSc libraries in which the basic "scalar" datatype, given in PETSc codes by `PetscScalar`, is defined as `complex` (or `complex<double>` for machines using templated complex class libraries). To work with complex numbers, the user should run `./configure` with the additional option `--with-scalar-type=complex`. The installation instructions provide detailed instructions for installing PETSc. You can use `--with-clanguage=c` (the default) to use the C99 complex numbers or `--with-clanguage=c++` to use the C++ complex type[9].

Recall that each variant of the PETSc libraries is stored in a different directory, given by `${PETSC_DIR}/lib/${PETSC_ARCH}`

according to the architecture. Thus, the libraries for complex numbers are maintained separately from those for real numbers. When using any of the complex numbers versions of PETSc, *all* vector and matrix elements are treated as complex, even if their imaginary components are zero. Of course, one can elect to use only the real parts of the complex numbers when using the complex versions of the PETSc libraries; however, when working *only* with real numbers in a code, one should use a version of PETSc for real numbers for best efficiency.

The program KSP Tutorial ex11 solves a linear system with a complex coefficient matrix. Its Fortran counterpart is KSP Tutorial ex11f.

3.5.8 Parallel Communication

When used in a message-passing environment, all communication within PETSc is done through MPI, the message-passing interface standard [For94]. Any file that includes `petscsys.h` (or any other PETSc include file) can freely use any MPI routine.

3.5.9 Graphics

The PETSc graphics library is not intended to compete with high-quality graphics packages. Instead, it is intended to be easy to use interactively with PETSc programs. We urge users to generate their publication-quality graphics using a professional graphics package. If a user wants to hook certain packages into PETSc, he or she should send a message to petsc-maint@mcs.anl.gov; we will see whether it is reasonable to try to provide direct interfaces.

Windows as PetscViewers

For drawing predefined PETSc objects such as matrices and vectors, one may first create a viewer using the command

```
PetscViewerDrawOpen(MPI_Comm comm,char *display,char *title,int x,int y,int w,int h,
↪PetscViewer *viewer);
```

This viewer may be passed to any of the `XXXView()` routines. Alternately, one may use command-line options to quickly specify viewer formats, including `PetscDraw`-based ones; see *Viewing From Options*.

To draw directly into the viewer, one must obtain the `PetscDraw` object with the command

```
PetscViewerDrawGetDraw(PetscViewer viewer,PetscDraw *draw);
```

---

[9] Note that this option is not required to use PETSc with C++

Then one can call any of the `PetscDrawXXX` commands on the `draw` object. If one obtains the `draw` object in this manner, one does not call the `PetscDrawOpenX()` command discussed below.

Predefined viewers, `PETSC_VIEWER_DRAW_WORLD` and `PETSC_VIEWER_DRAW_SELF`, may be used at any time. Their initial use will cause the appropriate window to be created.

Implementations using OpenGL, TikZ, and other formats may be selected with `PetscDrawSetType()`. PETSc can also produce movies; see `PetscDrawSetSaveMovie()`, and note that command-line options can also be convenient; see the `PetscDrawSetFromOptions()` man page.

By default, PETSc drawing tools employ a private colormap, which remedies the problem of poor color choices for contour plots due to an external program's mangling of the colormap. Unfortunately, this may cause flashing of colors as the mouse is moved between the PETSc windows and other windows. Alternatively, a shared colormap can be used via the option `-draw_x_shared_colormap`.

Simple PetscDrawing

With the default format, one can open a window that is not associated with a viewer directly under the X11 Window System or OpenGL with the command

```
PetscDrawCreate(MPI_Comm comm,char *display,char *title,int x,int y,int w,int h,
↪PetscDraw *win);
PetscDrawSetFromOptions(win);
```

All drawing routines are performed relative to the window's coordinate system and viewport. By default, the drawing coordinates are from `(0,0)` to `(1,1)`, where `(0,0)` indicates the lower left corner of the window. The application program can change the window coordinates with the command

```
PetscDrawSetCoordinates(PetscDraw win,PetscReal xl,PetscReal yl,PetscReal xr,
↪PetscReal yr);
```

By default, graphics will be drawn in the entire window. To restrict the drawing to a portion of the window, one may use the command

```
PetscDrawSetViewPort(PetscDraw win,PetscReal xl,PetscReal yl,PetscReal xr,PetscReal␣
↪yr);
```

These arguments, which indicate the fraction of the window in which the drawing should be done, must satisfy $0 \leq \mathtt{xl} \leq \mathtt{xr} \leq 1$ and $0 \leq \mathtt{yl} \leq \mathtt{yr} \leq 1$.

To draw a line, one uses the command

```
PetscDrawLine(PetscDraw win,PetscReal xl,PetscReal yl,PetscReal xr,PetscReal yr,int␣
↪cl);
```

The argument `cl` indicates the color (which is an integer between 0 and 255) of the line. A list of predefined colors may be found in `include/petscdraw.h` and includes `PETSC_DRAW_BLACK`, `PETSC_DRAW_RED`, `PETSC_DRAW_BLUE` etc.

To ensure that all graphics actually have been displayed, one should use the command

```
PetscDrawFlush(PetscDraw win);
```

When displaying by using double buffering, which is set with the command

```
PetscDrawSetDoubleBuffer(PetscDraw win);
```

all processes must call

```
PetscDrawFlush(PetscDraw win);
```

in order to swap the buffers. From the options database one may use `-draw_pause n`, which causes the PETSc application to pause `n` seconds at each `PetscDrawPause()`. A time of `-1` indicates that the application should pause until receiving mouse input from the user.

Text can be drawn with commands

```
PetscDrawString(PetscDraw win,PetscReal x,PetscReal y,int color,char *text);
PetscDrawStringVertical(PetscDraw win,PetscReal x,PetscReal y,int color,const char␣
↪*text);
PetscDrawStringCentered(PetscDraw win,PetscReal x,PetscReal y,int color,const char␣
↪*text);
PetscDrawStringBoxed(PetscDraw draw,PetscReal sxl,PetscReal syl,int sc,int bc,const␣
↪char text[],PetscReal *w,PetscReal *h);
```

The user can set the text font size or determine it with the commands

```
PetscDrawStringSetSize(PetscDraw win,PetscReal width,PetscReal height);
PetscDrawStringGetSize(PetscDraw win,PetscReal *width,PetscReal *height);
```

Line Graphs

PETSc includes a set of routines for manipulating simple two-dimensional graphs. These routines, which begin with `PetscDrawAxisDraw()`, are usually not used directly by the application programmer. Instead, the programmer employs the line graph routines to draw simple line graphs. As shown in the *listing below*, line graphs are created with the command

```
PetscDrawLGCreate(PetscDraw win,PetscInt ncurves,PetscDrawLG *ctx);
```

