PETSc/TAO Users Manual

Revision 3.15

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, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY


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
Alexandria, 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.
# Contents

## 1 Introduction to PETSc

1.1 About This Manual ........................................... 1
1.2 Getting Started ............................................ 2
  1.2.1 Suggested Reading ...................................... 2
  1.2.2 Running PETSc Programs .............................. 4
  1.2.3 Writing PETSc Programs .............................. 5
  1.2.4 Simple PETSc Examples ................................ 6
  1.2.5 Profiling Programs .................................... 17
  1.2.6 Writing Application Codes with PETSc ............... 19
  1.2.7 Directory Structure .................................. 20

## 2 Programming with PETSc/TAO

2.1 Vectors and Parallel Data ................................... 23
  2.1.1 Creating and Assembling Vectors ..................... 23
  2.1.2 Basic Vector Operations ................................ 26
  2.1.3 Indexing and Ordering ................................ 27
  2.1.4 Structured Grids Using Distributed Arrays .......... 30
  2.1.5 Vectors Related to Unstructured Grids ............. 36
2.2 Matrices .................................................. 39
  2.2.1 Creating and Assembling Matrices .................... 40
  2.2.2 Basic Matrix Operations ............................. 49
  2.2.3 Matrix-Free Matrices .................................. 50
  2.2.4 Other Matrix Operations ................................ 51
  2.2.5 Partitioning .......................................... 53
2.3 KSP: Linear System Solvers .................................. 55
  2.3.1 Using KSP .............................................. 55
  2.3.2 Solving Successive Linear Systems ................... 56
  2.3.3 Krylov Methods ...................................... 57
  2.3.4 Preconditioners ...................................... 62
  2.3.5 Solving Block Matrices .............................. 75
  2.3.6 Solving Singular Systems ............................ 78
  2.3.7 Using External Linear Solvers ....................... 78
2.4 SNES: Nonlinear Solvers .................................... 80
  2.4.1 Basic SNES Usage ...................................... 81
  2.4.2 The Nonlinear Solvers ................................ 89
  2.4.3 General Options ...................................... 93
  2.4.4 Inexact Newton-like Methods ......................... 95
  2.4.5 Matrix-Free Methods .................................. 96
  2.4.6 Finite Difference Jacobian Approximations ........ 111
  2.4.7 Variational Inequalities ............................ 113
3 Additional Information

3.1 PETSc for Fortran Users
   3.1.1 Synopsis ........................................ 193
   3.1.2 C vs. Fortran Interfaces ........................ 193
   3.1.3 Sample Fortran Programs ......................... 197

3.2 Using MATLAB with PETSc ...................................
   3.2.1 Dumping Data for MATLAB ........................... 209
   3.2.2 Sending Data to an Interactive MATLAB Session .... 209
   3.2.3 Using the MATLAB Compute Engine .................. 210
   3.2.4 Licensing the MATLAB Compute Engine on a cluster ... 212

3.3 Profiling ..........................................
   3.3.1 Basic Profiling Information ....................... 213
   3.3.2 Profiling Application Codes ........................ 221
   3.3.3 Profiling Multiple Sections of Code ................ 222
   3.3.4 Restricting Event Logging .......................... 223
   3.3.5 Interpreting -log_info Output: Informative Messages ... 223
   3.3.6 Time ........................................ 224
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.7 Saving Output to a File</td>
<td>224</td>
</tr>
<tr>
<td>3.3.8 Accurate Profiling and Paging Overheads</td>
<td>224</td>
</tr>
<tr>
<td>3.4 Hints for Performance Tuning</td>
<td>225</td>
</tr>
<tr>
<td>3.4.1 Maximizing Memory Bandwidth</td>
<td>225</td>
</tr>
<tr>
<td>3.4.2 Performance Pitfalls and Advice</td>
<td>231</td>
</tr>
<tr>
<td>3.5 The Use of BLAS and LAPACK in PETSc and external libraries</td>
<td>236</td>
</tr>
<tr>
<td>3.5.1 32 or 64 bit BLAS/LAPACK integers</td>
<td>236</td>
</tr>
<tr>
<td>3.5.2 Shared memory BLAS/LAPACK parallelism</td>
<td>236</td>
</tr>
<tr>
<td>3.5.3 Available BLAS/LAPACK libraries</td>
<td>237</td>
</tr>
<tr>
<td>3.6 Other PETSc Features</td>
<td>237</td>
</tr>
<tr>
<td>3.6.1 PETSc on a process subset</td>
<td>237</td>
</tr>
<tr>
<td>3.6.2 Runtime Options</td>
<td>237</td>
</tr>
<tr>
<td>3.6.3 Viewers: Looking at PETSc Objects</td>
<td>241</td>
</tr>
<tr>
<td>3.6.4 Using SAWs with PETSc</td>
<td>243</td>
</tr>
<tr>
<td>3.6.5 Debugging</td>
<td>244</td>
</tr>
<tr>
<td>3.6.6 Error Handling</td>
<td>244</td>
</tr>
<tr>
<td>3.6.7 Numbers</td>
<td>246</td>
</tr>
<tr>
<td>3.6.8 Parallel Communication</td>
<td>246</td>
</tr>
<tr>
<td>3.6.9 Graphics</td>
<td>246</td>
</tr>
<tr>
<td>3.6.10 Emacs Users</td>
<td>251</td>
</tr>
<tr>
<td>3.6.11 VS Code Users</td>
<td>252</td>
</tr>
<tr>
<td>3.6.12 Vi and Vim Users</td>
<td>252</td>
</tr>
<tr>
<td>3.6.13 Eclipse Users</td>
<td>254</td>
</tr>
<tr>
<td>3.6.14 Qt Creator Users</td>
<td>255</td>
</tr>
<tr>
<td>3.6.15 Visual Studio Users</td>
<td>257</td>
</tr>
<tr>
<td>3.6.16 XCode Users (The Apple GUI Development System)</td>
<td>257</td>
</tr>
<tr>
<td>3.7 Unimportant and Advanced Features of Matrices and Solvers</td>
<td>257</td>
</tr>
<tr>
<td>3.7.1 Extracting Submatrices</td>
<td>257</td>
</tr>
<tr>
<td>3.7.2 Matrix Factorization</td>
<td>258</td>
</tr>
<tr>
<td>3.7.3 Unimportant Details of KSP</td>
<td>260</td>
</tr>
<tr>
<td>3.7.4 Unimportant Details of PC</td>
<td>260</td>
</tr>
<tr>
<td>3.8 Running PETSc Tests</td>
<td>261</td>
</tr>
<tr>
<td>3.8.1 Quick start with the tests</td>
<td>261</td>
</tr>
<tr>
<td>3.8.2 Understanding test output and more information</td>
<td>262</td>
</tr>
<tr>
<td>3.9 Acknowledgments</td>
<td>263</td>
</tr>
</tbody>
</table>

Bibliography 267
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://petsc.org/), 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 /community/mailing for information on contacting support.