The argument `ncurves` indicates how many curves are to be drawn. Points can be added to each of the curves with the command

```
PetscDrawLGAddPoint(PetscDrawLG ctx,PetscReal *x,PetscReal *y);
```

The arguments `x` and `y` are arrays containing the next point value for each curve. Several points for each curve may be added with

```
PetscDrawLGAddPoints(PetscDrawLG ctx,PetscInt n,PetscReal **x,PetscReal **y);
```

The line graph is drawn (or redrawn) with the command

```
PetscDrawLGDraw(PetscDrawLG ctx);
```

A line graph that is no longer needed can be destroyed with the command

```
PetscDrawLGDestroy(PetscDrawLG *ctx);
```

To plot new curves, one can reset a linegraph with the command

```
PetscDrawLGReset(PetscDrawLG ctx);
```

The line graph automatically determines the range of values to display on the two axes. The user can change these defaults with the command

```
PetscDrawLGSetLimits(PetscDrawLG ctx,PetscReal xmin,PetscReal xmax,PetscReal ymin,
↪PetscReal ymax);
```

It is also possible to change the display of the axes and to label them. This procedure is done by first obtaining the axes context with the command

```
PetscDrawLGGetAxis(PetscDrawLG ctx,PetscDrawAxis *axis);
```

One can set the axes' colors and labels, respectively, by using the commands

```
PetscDrawAxisSetColors(PetscDrawAxis axis,int axis_lines,int ticks,int text);
PetscDrawAxisSetLabels(PetscDrawAxis axis,char *top,char *x,char *y);
```

It is possible to turn off all graphics with the option `-nox`. This will prevent any windows from being opened or any drawing actions to be done. This is useful for running large jobs when the graphics overhead is too large, or for timing.

The full example, Draw Test ex3, follows.

Listing: `src/classes/draw/tests/ex3.c`

```
static char help[] = "Plots a simple line graph.\n";

#if defined(PETSC_APPLE_FRAMEWORK)
#import <PETSc/petscsys.h>
#import <PETSc/petscdraw.h>
#else

#include <petscsys.h>
#include <petscdraw.h>
#endif

int main(int argc,char **argv)
{
  PetscDraw          draw;
  PetscDrawLG        lg;
  PetscDrawAxis      axis;
  PetscInt           n = 15,i,x = 0,y = 0,width = 400,height = 300,nports = 1;
  PetscBool          useports,flg;
  const char         *xlabel,*ylabel,*toplabel,*legend;
  PetscReal          xd,yd;
  PetscDrawViewPorts *ports = NULL;
  PetscErrorCode     ierr;

  toplabel = "Top Label"; xlabel = "X-axis Label"; ylabel = "Y-axis Label"; legend =
↪"Legend";

  ierr = PetscInitialize(&argc,&argv,NULL,help);if (ierr) return ierr;
  ierr = PetscOptionsGetInt(NULL,NULL,"-x",&x,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-y",&y,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-width",&width,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-height",&height,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-n",&n,NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(NULL,NULL,"-nports",&nports,&useports);CHKERRQ(ierr);
  ierr = PetscOptionsHasName(NULL,NULL,"-nolegend",&flg);CHKERRQ(ierr);
  if (flg) legend = NULL;
```

(continues on next page)

(continued from previous page)

```
  ierr = PetscOptionsHasName(NULL,NULL,"-notoplabel",&flg);CHKERRQ(ierr);
  if (flg) toplabel = NULL;
  ierr = PetscOptionsHasName(NULL,NULL,"-noxlabel",&flg);CHKERRQ(ierr);
  if (flg) xlabel = NULL;
  ierr = PetscOptionsHasName(NULL,NULL,"-noylabel",&flg);CHKERRQ(ierr);
  if (flg) ylabel = NULL;
  ierr = PetscOptionsHasName(NULL,NULL,"-nolabels",&flg);CHKERRQ(ierr);
  if (flg) {toplabel = NULL; xlabel = NULL; ylabel = NULL;}

  ierr = PetscDrawCreate(PETSC_COMM_WORLD,0,"Title",x,y,width,height,&draw);
↪CHKERRQ(ierr);
  ierr = PetscDrawSetFromOptions(draw);CHKERRQ(ierr);
  if (useports) {
    ierr = PetscDrawViewPortsCreate(draw,nports,&ports);CHKERRQ(ierr);
    ierr = PetscDrawViewPortsSet(ports,0);CHKERRQ(ierr);
  }
  ierr = PetscDrawLGCreate(draw,1,&lg);CHKERRQ(ierr);
  ierr = PetscDrawLGSetUseMarkers(lg,PETSC_TRUE);CHKERRQ(ierr);
  ierr = PetscDrawLGGetAxis(lg,&axis);CHKERRQ(ierr);
  ierr = PetscDrawAxisSetColors(axis,PETSC_DRAW_BLACK,PETSC_DRAW_RED,PETSC_DRAW_BLUE);
↪CHKERRQ(ierr);
  ierr = PetscDrawAxisSetLabels(axis,toplabel,xlabel,ylabel);CHKERRQ(ierr);
  ierr = PetscDrawLGSetLegend(lg,&legend);CHKERRQ(ierr);
  ierr = PetscDrawLGSetFromOptions(lg);CHKERRQ(ierr);

  for (i=0; i<=n; i++) {
    xd   = (PetscReal)(i - 5); yd = xd*xd;
    ierr = PetscDrawLGAddPoint(lg,&xd,&yd);CHKERRQ(ierr);
  }
  ierr = PetscDrawLGDraw(lg);CHKERRQ(ierr);
  ierr = PetscDrawLGSave(lg);CHKERRQ(ierr);

  ierr = PetscDrawViewPortsDestroy(ports);CHKERRQ(ierr);
  ierr = PetscDrawLGDestroy(&lg);CHKERRQ(ierr);
  ierr = PetscDrawDestroy(&draw);CHKERRQ(ierr);
  ierr = PetscFinalize();
  return ierr;
}
```

Graphical Convergence Monitor

For both the linear and nonlinear solvers default routines allow one to graphically monitor convergence of the iterative method. These are accessed via the command line with `-ksp_monitor_lg_residualnorm` and `-snes_monitor_lg_residualnorm`. See also *Convergence Monitoring* and *Convergence Monitoring*.

The two functions used are `KSPMonitorLGResidualNorm()` and `KSPMonitorLGResidualNormCreate()`. These can easily be modified to serve specialized needs.