We welcome corrections for our documentation. You can submit corrections to petsc-maint@mcs.anl.gov or post an issue. Add the label docs. Also add docs-only in the typical case where the only changes are to documentation.
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/TAO 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 consists 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/TAO
- 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.
## 1.2. Getting Started

Fig. 1.1: Numerical Libraries in PETSc

<table>
<thead>
<tr>
<th>Application Codes</th>
<th>Higher-Level Libraries</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETSc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TS (Time Steppers)

<table>
<thead>
<tr>
<th>Euler</th>
<th>Backward Euler</th>
<th>RK</th>
<th>BDF</th>
<th>SSP</th>
<th>ARKIMEX</th>
<th>Rosenbrock-W</th>
<th>...</th>
</tr>
</thead>
</table>

### SNES (Nonlinear Solvers)

<table>
<thead>
<tr>
<th>Newton Line Search</th>
<th>Newton Trust Region</th>
<th>FAS</th>
<th>NGMRES</th>
<th>NASM</th>
<th>ASPIN</th>
<th>...</th>
</tr>
</thead>
</table>

### KSP (Krylov Subspace Methods)

<table>
<thead>
<tr>
<th>GMRES</th>
<th>Richardson</th>
<th>CG</th>
<th>CGS</th>
<th>Bi-CGStab</th>
<th>TFQMR</th>
<th>MINRES</th>
<th>GCR</th>
<th>Chebyshev</th>
<th>Pipelined CG</th>
<th>...</th>
</tr>
</thead>
</table>

### PC (Preconditioners)

<table>
<thead>
<tr>
<th>Additive Schwarz</th>
<th>Block Jacobi</th>
<th>Jacobi</th>
<th>ICC</th>
<th>ILU</th>
<th>LU</th>
<th>SOR</th>
<th>MG</th>
<th>AMG</th>
<th>BDDC</th>
<th>Shell</th>
<th>...</th>
</tr>
</thead>
</table>

### Mat (Operators)

<table>
<thead>
<tr>
<th>Compressed Sparse Row</th>
<th>Block CSR</th>
<th>Symmetric Block CSR</th>
<th>Dense</th>
<th>CUSPARSE</th>
<th>ViennaCL</th>
<th>FFT</th>
<th>Shell</th>
<th>...</th>
</tr>
</thead>
</table>

### Vec (Vectors)

<table>
<thead>
<tr>
<th>Standard</th>
<th>CUDA</th>
<th>ViennaCL</th>
<th>...</th>
</tr>
</thead>
</table>

### IS (Index Sets)

<table>
<thead>
<tr>
<th>General</th>
<th>Block</th>
<th>Stride</th>
<th>...</th>
</tr>
</thead>
</table>

### BLAS/LAPACK

| BLAS/LAPACK | MPI | ... |
Programming with PETSc/TAO 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.

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 here. 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 enable Python bindings (i.e petsc4py) with the configure option --with-petsc4py=1. 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/petscmiexec -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

```c
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, `.petsrc`, 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

```fortran
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:

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

(continues on next page)
```c
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!");
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);
}
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. */
```

(continues on next page)
```c
ierr = VecSet(u,1.0);CHKERRQ(ierr);
ierr = MatMult(A,u,b);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 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.

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).

Include Files

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

```c
#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

```c
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 n, PetscInt *im, 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:

```c
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, $A$, and matrix from which the preconditioner is constructed, $P$). 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.

```bash
$ 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: 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
```

(continues on next page)
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 occurred. 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:

```c
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 <../../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\nInput parameters include: \n
- view_exact_sol : write exact solution vector to stdout\n";

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

#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;

  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);
  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 = MatMPIBAIJSetPreallocation(A,1,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatMPISELLSetPreallocation(A,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatSeqBAIJSetPreallocation(A,1,5,NULL);CHKERRQ(ierr);
  ierr = MatMPISELLSetPreallocation(A,5,NULL,5,NULL);CHKERRQ(ierr);
  ierr = MatSeqSBAIJSetPreallocation(A,1,5,NULL);CHKERRQ(ierr);

  (continues on next page)
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.

```c
ierr = MatGetOwnershipRange(A, &Istart, &Iend); CHKERRQ(ierr);
```

Set matrix elements for the 2-D, five-point stencil in parallel.
- 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 \( J = I_i + n \) instead of \( J = I_i + m \) as you might expect. The more standard ordering would first do all variables for \( y = h \), then \( y = 2h \) etc.

```c
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<n-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:
```c
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
```

A is symmetric. Set symmetric flag to enable ICC/Cholesky preconditioner
```c
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).

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

```c
/*
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);
```

```c
/* 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);}
```

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

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

```c
/*
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);
```

```c
/*
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);

/* 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://petsc.org/ or given in the file $PETSC_DIR/docs/faq.html.

```bash
$ 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 -Wquashed-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,-Wl,suppress -Wl,-Wno-strict-aliasing -Wno-unknown-pragmas -Wquashed-arguments -fvisibility=hidden -g3 -o ex2.o

$PETSC_DIR/lib/petsc/bin/petscmiexec -n 1 ./ex2
Norm of error 0.000156044 iterations 6
$PETSC_DIR/lib/petsc/bin/petscmiexec -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.

```bash
$ $PETSC_DIR/lib/petsc/bin/petscmiexec -n 1 ./ex2
...```

(continues on next page)
### Event Count Time (sec) Flops

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Time (sec)</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecMDot</td>
<td>11</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecNorm</td>
<td>14</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecScale</td>
<td>5</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecCopy</td>
<td>0</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecSet</td>
<td>11</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecAXPY</td>
<td>10</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecMAXPY</td>
<td>10</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>VecNormalize</td>
<td>14</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatMult</td>
<td>24</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatSolve</td>
<td>24</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatLUFactorNum</td>
<td>10</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatILUFactorSym</td>
<td>0</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatAssemblyBegin</td>
<td>1</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatAssemblyEnd</td>
<td>1</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatGetRowIJ</td>
<td>1</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatGetOrdering</td>
<td>1</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>MatView</td>
<td>2</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>KSPGMRESOrthog</td>
<td>10</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>KSPSetUp</td>
<td>11</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>KSPSolve</td>
<td>19</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>PCSetUp</td>
<td>11</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>PCApply</td>
<td>1</td>
<td>0.00e+00</td>
<td>0.00e+00</td>
</tr>
</tbody>
</table>
Memory usage is given in bytes:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Creations</th>
<th>Destructions</th>
<th>Memory</th>
<th>Descendants' Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reports information only for process 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--- Event Stage 0: Main Stage

<table>
<thead>
<tr>
<th>Vector</th>
<th>8</th>
<th>8</th>
<th>76224</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>2</td>
<td>2</td>
<td>134212</td>
<td>0</td>
</tr>
<tr>
<td>Krylov Solver</td>
<td>1</td>
<td>1</td>
<td>18400</td>
<td>0</td>
</tr>
<tr>
<td>Preconditioner</td>
<td>1</td>
<td>1</td>
<td>1032</td>
<td>0</td>
</tr>
<tr>
<td>Index Set</td>
<td>3</td>
<td>3</td>
<td>10328</td>
<td>0</td>
</tr>
<tr>
<td>Viewer</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

========================================================================================================================

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://petsc.org/release/documentation/](https://petsc.org/release/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 $PETSC_DIR/share/petsc/Makefile.user Makefile`
   Note: `Makefile.user` relies on the `pkg-config` tool, and PETSC_DIR and PETSC_ARCH variables. If `pkg-config` is not available, use `$PETSC_DIR/src/snes/tutorials/makefile`, it is useful for simple usages. `Makefile.user` provides a better template that helps in adapting to more complex usages (for example, an application project might already have a complex makefile, and is now adding in PETSc library usage to it).
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 getincludelibs 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 \texttt{CFLAGS=\{\textit{list of -I and other flags}\}} and \texttt{FFLAGS=\{\textit{list of -I and other flags}\}} in your makefile.