Disabling Graphics at Compile Time

To disable all X-window-based graphics, run `./configure` with the additional option `--with-x=0`

3.5.10 Emacs Users

Many PETSc developers use Emacs, which can be used as a “simple” text editor or a comprehensive development environment. For a more integrated development environment, we recommend using lsp-mode (or eglot) with clangd. The most convenient way to teach clangd what compilation flags to use is to install Bear (“build ear”) and run:

```
bear make -B
```

which will do a complete rebuild (`-B`) of PETSc and capture the compilation commands in a file named `compile_commands.json`, which will be automatically picked up by clangd. You can use the same procedure when building examples or your own project. It can also be used with any other editor that supports clangd, including VS Code and Vim. When lsp-mode is accompanied by flycheck, Emacs will provide real-time feedback and syntax checking, along with refactoring tools provided by clangd.

The easiest way to install packages in recent Emacs is to use the “Options” menu to select “Manage Emacs Packages”.

Tags

It is sometimes useful to cross-reference tags across projects. Regardless of whether you use lsp-mode, it can be useful to use GNU Global (install `gtags`) to provide reverse lookups (e.g. find all call sites for a given function) across all projects you might work on/browse. Tags for PETSc can be generated by running `make allgtags` from `${PETSC_DIR}`, or one can generate tags for all projects by running a command such as

```
find $PETSC_DIR/{include,src,tutorials,$PETSC_ARCH/include} any/other/paths \
   -regex '.*\.\(cc\|hh\|cpp\|cxx\|C\|hpp\|c\|h\|cu\)$' \
   | grep -v ftn-auto | gtags -f -
```

from your home directory or wherever you keep source code. If you are making large changes, it is useful to either set this up to run as a cron job or to make a convenient alias so that refreshing is easy. Then add the following to `~/.emacs` to enable gtags and specify key bindings.

```
(when (require 'gtags)
  (global-set-key (kbd "C-c f") 'gtags-find-file)
  (global-set-key (kbd "C-c .") 'gtags-find-tag)
  (global-set-key (kbd "C-c r") 'gtags-find-rtag)
  (global-set-key (kbd "C-c ,") 'gtags-pop-stack))
(add-hook 'c-mode-common-hook
          '(lambda () (gtags-mode t))) ; Or add to existing hook
```

A more basic alternative to the GNU Global (`gtags`) approach that does not require adding packages is to use the builtin `etags` feature. First, run `make alletags` from the PETSc home directory to generate the file `${PETSC_DIR}/TAGS`, and then from within Emacs, run

```
M-x visit-tags-table
```

where `M` denotes the Emacs Meta key, and enter the name of the `TAGS` file. Then the command `M-.` will cause Emacs to find the file and line number where a desired PETSc function is defined. Any string in any of the PETSc files can be found with the command `M-x tags-search`. To find repeated occurrences, one can simply use `M-,` to find the next occurrence.

3.5.11 VS Code Users

VS Code (unlike *Visual Studio Users*, described below) is an open source editor with a rich extension ecosystem. It has excellent integration with clangd and will automatically pick up `compile_commands.json` as produced by a command such as `bear make -B` (see *Emacs Users*). If you have no prior attachment to a specific code editor, we recommend trying VS Code.

3.5.12 Vi and Vim Users

See the *Emacs Users* discussion above for configuration of clangd, which provides integrated development environment.

If users develop application codes using Vi or Vim the `tags` feature can be used to search PETSc files quickly and efficiently. To use this feature, one should first check if the file, `${PETSC_DIR}/CTAGS` exists. If this file is not present, it should be generated by running `make alletags` from the PETSc home directory. Once the file exists, from Vi/Vim the user should issue the command

```
:set tags=CTAGS
```

from the `PETSC_DIR` directory and enter the name of the `CTAGS` file. Then the command "tag functionname" will cause Vi/Vim to find the file and line number where a desired PETSc function is defined. See, online tutorials for additional Vi/Vim options that allow searches, etc. It is also possible to use GNU Global with Vim; see the description for Emacs above.

3.5.13 Eclipse Users

If you are interested in developing code that uses PETSc from Eclipse or developing PETSc in Eclipse and have knowledge of how to do indexing and build libraries in Eclipse, please contact us at petsc-dev@mcs.anl.gov.

One way to index and build PETSc in Eclipse is as follows.

1. Open "File→Import→Git→Projects from Git". In the next two panels, you can either add your existing local repository or download PETSc from Bitbucket by providing the URL. Most Eclipse distributions come with Git support. If not, install the EGit plugin. When importing the project, select the wizard "Import as general project".
2. Right-click on the project (or the "File" menu on top) and select "New → Convert to a C/C++ Project (Adds C/C++ Nature)". In the setting window, choose "C Project" and specify the project type as "Shared Library".
3. Right-click on the C project and open the "Properties" panel. Under "C/C++ Build → Builder Settings", set the Build directory to `PETSC_DIR` and make sure "Generate Makefiles automatically" is unselected. Under the section "C/C++ General→Paths and Symbols", add the PETSc paths to "Includes".

```
     ${PETSC_DIR}/include
     ${PETSC_DIR}/${PETSC_ARCH}/include

Under the section “C/C++ General\ :math:`\rightarrow`\ index”, choose
“Use active build configuration”.
```

1. Configure PETSc normally outside Eclipse to generate a makefile and then build the project in Eclipse. The source code will be parsed by Eclipse.

If you launch Eclipse from the Dock on Mac OS X, `.bashrc` will not be loaded (a known OS X behavior, for security reasons). This will be a problem if you set the environment variables `PETSC_DIR` and `PETSC_ARCH` in `.bashrc`. A solution which involves replacing the executable can be found at `` `/questions/829749/launch-mac-eclipse-with-environment-variables-set `` </questions/829749/launch-mac-eclipse-with-environment-variables-set>‘__. Alternatively, you can add `PETSC_DIR` and `PETSC_ARCH` manually under “Properties → C/C++ Build → Environment”.

To allow an Eclipse code to compile with the PETSc include files and link with the PETSc libraries, a PETSc user has suggested the following.

1. Right-click on your C project and select “Properties → C/C++ Build → Settings”
2. A new window on the righthand side appears with various settings options. Select “Includes” and add the required PETSc paths,

```
${PETSC_DIR}/include
${PETSC_DIR}/${PETSC_ARCH}/include
```

1. Select “Libraries” under the header Linker and set the library search path:

```
   ${PETSC_DIR}/${PETSC_ARCH}/lib

and the libraries, for example
```

```
m, petsc, stdc++, mpichxx, mpich, lapack, blas, gfortran, dl, rt,gcc_s, pthread, X11
```

Another PETSc user has provided the following steps to build an Eclipse index for PETSc that can be used with their own code, without compiling PETSc source into their project.

1. In the user project source directory, create a symlink to the PETSC `src/` directory.
2. Refresh the project explorer in Eclipse, so the new symlink is followed.
3. Right-click on the project in the project explorer, and choose “Index → Rebuild”. The index should now be built.
4. Right-click on the PETSc symlink in the project explorer, and choose “Exclude from build…” to make sure Eclipse does not try to compile PETSc with the project.