### 1.2.7 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 \texttt{manual.pdf} contains the hyperlinked users manual, suitable for printing or on-screen viewing. Includes the subdirectory \texttt{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 \textit{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,
    - \texttt{* is} - index sets,
  - **mat** - matrices,
  - **ksp** - complete linear equations solvers,
    - \texttt{* ksp} - Krylov subspace accelerators,
    - \texttt{* 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,
    - \texttt{* logging} - PETSc logging and profiling routines,
    - \texttt{* classes} - low-level classes
      - \texttt{draw} - simple graphics,
      - \texttt{viewer} - mechanism for printing and visualizing PETSc objects,
      - \texttt{bag} - mechanism for saving and loading from disk user data stored in C structs.
      - \texttt{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/TAO

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

```c
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

```c
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

```c
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

```c
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

```c
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

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

followed by

```c
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

```c
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

```c
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

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

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

```c
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

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

To destroy an array of vectors, use the command

```c
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

```c
VecCreateSeqWithArray(PETSC_COMM_SELF, PetscInt bs, PetscInt n, PetscScalar *array, Vec *v);
```

and

```c
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: \( n \times \text{sizeof(PetscScalar)} \).
2.1.2 Basic Vector Operations

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

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 \( \sum |x_i| \), the 2-norm is \( (\sum x_i^2)^{1/2} \) and the infinity norm is \( \max |x_i| \).

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

```c
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:

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

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

```c
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.

```c
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 \textit{Array Arguments}. It is important to note that `VecGetArray()` and `VecRestoreArray()` do \textit{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

\begin{verbatim}
VecGetLocalSize(Vec v, PetscInt *size);
\end{verbatim}

The global vector length can be determined by

\begin{verbatim}
VecGetSize(Vec v, PetscInt *size);
\end{verbatim}

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

\begin{verbatim}
VecDot(Vec x, Vec y, PetscScalar *dot);
VecMDot(Vec x, Vec y, PetscInt nv, Vec y[], PetscScalar *dot);
VecNorm(Vec x, NormType NORM_2, PetscReal *norm2);
VecNorm(Vec x, NormType NORM_1, PetscReal *norm1);
\end{verbatim}

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

\begin{verbatim}
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);
\end{verbatim}

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.

\subsection*{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 relabel the nodes, requiring adjustment of the other data structures that refer to these objects; see Figure \textit{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 \texttt{AO} (application ordering), which enables mapping between different global (cross-process) numbering schemes and the \texttt{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

```c
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

\[ \{0,1,2,3,4\} \rightarrow \{3,2,1,4,0\} \]

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

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

on the first process and

```c
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

```c
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

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

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

The mapping routines

```c
AOPetscToApplicationIS(AO ao, IS indices);
AOApplicationToPetscIS(AO ao, IS indices);
```
will map index sets (IS objects) between orderings. Both the A0XxxToYyy() and A0XxxToYyyIS() routines can be used regardless of whether the AO was created with a A0CreateBasic() or A0CreateBasicIS().

The AO context should be destroyed with AODestroy(AO *ao) and viewed with A0View(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 A0XxToxx() 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, _PetscCopyMode *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

\[
\text{VecSetLocalToGlobalMapping(Vec } v, \text{ISLocalToGlobalMapping ctx)};
\]

and then call

\[
\text{VecSetValuesLocal(Vec } x, \text{PetscInt n,<const PetscInt indices[]}, \text{const PetscScalar values[], INSERT_VALUES});
\]

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, DMDAs 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 and Star-type stencil](image)

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

```c
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; \( m \times n \) must equal the number of processes in the MPI communicator, \( \text{comm} \). Instead of specifying the process layout, one may use \text{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 \text{xperiod} and \text{yperiod}, which can be \text{DM_BOUNDARY_NONE} (no periodicity), \text{DM_BOUNDARY_PERIODIC} (periodic in that direction), \text{DM_BOUNDARY_TWIST} (periodic in that direction, but identified in reverse order), \text{DM_BOUNDARY_GHOSTED}, or \text{DM_BOUNDARY_MIRROR}. The argument \( \text{dof} \) 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 \( \text{lx} \) and \( \text{ly} \) may contain the number of nodes along the \( x \) and \( y \) axis for each cell, i.e. the dimension of \( \text{lx} \) is \( m \) and the dimension of \( \text{ly} \) is \( n \); alternately, \text{NULL} may be passed in.

Two types of distributed array communication data structures can be created, as specified by \text{st}. Star-type stencils that radiate outward only in the coordinate directions are indicated by \text{DMDA_STENCIL_STAR}, while box-type stencils are specified by \text{DMDA_STENCIL_BOX}. For example, for the two-dimensional case, \text{DMDA_STENCIL_STAR} with width 1 corresponds to the standard 5-point stencil, while \text{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 \text{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:

```c
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 w,
   _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 \( \text{comm} \) must call \text{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)
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.

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

and write residual evaluation using

```plaintext
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

```plaintext
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

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

followed by
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.

<table>
<thead>
<tr>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 27 28</td>
<td>29 30</td>
<td>22 23 24</td>
<td>29 30</td>
</tr>
<tr>
<td>21 22 23</td>
<td>24 25</td>
<td>19 20 21</td>
<td>27 28</td>
</tr>
<tr>
<td>16 17 18</td>
<td>19 20</td>
<td>16 17 18</td>
<td>25 26</td>
</tr>
<tr>
<td>11 12 13</td>
<td>14 15</td>
<td>11 12 13</td>
<td>14 15</td>
</tr>
<tr>
<td>6 7 8</td>
<td>9 10</td>
<td>4 5</td>
<td>6 12 13</td>
</tr>
<tr>
<td>1 2 3</td>
<td>4 5</td>
<td>1 2 3</td>
<td>10 11</td>
</tr>
</tbody>
</table>

Processor 0     Processor 1                       Processor 0     Processor 1
Natural Ordering PETSc Ordering

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, 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:

```c
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

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

Index sets can be destroyed with the command

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

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

```c
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

```c
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

```c
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:

```c
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

```c
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.
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

```c
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

```c
VecCreateGhost(MPI_Comm comm, PetscInt n, PetscInt N, PetscInt nghost, PetscInt * ghosts, Vec *vv)
```
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 \( \text{ghosts} \) 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);

Do local calculations from localin to localout ...
VecGhostRestoreLocalForm(Vec globalin, Vec *localin);
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

```c
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

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

```c
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 $m \times n$ 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

```c
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 `MatSetValues-
BlockedLocal()`.

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

```c
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

```c
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

```c
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

```plaintext
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

```c
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 \texttt{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.

\[
\begin{pmatrix}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34 \\
\end{pmatrix}
\]

The “diagonal” submatrix, \(\text{d}\), 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, \(\text{o}\), 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 \(\text{d\_nz}\) to 2 (since each row has 2 nonzeros) or, alternatively, set \(\text{d\_nnz}\) to \(\{2,2,2\}\). The \(\text{o\_nz}\) could be set to 2 since each row of the \(\text{o}\) matrix has 2 nonzeros, or \(\text{o\_nnz}\) could be set to \(\{2,2,2\}\).