For further examples of using Eclipse with a PETSc-based application, see the documentation for LaMEM[10].

3.5.14 Qt Creator Users

This information was provided by Mohammad Mirzadeh. The Qt Creator IDE is part of the Qt SDK, developed for cross-platform GUI programming using C++. It is available under GPL v3, LGPL v2 and a commercial license and may be obtained, either as part of the Qt SDK or as stand-alone software. It supports automatic makefile generation using cross-platform `qmake` and `cmake` build systems as well as allowing one to import projects based on existing, possibly hand-written, makefiles. Qt Creator has a visual debugger using GDB and LLDB (on Linux and OS X) or Microsoft’s CDB (on Windows) as backends. It also has an interface to Valgrind’s “memcheck” and “callgrind” tools to detect memory leaks and profile code. It has built-in support for a variety of version control systems including git, mercurial, and subversion. Finally, Qt Creator comes fully equipped with auto-completion, function look-up, and code refactoring tools. This enables one to easily browse source files, find relevant functions, and refactor them across an entire project.

---

[10] `doc/` at https://bitbucket.org/bkaus/lamem

Creating a Project

When using Qt Creator with `qmake`, one needs a `.pro` file. This configuration file tells Qt Creator about all build/compile options and locations of source files. One may start with a blank `.pro` file and fill in configuration options as needed. For example:

```
# The name of the application executable
TARGET = ex1

# There are two ways to add PETSc functionality
# 1-Manual: Set all include path and libs required by PETSc
PETSC_INCLUDE = path/to/petsc_includes # e.g. obtained via running `make␣
 ↪getincludedirs'
PETSC_LIBS = path/to/petsc_libs # e.g. obtained via running `make getlinklibs'

INCLUDEPATH += $$PETSC_INCLUDES
LIBS += $$PETSC_LIBS

# 2-Automatic: Use the PKGCONFIG funtionality
# NOTE: PETSc.pc must be in the pkgconfig path. You might need to adjust PKG_CONFIG_
 ↪PATH
CONFIG += link_pkgconfig
PKGCONFIG += PETSc

# Set appropriate compiler and its flags
QMAKE_CC = path/to/mpicc
QMAKE_CXX = path/to/mpicxx # if this is a cpp project
QMAKE_LINK = path/to/mpicxx # if this is a cpp project

QMAKE_CFLAGS   += -O3 # add extra flags here
QMAKE_CXXFLAGS += -O3
QMAKE_LFLAGS   += -O3

# Add all files that must be compiled
SOURCES += ex1.c source1.c source2.cpp

HEADERS += source1.h source2.h

# OTHER_FILES are ignored during compilation but will be shown in file panel in Qt␣
 ↪Creator
OTHER_FILES += \
    path/to/resource_file \
    path/to/another_file
```

In this example, keywords include:

- `TARGET`: The name of the application executable.
- `INCLUDEPATH`: Used at compile time to point to required include files. Essentially, it is used as an `-I \$\$INCLUDEPATH` flag for the compiler. This should include all application-specific header files and those related to PETSc (which may be found via `make getincludedirs`).
- `LIBS`: Defines all required external libraries to link with the application. To get PETSc's linking libraries, use `make getlinklibs`.
- `CONFIG`: Configuration options to be used by `qmake`. Here, the option `link_pkgconfig` instructs `qmake` to internally use `pkgconfig` to resolve `INCLUDEPATH` and `LIBS` variables.
- `PKGCONFIG`: Name of the configuration file (the `.pc` file – here `PETSc.pc`) to be passed to `pkg-config`. Note that for this functionality to work, `PETSc.pc` must be in path which might require

adjusting the `PKG_CONFIG_PATH` enviroment variable. For more information see the Qt Creator documentation.

- `QMAKE_CC` and `QMAKE_CXX`: Define which C/C++ compilers use.
- `QMAKE_LINK`: Defines the proper linker to be used. Relevant if compiling C++ projects.
- `QMAKE_CFLAGS`, `QMAKE_CXXFLAGS` and `QMAKE_LFLAGS`: Set the corresponding compile and linking flags.
- `SOURCES`: Source files to be compiled.
- `HEADERS`: Header files required by the application.
- `OTHER_FILES`: Other files to include (source, header, or any other extension). Note that none of the source files placed here are compiled.

More options can be included in a `.pro` file; see https://doc.qt.io/qt-5/qmake-project-files.html. Once the `.pro` file is generated, the user can simply open it via Qt Creator. Upon opening, one has the option to create two different build options, debug and release, and switch between the two. For more information on using the Qt Creator interface and other more advanced aspects of the IDE, refer to https://www.qt.io/qt-features-libraries-apis-tools-and-ide/

3.5.15 Visual Studio Users

To use PETSc from MS Visual Studio, one would have to compile a PETSc example with its corresponding makefile and then transcribe all compiler and linker options used in this build into a Visual Studio project file, in the appropriate format in Visual Studio project settings.

3.5.16 XCode Users (The Apple GUI Development System)

Mac OS X

Follow the instructions in `$PETSC_DIR/systems/Apple/OSX/bin/makeall` to build the PETSc framework and documentation suitable for use in XCode.

You can then use the PETSc framework in `$PETSC_DIR/arch-osx/PETSc.framework` in the usual manner for Apple frameworks. See the examples in `$PETSC_DIR/systems/Apple/OSX/examples`. When working in XCode, things like function name completion should work for all PETSc functions as well as MPI functions. You must also link against the Apple `Accelerate.framework`.

iPhone/iPad iOS

Follow the instructions in `$PETSC_DIR/systems/Apple/iOS/bin/iosbuilder.py` to build the PETSc library for use on the iPhone/iPad.

You can then use the PETSc static library in `$PETSC_DIR/arch-osx/libPETSc.a` in the usual manner for Apple libraries inside your iOS XCode projects; see the examples in `$PETSC_DIR/systems/Apple/iOS/examples`. You must also link against the Apple `Accelerate.framework`.

3.6 Unimportant and Advanced Features of Matrices and Solvers

This chapter introduces additional features of the PETSc matrices and solvers. Since most PETSc users should not need to use these features, we recommend skipping this chapter during an initial reading.

3.6.1 Extracting Submatrices

One can extract a (parallel) submatrix from a given (parallel) using

```
MatCreateSubMatrix(Mat A,IS rows,IS cols,MatReuse call,Mat *B);
```

This extracts the `rows` and `col`umns of the matrix `A` into `B`. If call is `MAT_INITIAL_MATRIX` it will create the matrix `B`. If call is `MAT_REUSE_MATRIX` it will reuse the `B` created with a previous call.

3.6.2 Matrix Factorization

Normally, PETSc users will access the matrix solvers through the `KSP` interface, as discussed in *KSP: Linear System Solvers*, but the underlying factorization and triangular solve routines are also directly accessible to the user.

The LU and Cholesky matrix factorizations are split into two or three stages depending on the user's needs. The first stage is to calculate an ordering for the matrix. The ordering generally is done to reduce fill in a sparse factorization; it does not make much sense for a dense matrix.

```
MatGetOrdering(Mat matrix,MatOrderingType type,IS* rowperm,IS* colperm);
```

The currently available alternatives for the ordering `type` are

- `MATORDERINGNATURAL` - Natural
- `MATORDERINGND` - Nested Dissection
- `MATORDERING1WD` - One-way Dissection
- `MATORDERINGRCM` - Reverse Cuthill-McKee
- `MATORDERINGQMD` - Quotient Minimum Degree

These orderings can also be set through the options database.

Certain matrix formats may support only a subset of these; more options may be added. Check the manual pages for up-to-date information. All of these orderings are symmetric at the moment; ordering routines that are not symmetric may be added. Currently we support orderings only for sequential matrices.

Users can add their own ordering routines by providing a function with the calling sequence

```
int reorder(Mat A,MatOrderingType type,IS* rowperm,IS* colperm);
```

Here `A` is the matrix for which we wish to generate a new ordering, `type` may be ignored and `rowperm` and `colperm` are the row and column permutations generated by the ordering routine. The user registers the ordering routine with the command

```
MatOrderingRegister(MatOrderingType ordname,char *path,char *sname,PetscErrorCode␣
↪(*reorder)(Mat,MatOrderingType,IS*,IS*)));
```

The input argument `ordname` is a string of the user's choice, either an ordering defined in `petscmat.h` or the name of a new ordering introduced by the user. See the code in `src/mat/impls/order/sorder.c` and other files in that directory for examples on how the reordering routines may be written.

Once the reordering routine has been registered, it can be selected for use at runtime with the command line option `-pc_factor_mat_ordering_type ordname`. If reordering from the API, the user should provide the `ordname` as the second input argument of `MatGetOrdering()`.

The following routines perform complete, in-place, symbolic, and numerical factorizations for symmetric and nonsymmetric matrices, respectively:

```
MatCholeskyFactor(Mat matrix,IS permutation,const MatFactorInfo *info);
MatLUFactor(Mat matrix,IS rowpermutation,IS columnpermutation,const MatFactorInfo␣
↪*info);
```

The argument `info->fill > 1` is the predicted fill expected in the factored matrix, as a ratio of the original fill. For example, `info->fill=2.0` would indicate that one expects the factored matrix to have twice as many nonzeros as the original.

For sparse matrices it is very unlikely that the factorization is actually done in-place. More likely, new space is allocated for the factored matrix and the old space deallocated, but to the user it appears in-place because the factored matrix replaces the unfactored matrix.

The two factorization stages can also be performed separately, by using the out-of-place mode, first one obtains that matrix object that will hold the factor

```
MatGetFactor(Mat matrix,MatSolverType package,MatFactorType ftype,Mat *factor);
```

and then performs the factorization

```
MatCholeskyFactorSymbolic(Mat factor,Mat matrix,IS perm,const MatFactorInfo *info);
MatLUFactorSymbolic(Mat factor,Mat matrix,IS rowperm,IS colperm,const MatFactorInfo␣
↪*info);
MatCholeskyFactorNumeric(Mat factor,Mat matrix,const MatFactorInfo);
MatLUFactorNumeric(Mat factor,Mat matrix,const MatFactorInfo *info);
```

In this case, the contents of the matrix `result` is undefined between the symbolic and numeric factorization stages. It is possible to reuse the symbolic factorization. For the second and succeeding factorizations, one simply calls the numerical factorization with a new input `matrix` and the *same* factored `result` matrix. It is *essential* that the new input matrix have exactly the same nonzero structure as the original factored matrix. (The numerical factorization merely overwrites the numerical values in the factored matrix and does not disturb the symbolic portion, thus enabling reuse of the symbolic phase.) In general, calling `XXXFactorSymbolic` with a dense matrix will do nothing except allocate the new matrix; the `XXXFactorNumeric` routines will do all of the work.

Why provide the plain `XXXfactor` routines when one could simply call the two-stage routines? The answer is that if one desires in-place factorization of a sparse matrix, the intermediate stage between the symbolic and numeric phases cannot be stored in a `result` matrix, and it does not make sense to store the intermediate values inside the original matrix that is being transformed. We originally made the combined factor routines do either in-place or out-of-place factorization, but then decided that this approach was not needed and could easily lead to confusion.

We do not currently support sparse matrix factorization with pivoting for numerical stability. This is because trying to both reduce fill and do pivoting can become quite complicated. Instead, we provide a poor stepchild substitute. After one has obtained a reordering, with `MatGetOrdering(Mat A,MatOrdering type,IS *row,IS *col)` one may call

```
MatReorderForNonzeroDiagonal(Mat A,PetscReal tol,IS row, IS col);
```

which will try to reorder the columns to ensure that no values along the diagonal are smaller than `tol` in a absolute value. If small values are detected and corrected for, a nonsymmetric permutation of the rows and columns will result. This is not guaranteed to work, but may help if one was simply unlucky in the original ordering. When using the `KSP` solver interface the option `-pc_factor_nonzeros_along_diagonal <tol>` may be used. Here, `tol` is an optional tolerance to decide if a value is nonzero; by default it is `1.e-10`.

Once a matrix has been factored, it is natural to solve linear systems. The following four routines enable this process:

```
MatSolve(Mat A,Vec x, Vec y);
MatSolveTranspose(Mat A, Vec x, Vec y);
MatSolveAdd(Mat A,Vec x, Vec y, Vec w);
MatSolveTransposeAdd(Mat A, Vec x, Vec y, Vec w);
```

matrix `A` of these routines must have been obtained from a factorization routine; otherwise, an error will be generated. In general, the user should use the `KSP` solvers introduced in the next chapter rather than using these factorization and solve routines directly.

3.6.3 Unimportant Details of KSP

`PetscDrawAxisDraw()`, are usually not used directly by the application programmer Again, virtually all users should use `KSP` through the `KSP` interface and, thus, will not need to know the details that follow.