For the second process the \(\text{d}\) submatrix is given by

\[
\begin{pmatrix}
15 & 16 & 17 \\
19 & 20 & 21 \\
22 & 23 & 0 \\
\end{pmatrix},
\]

Thus, one could set \(\text{d\_nz}\) to 3, since the maximum number of nonzeros in each row is 3, or alternatively one could set \(\text{d\_nnz}\) to \(\{3,3,2\}\), thereby indicating that the first two rows will have 3 nonzeros while the third has 2. The corresponding \(\text{o}\) 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 \(\text{o\_nz}\) to 2 or \(\text{o\_nnz}\) to \(\{2,1,1\}\).

Note that the user never directly works with the \(\text{d}\) and \(\text{o}\) 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 \texttt{-info} will print information about the success of preallocation during matrix assembly. For the \texttt{MATMPIAIJ} and \texttt{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
indicate that very few values have been generated on different processes. On the other hand, 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
```

display 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>
<thead>
<tr>
<th>Method</th>
<th>PETSc Type</th>
<th>Name</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Good” Broyden [ref-Gri12]</td>
<td>MATLMVMBrdn</td>
<td>lmvmbrdn</td>
<td>Square</td>
</tr>
<tr>
<td>“Bad” Broyden [ref-Gri12]</td>
<td>MATLMVMBadBrdn</td>
<td>lmvmbadbrdn</td>
<td>Square</td>
</tr>
<tr>
<td>Symmetric Rank-1 [ref-NW99]</td>
<td>MATLMVMSR1</td>
<td>lmvmsr1</td>
<td>Symmetric</td>
</tr>
<tr>
<td>Davidon-Fletcher-Powell (DFP) [ref-NW99]</td>
<td>MATLMVMDFP</td>
<td>lmvmdfp</td>
<td>SPD</td>
</tr>
<tr>
<td>Broyden-Fletcher-Goldfarb-Shanno (BFGS) [ref-NW99]</td>
<td>MATLMVMBFGS</td>
<td>lmvmbfgs</td>
<td>SPD</td>
</tr>
<tr>
<td>Restricted Broyden Family [ref-EM17]</td>
<td>MATLMVMSymBrdn</td>
<td>lmvmsymbrdn</td>
<td>SPD</td>
</tr>
<tr>
<td>Restricted Broyden Family (full-memory diagonal)</td>
<td>MATLMVMDiagBrdn</td>
<td>lmvmdiagbrdn</td>
<td>SPD</td>
</tr>
</tbody>
</table>

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 \( \sigma \) 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 [ref-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 [ref-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 \( m \times n \), the user should call

```c
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
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 $3 \times 3$ 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

```c
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`.
Fig. 2.3: The relationship between spaces used for coupled assembly.
that does not provide values or implement `MatMult()`, but does implement `MatSetValueLocal()` and, if `isrow`, `iscol` have a constant block size, `MatSetValueBlockedLocal()`. 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 Tutorial 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

\[
\text{MatMult(Mat } A, \text{ Vec } x, \text{ Vec } y); 
\]

the vectors \( x \) and \( y \) should be generated with

\[
\text{VecCreateMPI(MPI Comm comm, n, N, } &x); \\
\text{VecCreateMPI(MPI Comm comm, m, M, } &y); 
\]

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,

\[
\text{MatMultTranspose(Mat } A, \text{ Vec } x, \text{ Vec } y); 
\]

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

\[
\text{MatMultAdd(Mat } A, \text{ Vec } x, \text{ Vec } y, \text{ Vec } w); \\
\text{MatMultTransposeAdd(Mat } A, \text{ Vec } x, \text{ Vec } y, \text{ Vec } w); 
\]

These routines, respectively, produce \( w = A \times x + y \) and \( w = A^T \times x + y \). 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

\[
\text{MatView(Mat mat,PETSC_VIEWER_STDOUT_WORLD);} 
\]

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

\[
\text{MatView(Mat mat,PETSC_VIEWER_DRAW_WORLD);} 
\]

Also one can use

\[
\text{MatView(Mat mat,PetscViewer viewer);} 
\]

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

2.2. Matrices 49
### Table 2.3: PETSc Matrix Operations

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

### Table 2.4: Values of MatStructure

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAME_NONZERO_PATTERN</td>
<td>the matrices have an identical nonzero pattern</td>
</tr>
<tr>
<td>DIFFERENT_NONZERO_PATTERN</td>
<td>the matrices may have a different nonzero pattern</td>
</tr>
<tr>
<td>SUBSET_NONZERO_PATTERN</td>
<td>the second matrix has a subset of the nonzeros in the first matrix</td>
</tr>
<tr>
<td>UNKNOWN_NONZERO_PATTERN</td>
<td>there is nothing known about the relation between the nonzero patterns of the two matrices</td>
</tr>
</tbody>
</table>

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:

```c
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

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

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

```c
MatShellSetOperation(Mat mat, MatOperation MATOP_MULT, (void (*)(void))PetscErrorCode, /*UserMult(Mat,Vec,Vec));
```

---

50 Chapter 2. Programming with PETSc/TAO
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

```c
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

```c
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.