It is possible to generate a Krylov subspace context with the command

```
KSPCreate(MPI_Comm comm,KSP *kps);
```

Before using the Krylov context, one must set the matrix-vector multiplication routine and the preconditioner with the commands

```
PCSetOperators(PC pc,Mat Amat,Mat Pmat);
KSPSetPC(KSP ksp,PC pc);
```

In addition, the `KSP` solver must be initialized with

```
KSPSetUp(KSP ksp);
```

Solving a linear system is done with the command

```
KSPSolve(KSP ksp,Vec b,Vec x);
```

Finally, the `KSP` context should be destroyed with

```
KSPDestroy(KSP *ksp);
```

It may seem strange to put the matrix in the preconditioner rather than directly in the `KSP`; this decision was the result of much agonizing. The reason is that for SSOR with Eisenstat's trick, and certain other preconditioners, the preconditioner has to change the matrix-vector multiply. This procedure could not be done cleanly if the matrix were stashed in the `KSP` context that `PC` cannot access.

Any preconditioner can supply not only the preconditioner, but also a routine that essentially performs a complete Richardson step. The reason for this is mainly SOR. To use SOR in the Richardson framework, that is,

$$u^{n+1} = u^n + B(f - Au^n),$$

is much more expensive than just updating the values. With this addition it is reasonable to state that *all* our iterative methods are obtained by combining a preconditioner from the `PC` package with a Krylov method from the `KSP` package. This strategy makes things much simpler conceptually, so (we hope) clean code will result. *Note*: We had this idea already implicitly in older versions of `KSP`, but, for instance, just doing Gauss-Seidel with Richardson in old `KSP` was much more expensive than it had to be. With PETSc this should not be a problem.

3.6.4 Unimportant Details of PC

Most users will obtain their preconditioner contexts from the `KSP` context with the command `KSPGetPC()`. It is possible to create, manipulate, and destroy `PC` contexts directly, although this capability should rarely be needed. To create a `PC` context, one uses the command

```
PCCreate(MPI_Comm comm,PC *pc);
```

The routine

```
PCSetType(PC pc,PCType method);
```

sets the preconditioner method to be used. The routine

```
PCSetOperators(PC pc,Mat Amat,Mat Pmat);
```

set the matrices that are to be used with the preconditioner. The routine

```
PCGetOperators(PC pc,Mat *Amat,Mat *Pmat);
```

returns the values set with `PCSetOperators()`.

The preconditioners in PETSc can be used in several ways. The two most basic routines simply apply the preconditioner or its transpose and are given, respectively, by

```
PCApply(PC pc,Vec x,Vec y);
PCApplyTranspose(PC pc,Vec x,Vec y);
```

In particular, for a preconditioner matrix, `B`, that has been set via `PCSetOperators(pc,Amat,Pmat)`, the routine PCApply(pc,x,y) computes $y = B^{-1}x$ by solving the linear system $By = x$ with the specified preconditioner method.

Additional preconditioner routines are

```
PCApplyBAorAB(PC pc,PCSide right,Vec x,Vec y,Vec work);
PCApplyBAorABTranspose(PC pc,PCSide right,Vec x,Vec y,Vec work);
PCApplyRichardson(PC pc,Vec x,Vec y,Vec work,PetscReal rtol,PetscReal atol, PetscReal␣
↪dtol,PetscInt maxits,PetscBool zeroguess,PetscInt *outits,
↪PCRichardsonConvergedReason*);
```

The first two routines apply the action of the matrix followed by the preconditioner or the preconditioner followed by the matrix depending on whether the `right` is `PC_LEFT` or `PC_RIGHT`. The final routine applies `its` iterations of Richardson's method. The last three routines are provided to improve efficiency for certain Krylov subspace methods.

A `PC` context that is no longer needed can be destroyed with the command

```
PCDestroy(PC *pc);
```

3.7 Running PETSc Tests

3.7.1 Quick start with the tests

For testing builds, the general invocation from the `PETSC_DIR` is:

```
make [-j <n>] -f gmakefile test PETSC_ARCH=<PETSC_ARCH>
```

For testing `./configure` that used the `--prefix` option, the general invocation from the installation (prefix) directory is:

```
make [-j <n>] -f share/petsc/examples/gmakefile test
```

For a full list of options, use

```
make -f gmakefile help-test
```

3.7.2 Understanding test output and more information

As discussed in *Running PETSc Tests*, users should set `PETSC_DIR` and `PETSC_ARCH` before running the tests, or can provide them on the command line as below.

To check if the libraries are working do:

```
make PETSC_DIR=<PETSC_DIR> PETSC_ARCH=<PETSC_ARCH> test
```

A comprehensive set of tests can be run with

```
make PETSC_DIR=<PETSC_DIR> PETSC_ARCH=<PETSC_ARCH> alltests
```

or

```
make [-j <n>] -f gmakefile test PETSC_ARCH=<PETSC_ARCH>
```

Depending on your machine's configuration running the full test suite (above) can take from a few minutes to a couple hours. Note that currently we do not have a mechanism for automatically running the test suite on batch computer systems except to obtain an interactive compute node (via the batch system) and run the tests on that node (this assumes that the compilers are available on the interactive compute nodes.

The test reporting system classifies them according to the Test Anywhere Protocal (TAP)[11]. In brief, the categories are

- `ok` The test passed.
- `not ok` The test failed.
- `not ok #SKIP` The test was skipped, usually because build requirements were not met (for example, an external solver library was required, but PETSc was not `./configure` for that library.) compiled against it).
- `ok #TODO` The test is under development by the developers.

---

[11] See https://testanything.org/tap-specification.html

The tests are a series of shell scripts, generated by information contained within the test source file, that are invoked by the makefile system. The tests are run in `${PETSC_DIR}/${PETSC_ARCH}/tests` with the same directory as the source tree underneath. For testing installs, the default location is `${PREFIX_DIR}/tests` but this can be changed with the `TESTDIR` location. (See *Directory Structure*). A label is used to denote where it can be found within the source tree. For example, test `vec_vec_tutorials-ex6`, which can be run e.g. with

```
make -f gmakefile test search='vec_vec_tutorials-ex6'
```

(see the discussion of `search` below), denotes the shell script:

```
${PETSC_DIR}/${PETSC_ARCH}/tests/vec/vec/tutorials/runex6.sh
```

These shell scripts can be run independently in those directories, and take arguments to show the commands run, change arguments, etc. Use the `-h` option to the shell script to see these options.

Often, you want to run only a subset of tests. Our makefiles use `gmake`'s wildcard syntax. In this syntax, `%` is a wild card character and is passed in using the `search` argument. Two wildcard characters cannot be used in a search, so the `searchin` argument is used to provide the equivalent of `%pattern%` search. The default examples have default arguments, and we often wish to test examples with various arguments; we use the `argsearch` argument for these searches. Like `searchin`, it does not use wildcards, but rather whether the string is within the arguments.