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

```c
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

```c
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

$$Ax = b,$$  

(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

\[
\text{KSPDestroy(KSP } \ast \text{ksp);}
\]

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

\[
\text{KSPSetUp(KSP ksp);}
\]

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,

\[
\text{KSPGetPC(KSP ksp, PC } \ast \text{pc);}
\]

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

```c
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 KSP-Type 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:

```c
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

```c
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

```c
KSPGMRESSetCGSRefinementType(KSP ksp, KSPGMRES CGSRefinementType type)
```

or via the options database with

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

The values for `KSPGMRES CGSRefinementType()` 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 `KSPGMRES ModifiedGramSchmidtOrthogonalization()` 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

```c
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

\[(M_L^{-1}AM_R^{-1})(M_Rx) = M_L^{-1}b,\]  

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),

\[r = b - Ax = b - AM_R^{-1}M_Rx,\]

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

\[r_L = M_L^{-1}b - M_L^{-1}Ax = M_L^{-1}r.\]

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>
<thead>
<tr>
<th>Method</th>
<th>KSPType</th>
<th>Options Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richardson</td>
<td>KSPRICHARDSON</td>
<td>richardson</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>KSPCHEBYSHEV</td>
<td>chebyshev</td>
</tr>
<tr>
<td>Conjugate Gradient [HS52]</td>
<td>KSPCG</td>
<td>cg</td>
</tr>
<tr>
<td>Pipelined Conjugate Gradients [GV14]</td>
<td>KSPPIPECG</td>
<td>pipecg</td>
</tr>
<tr>
<td>Pipelined Conjugate Gradients (Gropp)</td>
<td>KSPGROPPCG</td>
<td>groppcg</td>
</tr>
<tr>
<td>Pipelined Conjugate Gradients with Residual Replacement</td>
<td>KSPPIPEGRR</td>
<td>pipecgrr</td>
</tr>
<tr>
<td>Conjugate Gradients for the Normal Equations</td>
<td>KSPCGNE</td>
<td>cg</td>
</tr>
<tr>
<td>Flexible Conjugate Gradients [Not00]</td>
<td>KSPFCG</td>
<td>fcg</td>
</tr>
<tr>
<td>Pipelined, Flexible Conjugate Gradients [SSM16]</td>
<td>KSPPIPEFCG</td>
<td>pipef cg</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>KSPType</th>
<th>Options Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjugate Gradients for Least Squares</td>
<td>KSPCGLS</td>
<td>cgls</td>
</tr>
<tr>
<td>Conjugate Gradients with Constraint (1)</td>
<td>KSPNASH</td>
<td>nash</td>
</tr>
<tr>
<td>Conjugate Gradients with Constraint (2)</td>
<td>KSPSTCG</td>
<td>stcg</td>
</tr>
<tr>
<td>Conjugate Gradients with Constraint (3)</td>
<td>KSPGLTR</td>
<td>gltr</td>
</tr>
<tr>
<td>Conjugate Gradients with Constraint (4)</td>
<td>KSPQCG</td>
<td>qcg</td>
</tr>
<tr>
<td>BiConjugate Gradient</td>
<td>KSPBICG</td>
<td>bicg</td>
</tr>
<tr>
<td>BiCGSTAB [vandVorst92]</td>
<td>KSPBCGS</td>
<td>bcgs</td>
</tr>
<tr>
<td>Improved BiCGSTAB</td>
<td>KSPIBCGS</td>
<td>ibcgs</td>
</tr>
<tr>
<td>Flexible BiCGSTAB</td>
<td>KSFBBCGS</td>
<td>fbcgs</td>
</tr>
<tr>
<td>Flexible BiCGSTAB (variant)</td>
<td>KSFBBCGSR</td>
<td>fbcgsr</td>
</tr>
<tr>
<td>Enhanced BiCGSTAB(L)</td>
<td>KSPBCGSL</td>
<td>bcgsl</td>
</tr>
<tr>
<td>Minimal Residual Method [PS75]</td>
<td>KSPMINRES</td>
<td>minres</td>
</tr>
<tr>
<td>Generalized Minimal Residual [SS86]</td>
<td>KSPGMRES</td>
<td>gmres</td>
</tr>
<tr>
<td>Flexible Generalized Minimal Residual [Saa93]</td>
<td>KSPFGMRES</td>
<td>fgmres</td>
</tr>
<tr>
<td>Deflated Generalized Minimal Residual</td>
<td>KSPDGMRES</td>
<td>dgmres</td>
</tr>
<tr>
<td>Pipelined Generalized Minimal Residual [GAMV13]</td>
<td>KSPPGMRES</td>
<td>pgmres</td>
</tr>
<tr>
<td>Pipelined, Flexible Generalized Minimal Residual [SSM16]</td>
<td>KSPPIPEFGMRES</td>
<td>pipefgmres</td>
</tr>
<tr>
<td>Generalized Minimal Residual with Accelerated Restart</td>
<td>KSPLGMRES</td>
<td>lgmres</td>
</tr>
<tr>
<td>Conjugate Residual [EES83]</td>
<td>KSPCR</td>
<td>cr</td>
</tr>
<tr>
<td>Generalized Conjugate Residual</td>
<td>KSPGCR</td>
<td>gcr</td>
</tr>
<tr>
<td>Pipelined Conjugate Residual</td>
<td>KSPPIPECR</td>
<td>pipecr</td>
</tr>
<tr>
<td>Pipelined, Flexible Conjugate Residual [SSM16]</td>
<td>KSPPIPEGCR</td>
<td>pipegcr</td>
</tr>
<tr>
<td>FETI-DP</td>
<td>KSPFETIDP</td>
<td>fetidp</td>
</tr>
<tr>
<td>Conjugate Gradient Squared [Son89]</td>
<td>KSPCGS</td>
<td>cgs</td>
</tr>
<tr>
<td>Transpose-Free Quasi-Minimal Residual (1)</td>
<td>KSPTFQMR</td>
<td>tfqmr</td>
</tr>
<tr>
<td>Transpose-Free Quasi-Minimal Residual (2)</td>
<td>KSPTQMR</td>
<td>tcqmr</td>
</tr>
<tr>
<td>Least Squares Method</td>
<td>KSPLSQR</td>
<td>lsqr</td>
</tr>
<tr>
<td>Symmetric LQ Method [PS75]</td>
<td>KSPSYMMLQ</td>
<td>symmlq</td>
</tr>
<tr>
<td>TSIRMM</td>
<td>KSPTSIRMM</td>
<td>tsirmm</td>
</tr>
<tr>
<td>Python Shell</td>
<td>KSPPYTHON</td>
<td>python</td>
</tr>
<tr>
<td>Shell for no KSP method</td>
<td>KSPPREONLY</td>
<td>preonly</td>
</tr>
</tbody>
</table>

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 constrained 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, \texttt{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, \texttt{rtol}, the absolute size of the residual norm, \texttt{atol}, and the relative increase in the residual, \texttt{dtol}. Convergence is detected at iteration $k$ if

$$
\|r_k\|_2 < \max(rtol \times \|b\|_2, atol),
$$

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

$$
\|r_k\|_2 > dtol \times \|b\|_2.
$$

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

\begin{verbatim}
KSPSetTolerances(KSP ksp,PetscReal rtol,PetscReal atol,PetscReal dtol,PetscInt maxits);
\end{verbatim}

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

\begin{verbatim}
KSPSetConvergenceTest(KSP ksp,PetscErrorCode (*test)(KSP ksp,PetscInt it,PetscReal rnorm,KSPConvergedReason *reason,void *ctx),void *ctx,PetscErrorCode (*destroy)(void *ctx));
\end{verbatim}

The final routine argument, \texttt{ctx}, is an optional context for private data for the user-defined convergence routine, \texttt{test}. Other \texttt{test} routine arguments are the iteration number, \texttt{it}, and the residual's $l_2$ norm, \texttt{rnorm}. The routine for detecting convergence, \texttt{test}, should set \texttt{reason} to positive for convergence, 0 for no convergence, and negative for failure to converge. A full list of possible values for \texttt{KSPConvergedReason} is given in \texttt{include/petscksp.h}. You can use \texttt{KSPGetConvergedReason()} after \texttt{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 \texttt{-ksp_monitor} within the options database. To display the residual norms in a graphical window (running under X Windows), one should use \texttt{-ksp_monitor draw::draw_lg}. Application programmers can also provide their own routines to perform the monitoring by using the command

\begin{verbatim}
KSPMonitorSet(KSP ksp,PetscErrorCode (*mon)(KSP ksp,PetscInt it,PetscReal rnorm,void *ctx,void *ctx,PetscErrorCode (*mondestroy)(void *ctx));
\end{verbatim}

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

Several monitoring routines are supplied with PETSc, including
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 command `-ksp_monitor draw::draw_lg`. 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

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

Then after the KSP solve one calls

```c
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

```c
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

```c
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

```c
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 even require a vector copy.

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

```c
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

```c
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 2.6: PETSc Preconditioners (partial list)

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

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

```c
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

```c
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

```c
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 `lwd`, 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`, `KSTYPE=KSPREONLY`); 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

```c
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.

```c
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, PetscInt N) with the total number of subdomains N that is smaller than the MPI communicator size. In this case PCGASM will coalesce size/N consecutive 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, BAIIJ matrices are not supported. One can use AIJ instead of BAIIJ 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., \(x_1, y_1, z_1, x_2, y_2, \ldots\)).

#### 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()`.

2.3. KSP: Linear System Solvers 67
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 <xx>` 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 coarsen 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 ratio 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 Dirichlet 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

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

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

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

to set this data structure and

```c
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()`).

```c
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 `PCShellSetCon-
text()`.

**Combining Preconditioners**

The PC type `PCCOMPOSITE` allows one to form new preconditioners by combining already-defined precondi-
tioners 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 precondition-
er $B = B_1 + B_2$ can be obtained with

```c
PCSetType(pc,PCCOMPOSITE);
PCompositeAddPCType(pc,type1);
PCompositeAddPCType(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

```c
PCompositeSetType(PC pc,PCompositeType 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

\[
\begin{align*}
    y &= B_1 x \\
    w_1 &= x - Ay \\
    y &= y + B_2 w_1
\end{align*}
\]

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 encapsulated 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:

\[
\begin{align*}
&\text{KSPCreate}(\text{MPI_Comm } \text{comm}, \text{KSP } \ast \text{ksp}); \\
&\text{KSPGetPC}(\text{KSP } \text{ksp}, \text{PC } \ast \text{pc}); \\
&\text{PCSetType}(\text{PC } \text{pc}, \text{PCMG}); \\
&\text{PCMGSetLevels}(\text{pc}, \text{PetscInt } \text{levels}, \text{MPI_Comm } \ast \text{comms});
\end{align*}
\]

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

\[
\text{PCMGSetType}(\text{PC } \text{pc}, \text{PCMGType } \text{mode});
\]

indicates which form of multigrid to apply [SBjorstadG96].

For standard V or W-cycle multigrids, one sets the \text{mode} to be \text{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 \text{PC\_MG\_ADDITIVE} as the \text{mode}. For a variant of full multigrid, one can use \text{PC\_MG\_FULL}, and for the Kaskade algorithm \text{PC\_MG\_KASKADE}. For the multiplicative and full multigrid options, one can use a W-cycle by calling

\[
\text{PCMGSetCycleType}(\text{PC } \text{pc}, \text{PCMGCycleType } \text{ctype});
\]

with a value of \text{PC\_MG\_CYCLE\_W} for \text{ctype}. The commands above can also be set from the options database. The option names are \text{-pc\_mg\_type} [multiplicative, additive, full, kaskade], and \text{-pc\_mg\_cycle\_type}\ <\text{ctype}>

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 \text{-pc\_mg\_distinct\_smoothup} or the routine

\[
\text{PCMGSetDistinctSmoothUp}(\text{PC } \text{pc});
\]

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

\[
\text{PCMGGetCoarseSolve}(\text{PC } \text{pc}, \text{KSP } \ast \text{ksp});
\]

and set the appropriate options in \text{ksp}. Similarly, the smoothers are controlled by first calling

\[
\text{PCMGGetSmotherer}(\text{PC } \text{pc}, \text{PetscInt } \text{level}, \text{KSP } \ast \text{ksp});
\]

and then setting the various options in the \text{ksp}. For example,

\[
\begin{align*}
&\text{PCMGGetSmotherer}(\text{pc}, 1, \ast \text{ksp}); \\
&\text{KSPSetOperators}(\text{ksp}, A_1, A_1);
\end{align*}
\]

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

\[
\begin{align*}
&\text{PCMGGetSmotherer}(\text{pc}, 1, \ast \text{ksp}); \\
&\text{KSPGetPC}(\text{ksp}, \ast \text{pc}); \\
&\text{PCSetType}(\text{pc}, \text{PCSOR});
\end{align*}
\]

\(^2\) See \text{Algebraic Multigrid (AMG) Preconditioners} for information on using algebraic multigrid.
sets SOR as the smoother to use on level 1.

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

```c
PCMGGetSmotherUp(PC pc, PetscInt level, KSP *upksp);
PCMGGetSmotherDown(PC pc, PetscInt level, KSP *downksp);
```

Use

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

and

```c
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:

```c
PCMGSetResidual(PC pc, PetscInt level, PetscErrorCode (*residual)(Mat, Vec, Vec, Vec), Mat u, ...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

```c
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

\[
\begin{pmatrix}
  A_{00,0} & A_{00,1} & A_{00,2} & \ldots & A_{01,0} & A_{01,2} & \ldots \\
  A_{00,1} & A_{01,1} & A_{01,2} & \ldots & A_{01,0} & A_{01,2} & \ldots \\
  A_{00,2} & A_{01,2} & A_{02,2} & \ldots & A_{02,0} & A_{02,2} & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\
  A_{10,0} & A_{10,1} & A_{10,2} & \ldots & A_{11,0} & A_{11,2} & \ldots \\
  A_{10,1} & A_{11,1} & A_{11,2} & \ldots & A_{11,0} & A_{11,2} & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots
\end{pmatrix}
\]

or interlaced, for example with two blocks

\[
\begin{pmatrix}
  A_{00,0} & A_{01,0} & A_{00,1} & A_{01,1} & \ldots \\
  A_{10,0} & A_{11,0} & A_{10,1} & A_{11,1} & \ldots \\
  \vdots & \vdots & \vdots & \vdots & \ddots \\
  A_{00,1} & A_{01,1} & A_{00,2} & A_{01,2} & \ldots \\
  A_{10,1} & A_{11,1} & A_{10,2} & A_{11,2} & \ldots \\
  \vdots & \vdots & \vdots & \vdots & \ddots
\end{pmatrix}
\]