Some examples are:

```
make -f gmakefile test search='ts%'                      # Run all TS examples
make -f gmakefile test searchin='tutorials'              # Run all tutorials
make -f gmakefile test search='ts%' searchin='tutorials' # Run all TS tutorials
make -f gmakefile test argsearch='cuda'                  # Run examples with cuda in␣
↪arguments
make -f gmakefile test test-fail='1'
make -f gmakefile test query='requires' queryval='*MPI_PROCESS_SHARED_MEMORY*'
```

It is useful before invoking the tests to see what targets will be run. The `print-test` target helps with this:

```
make -f gmakefile print-test argsearch='cuda'
```

To see all of the test targets which would be run, this command can be used:

```
make -f gmakefile print-test
```

For testing in install directories, some examples are:

```
cd ${PREFIX_DIR}; make -f share/petsc/examples/gmakefile.test test TESTDIR=mytests
```

or

```
cd ${PREFIX_DIR}/share/petsc/examples; make -f gmakefile test TESTDIR=$PWD/mytests
```

where the latter is needed to make have it run in the local directory instead of `$PREFIX_DIR`.

To learn more about the test system details, one can look at the the PETSc developers documentation.

3.8 Acknowledgments

We thank all PETSc users for their many suggestions, bug reports, and encouragement.

Recent contributors to PETSc can be seen by visualizing the history of the PETSc git repository, for example at github.com/petsc/petsc/graphs/contributors.

Earlier contributors to PETSc include:

- Asbjorn Hoiland Aarrestad - the explicit Runge-Kutta implementations (`TSRK`)
- G. Anciaux and J. Roman - the interfaces to the partitioning packages PTScotch, Chaco, and Party;
- Allison Baker - the flexible GMRES (`KSPFGMRES`) and LGMRES (`KSPLGMRES`) code;
- Chad Carroll - Win32 graphics;
- Ethan Coon - the `PetscBag` and many bug fixes;
- Cameron Cooper - portions of the `VecScatter` routines;
- Patrick Farrell - `PCPATCH` and `SNESPATCH`;
- Paulo Goldfeld - the balancing Neumann-Neumann preconditioner (`` `PCNN``);
- Matt Hille;
- Joel Malard - the BICGStab(l) implementation (`KSPBCGSL`);
- Paul Mullowney, enhancements to portions of the NVIDIA GPU interface;
- Dave May - the GCR implementation (`KSPGCR`);
- Peter Mell - portions of the `DMDA` routines;
- Richard Mills - the `AIJPERM` matrix format (`MATAIJPERM`) for the Cray X1 and universal F90 array interface;
- Victor Minden - the NVIDIA GPU interface;
- Lawrence Mitchell - `PCPATCH` and `SNESPATCH`;
- Todd Munson - the LUSOL (sparse solver in MINOS) interface (`MATSOLVERLUSOL`) and several Krylov methods;
- Adam Powell - the PETSc Debian package;
- Robert Scheichl - the MINRES implementation (`KSPMINRES`);
- Kerry Stevens - the pthread-based `Vec` and `Mat` classes plus the various thread pools (no longer available);
- Karen Toonen - design and implementation of much of the PETSc web pages;
- Desire Nuentsa Wakam - the deflated GMRES implementation (`KSPDGMRES`);
- Florian Wechsung - `PCPATCH` and `SNESPATCH`;
- Liyang Xu - the interface to PVODE, now SUNDIALS/CVODE (`TSSUNDIALS`).

PETSc source code contains modified routines from the following public domain software packages:

- LINPACK - dense matrix factorization and solve; converted to C using `f2c` and then hand-optimized for small matrix sizes, for block matrix data structures;
- MINPACK - see page ; sequential matrix coloring routines for finite difference Jacobian evaluations; converted to C using `f2c`;

- SPARSPAK - see page ; matrix reordering routines, converted to C using `f2c`;
- libtfs - the efficient, parallel direct solver developed by Henry Tufo and Paul Fischer for the direct solution of a coarse grid problem (a linear system with very few degrees of freedom per processor).

PETSc interfaces to the following external software:

- BLAS and LAPACK - numerical linear algebra;
- Chaco - A graph partitioning package;
  http://www.cs.sandia.gov/CRF/chac.html
- Elemental - Jack Poulson's parallel dense matrix solver package;
  http://libelemental.org/
- HDF5 - the data model, library, and file format for storing and managing data,
  https://support.hdfgroup.org/HDF5/
- hypre - the LLNL preconditioner library;
  https://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods
- LUSOL - sparse LU factorization code (part of MINOS) developed by Michael Saunders, Systems Optimization Laboratory, Stanford University;
  http://www.sbsi-sol-optimize.com/
- MATLAB - see page ;
- Metis/ParMeTiS - see page , parallel graph partitioner,
  https://www-users.cs.umn.edu/~karypis/metis/
- MUMPS - see page , MUltifrontal Massively Parallel sparse direct Solver developed by Patrick Amestoy, Iain Duff, Jacko Koster, and Jean-Yves L'Excellent;
  http://mumps.enseeiht.fr/
- Party - A graph partitioning package;
- PaStiX - Parallel sparse LU and Cholesky solvers;
  http://pastix.gforge.inria.fr/
- PTScotch - A graph partitioning package;
  http://www.labri.fr/Perso/~pelegrin/scotch/
- SPAI - for parallel sparse approximate inverse preconditioning;
  https://cccs.unibas.ch/lehre/software-packages/
- SuiteSparse - sequential sparse solvers, see page , developed by Timothy A. Davis;
  http://faculty.cse.tamu.edu/davis/suitesparse.html
- SUNDIALS/CVODE - see page , parallel ODE integrator;
  https://computation.llnl.gov/projects/sundials
- SuperLU and SuperLU_Dist - see page , the efficient sparse LU codes developed by Jim Demmel, Xiaoye S. Li, and John Gilbert;
  https://crd-legacy.lbl.gov/~xiaoye/SuperLU
- STRUMPACK - the STRUctured Matrix Package;
  https://portal.nersc.gov/project/sparse/strumpack/
- Triangle and Tetgen - mesh generation packages;
  https://www.cs.cmu.edu/~quake/triangle.html
  http://wias-berlin.de/software/tetgen/

- Trilinos/ML - Sandia's main multigrid preconditioning package; https://trilinos.github.io/
- Zoltan - graph partitioners from Sandia National Laboratory; http://www.cs.sandia.gov/zoltan/

These are all optional packages and do not need to be installed to use PETSc.

PETSc software is developed and maintained using

- Emacs editor
- Git revision control system
- Python

PETSc documentation has been generated using

- Sowing text processing tools developed by Bill Gropp
- c2html

BIBLIOGRAPHY

[For94] MPI Forum. MPI: a message-passing interface standard. *International J. Supercomputing Applications*, 1994.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[EM17] Jennifer B Erway and Roummel F Marcia. On solving large-scale limited-memory quasi-newton equations. *Linear Algebra and its Applications*, 515:196–225, 2017.