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

The PETSc \texttt{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 \texttt{PCFieldSplitSetFields()} can be called repeatedly to indicate which fields belong to each block. More generally \texttt{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 \texttt{-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

\[
\begin{pmatrix}
  A_{00} & A_{01} \\
  A_{10} & A_{11}
\end{pmatrix}
\]
On occasion the user may provide another matrix that is used to construct parts of the preconditioner
\[
\begin{pmatrix}
A_{p00} & A_{p01} \\
A_{p10} & A_{p11}
\end{pmatrix}
\]

For notational simplicity define \(ksp(A, Ap)\) 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,
\[
\begin{pmatrix}
\text{ksp}(A_{00}, A_{p00}) & 0 \\
0 & \text{ksp}(A_{11}, A_{p11})
\end{pmatrix}
\]

block Gauss-Seidel,
\[
\begin{pmatrix}
I & 0 \\
0 & A_{11}^{-1}
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
-A_{10} & I
\end{pmatrix}
\begin{pmatrix}
A_{00}^{-1} & 0 \\
0 & I
\end{pmatrix}
\]

which is implemented\(^3\) as
\[
\begin{pmatrix}
I & 0 \\
0 & \text{ksp}(A_{11}, A_{p11})
\end{pmatrix}
\begin{pmatrix}
0 & 0 \\
0 & I
\end{pmatrix}
+ \begin{pmatrix}
I & 0 \\
0 & -A_{11}
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
0 & 0
\end{pmatrix}
\begin{pmatrix}
\text{ksp}(A_{00}, A_{p00}) & 0 \\
0 & I
\end{pmatrix}
\]

and symmetric block Gauss-Seidel
\[
\begin{pmatrix}
A_{00}^{-1} & 0 \\
0 & I
\end{pmatrix}
\begin{pmatrix}
I & -A_{01} \\
0 & I
\end{pmatrix}
\begin{pmatrix}
A_{00} & 0 \\
0 & A_{11}^{-1}
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
-A_{10} & I
\end{pmatrix}
\begin{pmatrix}
A_{00}^{-1} & 0 \\
0 & I
\end{pmatrix}
\]

These can be accessed with \(-\text{pc_fieldsplit_type}<\text{additive, multiplicative, symmetric_multiplicative}>\) or the function \(\text{PCFieldSplitSetType}()\). The option prefixes for the internal KSPs are given by \(-\text{fieldsplit_name}_\ldots\).

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

For two by two blocks only there are another family of solvers, based on Schur complements. The inverse of the Schur complement factorization is
\[
\begin{pmatrix}
I & 0 \\
0 & I
\end{pmatrix}
\begin{pmatrix}
A_{00} & 0 \\
0 & S^{-1}
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
0 & I
\end{pmatrix}
\begin{pmatrix}
A_{00}^{-1} & 0 \\
0 & I
\end{pmatrix}
\]

\(^3\) 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.
The preconditioner is accessed with `-pc_fieldsplit_type schur` and is implemented as
\[
\begin{pmatrix}
\text{ksp}(A_{00}, A_{p00}) & 0 \\
0 & I
\end{pmatrix}
\begin{pmatrix}
I & -A_{01} \\
0 & I
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
0 & \text{ksp}(\hat{S}, \hat{Sp})
\end{pmatrix}
\begin{pmatrix}
I \\
-A_{10}\text{ksp}(A_{00}, A_{p00}) & I
\end{pmatrix}.
\]

Where \( \hat{S} = A_{11} - A_{10}\text{ksp}(A_{00}, A_{p00}) A_{01} \) 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
\[
\begin{pmatrix}
\text{ksp}(A_{00}, A_{p00}) & 0 \\
0 & \text{-ksp}(\hat{S}, \hat{Sp})
\end{pmatrix}.
\]

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 \( \hat{Sp} \) for the Schur complement matrix. In general, you are responsible for supplying \( \hat{Sp} \) via `PCFieldSplitSchurPrecondition(pc, PC_FIELDSPLIT_SCHUR_PRE_USER, Sp)`.

In the absence of a good problem-specific \( \hat{Sp} \), 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 all` (default) \( \hat{Sp} = A_{11} \) is used to build a preconditioner for \( \hat{S} \).

Otherwise, `-pc_fieldsplit_schur_precondition self` will set \( \hat{Sp} = \hat{S} \) 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 \( \hat{Sp} \) as much as possible. Specifically, using `-pc_fieldsplit_schur_precondition selfp` will assemble \( \hat{Sp} = A_{11} - A_{10}\text{inv}(A_{00}) A_{01} \).

By default \( \text{inv}(A_{00}) \) 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
\[
\text{ksp}(A_{10} A_{01}, A_{10} A_{01}) A_{10} A_{00} A_{01} \text{ksp}(A_{10} A_{01}, A_{10} A_{01})
\]
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 \( \hat{Sp} \) matrix they provide with

```c
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 solve 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

```c
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

```c
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). 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>
<thead>
<tr>
<th>MatType</th>
<th>PCType</th>
<th>MatSolverType</th>
<th>Package (-pc_factor_mat_solver_type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERESSL</td>
<td>essl</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERLUSOL</td>
<td>lusol</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERMATLAB</td>
<td>matlab</td>
</tr>
<tr>
<td>aij</td>
<td>lu</td>
<td>MATSOLVERMUMPS</td>
<td>mumps</td>
</tr>
<tr>
<td>aij</td>
<td>cholesky</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.7: Options for External Solvers

continues on next page
Table 2.7 – continued from previous page

<table>
<thead>
<tr>
<th>MatType</th>
<th>PCType</th>
<th>MatSolverType</th>
<th>Package (\texttt{pc_factor_mat_solver_type})</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbaij</td>
<td>cholesky</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERSUPERLU</td>
<td>superlu</td>
</tr>
<tr>
<td>aij</td>
<td>lu</td>
<td>MATSOLVERSUPERLU_PERLU_DIST</td>
<td>superlu_dist</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERUMFPACK</td>
<td>umfpack</td>
</tr>
<tr>
<td>seqaij</td>
<td>cholesky</td>
<td>MATSOLVERCHOLMOD</td>
<td>cholmod</td>
</tr>
<tr>
<td>aij</td>
<td>lu</td>
<td>MATSOLVERCLIQUE</td>
<td>clique</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERKLU</td>
<td>klu</td>
</tr>
<tr>
<td>dense</td>
<td>lu</td>
<td>MATSOLVERELEMENTAL</td>
<td>elemental</td>
</tr>
<tr>
<td>dense</td>
<td>cholesky</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>seqaij</td>
<td>lu</td>
<td>MATSOLVERMKL_PARDISO</td>
<td>mkl_pardiso</td>
</tr>
<tr>
<td>aij</td>
<td>lu</td>
<td>MATSOLVERMKL_CPARDISO</td>
<td>mkl_cpardiso</td>
</tr>
<tr>
<td>aij</td>
<td>lu</td>
<td>MATSOLVERPASTIX</td>
<td>pastix</td>
</tr>
<tr>
<td>aijcusparsel</td>
<td>cholesky</td>
<td>MATSOLVERBAS</td>
<td>bas</td>
</tr>
<tr>
<td>aijcusparsel</td>
<td>lu</td>
<td>MATSOLVERCUSPARSE</td>
<td>cusparse</td>
</tr>
<tr>
<td>aij</td>
<td>lu, cholesky</td>
<td>MATSOLVERPETSC</td>
<td>petsc</td>
</tr>
<tr>
<td>baij</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>aijcrl</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>aijperm</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>seqdense</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>aij</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>baij</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>aijcrl</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
Table 2.7 – continued from previous page

<table>
<thead>
<tr>
<th>MatType</th>
<th>PCTYPE</th>
<th>MatSolverType</th>
<th>Package (-pc_factor_mat_solver_type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aijperm</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>seqdense</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

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`:

```c
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

\[
F(x) = 0,
\]

where \( F : \mathbb{R}^n \to \mathbb{R}^n \). 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

\[
x_{k+1} = x_k - J(x_k)^{-1}F(x_k), \quad k = 0, 1, \ldots,
\]

where \( x_0 \) is an initial approximation to the solution and \( J(x_k) = F'(x_k) \), the Jacobian, is nonsingular at each iteration. In practice, the Newton iteration (2.4) is implemented by the following two steps:

1. (Approximately) solve \( J(x_k)\Delta x_k = -F(x_k) \).
2. Update \( x_{k+1} = x_k + \Delta x_k \).

Other defect-correction algorithms can be implemented by using different choices for \( J(x_k) \).

### 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**

```c
static char help[] = "Newton's method for a two-variable system, sequential.\n\n";

/*
Concepts: SNES^basic example
*/

/* 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
   petsciss.h - index sets           petscksp.h - Krylov subspace methods
   petscviewer.h - viewers          petscpc.h - preconditioners
   petscksp.h - linear solvers      */
```

(continues on next page)
This example solves either
\begin{equation}
F\{x_0,x_1\} = \{x_0^2 + x_0 x_1 - 3; x_0 x_1 - 1 + x_0^2 - 6\}
\end{equation}
or if the \texttt{-hard} option is given
\begin{equation}
F\{x_0,x_1\} = \{\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 */
```c
/*
* 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 = SNESGetFunction(snes,r,FormFunction1,NULL);CHKERRQ(ierr);
*   /*
*   * Set Jacobian matrix data structure and Jacobian evaluation routine
*   */
*   ierr = SNESGetJacobian(snes,J,J,FormJacobian1,NULL);CHKERRQ(ierr);
*  } else {
*   ierr = SNESGetFunction(snes,r,FormFunction2,NULL);CHKERRQ(ierr);
*   ierr = SNESGetJacobian(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 = SNESGetFromOptions(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
*  */
*/```
for the nonlinear solver prior to calling SNESolve(). In particular,
to employ an initial guess of zero, the user should explicitly set
this vector to zero by calling VecSet().
*/

ierr = SNESolve(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);
ifilex = VecDestroy(&r); CHKERRQ(ierr);
ifilex = MatDestroy(&J); CHKERRQ(ierr);
ifilex = SNESDestroy(&snes); CHKERRQ(ierr);

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;
}/* continued from next 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];
  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;
}
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;
    ierr = MatSetValues(B, 2, idx, 2, idx, A, INSERT_VALUES); CHKERRQ(ierr);

    /* Restore vector */
}
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

```
SNESolve(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 residual function \((2.3)\), which is set using

\[
\text{SNESSetFunction(SNES \text{ snes}, \text{Vec } f, \text{PetscErrorCode } (*\text{FormFunction})(\text{SNES } \text{ snes}, \text{Vec } x, \text{Vec } f, \_\text{void } *\text{ctx}), \text{void } *\text{ctx});}
\]

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

\[
\text{SNESSetJacobian(SNES \text{ snes}, \text{Mat } Amat, \text{Mat } Pmat, \text{PetscErrorCode } (*\text{FormJacobian})(\text{SNES } \text{ snes}, \_\text{Vec } x, \text{Mat } A, \text{Mat } B, \text{void } *\text{ctx}), \text{void } *\text{ctx});}
\]

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 \(-\text{snes}_\text{mf}_\text{operator}\) 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), \(-\text{snes}_\text{mf}_\text{operator}\) 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

\[
F(x) := A(x)x - b = 0
\]

where \(A(x)\) 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 \(-\text{snes}_\text{mf}_\text{operator}\) with this approximation to the Jacobian, the Picard
iterative procedure can be performed by defining $J(x)$ to be $A(x)$. 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>
<thead>
<tr>
<th>Method</th>
<th>SNESType</th>
<th>Options Name</th>
<th>Default Line Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Search Newton</td>
<td>SNESNEWTONLS</td>
<td>newtonls</td>
<td>SNESLINESEARCHBT</td>
</tr>
<tr>
<td>Trust region Newton</td>
<td>SNESNEWTONTR</td>
<td>newtontr</td>
<td>—</td>
</tr>
<tr>
<td>Nonlinear Richardson</td>
<td>SNESNRICHARDSON</td>
<td>nrichardson</td>
<td>SNESLINESEARCHL2</td>
</tr>
<tr>
<td>Nonlinear CG</td>
<td>SNESNCG</td>
<td>nCG</td>
<td>SNESLINESEARCHCP</td>
</tr>
<tr>
<td>Nonlinear GMRES</td>
<td>SNESNGMRES</td>
<td>ngmres</td>
<td>SNESLINESEARCHL2</td>
</tr>
<tr>
<td>Quasi-Newton</td>
<td>SNESQN</td>
<td>qn</td>
<td>see PETSc quasi-Newton solvers</td>
</tr>
<tr>
<td>Full Approximation Scheme</td>
<td>SNESFAS</td>
<td>fas</td>
<td>—</td>
</tr>
<tr>
<td>Nonlinear ASM</td>
<td>SNESNASM</td>
<td>nasm</td>
<td>—</td>
</tr>
<tr>
<td>ASPIN</td>
<td>SNESASPIN</td>
<td>aspin</td>
<td>SNESLINESEARCHBT</td>
</tr>
<tr>
<td>Nonlinear Gauss-Seidel</td>
<td>SNESNGS</td>
<td>ngs</td>
<td>—</td>
</tr>
<tr>
<td>Anderson Mixing</td>
<td>SNESANDERSON</td>
<td>anderson</td>
<td>—</td>
</tr>
<tr>
<td>Newton with constraints (1)</td>
<td>SNESVINEWTONRSLS</td>
<td>vinewtonrsls</td>
<td>SNESLINESEARCHBT</td>
</tr>
<tr>
<td>Newton with constraints (2)</td>
<td>SNESVINEWTONSSLS</td>
<td>vinewtonssl</td>
<td>SNESLINESEARCHBT</td>
</tr>
<tr>
<td>Multi-stage Smoothers</td>
<td>SNESMS</td>
<td>ms</td>
<td