[GL89] J. C. Gilbert and C. Lemarechal. Some numerical experiments with variable-storage Quasi-Newton algorithms. *Mathematical Programming*, 45:407–434, 1989.

[Gri12] Andreas Griewank. Broyden updating, the good and the bad! *Optimization Stories, Documenta Mathematica. Extra Volume: Optimization Stories*, pages 301–315, 2012.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

[CS97] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. Technical Report CU-CS 843-97, Computer Science Department, University of Colorado-Boulder, 1997. (accepted by SIAM J. of Scientific Computing).

[Eis81] S. Eisenstat. Efficient implementation of a class of CG methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[EES83] S.C. Eisenstat, H.C. Elman, and M.H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357, 1983.

[FGN92] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica. Cambridge University Press, 1992.

[Fre93] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Stat. Comput.*, 14:470–482, 1993.

[GAMV13] P. Ghysels, T.J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013.

[GV14] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications. doi:10.1016/j.parco.2013.06.001.

[HS52] Magnus R. Hestenes and Eduard Steifel. Methods of conjugate gradients for solving linear systems. *J. Research of the National Bureau of Standards*, 49:409–436, 1952.

[ISG15] Tobin Isaac, Georg Stadler, and Omar Ghattas. Solution of nonlinear Stokes equations discretized by high-order finite elements on nonconforming and anisotropic meshes, with application to ice sheet dynamics. *SIAM J. Sci. Comput.*, 37(6):804–833, 2015.

[Not00] Yvan Notay. Flexible Conjugate Gradients. *SIAM Journal on Scientific Computing*, 22(4):1444–1460, 2000.

[PS75] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[Saa93] Youcef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993. doi:10.1137/0914028.

[SS86] Youcef Saad and Martin H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003. doi:10.1016/S1570-579X(01)80025-2.

[Son89] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[vdV03] H. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, 2003. ISBN 9780521818285.

[vandVorst92] H. A. van der Vorst. BiCGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[SSM16] P. Sanan, S. M. Schnepp, and D. A. May. Pipelined, flexible Krylov subspace methods. *SIAM Journal on Scientific Computing*, 38(5):C441–C470, 2016. doi:10.1137/15M1049130.

[SBjorstadG96] Barry F. Smith, Petter Bjørstad, and William D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996. URL: http://www.mcs.anl.gov/~bsmith/ddbook.html.

[BS90] Peter N. Brown and Youcef Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[DS83] J. E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[EW96] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact Newton method. *SIAM J. Scientific Computing*, 17:16–32, 1996.

[JP93] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[MoreSGH84] Jorge J. Moré, Danny C. Sorenson, Burton S. Garbow, and Kenneth E. Hillstrom. The MINPACK project. In Wayne R. Cowell, editor, *Sources and Development of Mathematical Software*, 88–111. 1984.

[PW98] M. Pernice and H. F. Walker. NITSOL: a Newton iterative solver for nonlinear systems. *SIAM J. Sci. Stat. Comput.*, 19:302–318, 1998.

[BKST15] Peter R. Brune, Matthew G. Knepley, Barry F. Smith, and Xuemin Tu. Composing scalable nonlinear algebraic solvers. *SIAM Review*, 57(4):535–565, 2015. http://www.mcs.anl.gov/papers/P2010-0112.pdf. URL: http://www.mcs.anl.gov/papers/P2010-0112.pdf, doi:10.1137/130936725.

[ARS97] U.M. Ascher, S.J. Ruuth, and R.J. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25:151–167, 1997.

[AP98] Uri M Ascher and Linda R Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Volume 61. SIAM, 1998.

[BPR11] S. Boscarino, L. Pareschi, and G. Russo. Implicit-explicit Runge-Kutta schemes for hyperbolic systems and kinetic equations in the diffusion limit. Arxiv preprint arXiv:1110.4375, 2011.

[BJW07] J.C. Butcher, Z. Jackiewicz, and W.M. Wright. Error propagation of general linear methods for ordinary differential equations. *Journal of Complexity*, 23(4-6):560–580, 2007. doi:10.1016/j.jco.2007.01.009.

[Con16] E.M. Constantinescu. Estimating global errors in time stepping. *ArXiv e-prints*, March 2016. arXiv:1503.05166.

[CS10] E.M. Constantinescu and A. Sandu. Extrapolated implicit-explicit time stepping. *SIAM Journal on Scientific Computing*, 31(6):4452–4477, 2010. doi:10.1137/080732833.

[GKC13] F.X. Giraldo, J.F. Kelly, and E.M. Constantinescu. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA). *SIAM Journal on Scientific Computing*, 35(5):B1162–B1194, 2013. doi:10.1137/120876034.

[JWH00] K.E. Jansen, C.H. Whiting, and G.M. Hulbert. A generalized-alpha method for integrating the filtered Navier–Stokes equations with a stabilized finite element method. *Computer Methods in Applied Mechanics and Engineering*, 190(3):305–319, 2000.

[KC03] C.A. Kennedy and M.H. Carpenter. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.*, 44(1-2):139–181, 2003. doi:10.1016/S0168-9274(02)00138-1.

[Ket08] D.I. Ketcheson. Highly efficient strong stability-preserving Runge–Kutta methods with low-storage implementations. *SIAM Journal on Scientific Computing*, 30(4):2113–2136, 2008. doi:10.1137/07070485X.

[OColomesB16] Oriol Colomés and Santiago Badia. Segregated Runge–Kutta methods for the incompressible Navier–Stokes equations. *International Journal for Numerical Methods in Engineering*, 105(5):372–400, 2016.

[PR05] L. Pareschi and G. Russo. Implicit-explicit Runge-Kutta schemes and applications to hyperbolic systems with relaxation. *Journal of Scientific Computing*, 25(1):129–155, 2005.

[RA05] J. Rang and L. Angermann. New Rosenbrock W-methods of order 3 for partial differential algebraic equations of index 1. *BIT Numerical Mathematics*, 45(4):761–787, 2005.

[SVB+97] A. Sandu, J.G. Verwer, J.G. Blom, E.J. Spee, G.R. Carmichael, and F.A. Potra. Benchmarking stiff ode solvers for atmospheric chemistry problems II: Rosenbrock solvers. *Atmospheric Environment*, 31(20):3459–3472, 1997.

[BBKL11] Achi Brandt, James Brannick, Karsten Kahl, and Irene Livshits. Bootstrap AMG. *SIAM Journal on Scientific Computing*, 33(2):612–632, 2011.

[Getal] William Gropp and et. al. MPICH Web page. http://www.mpich.org. URL: http://www.mpich.org.

[HL91] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, August 1991.

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC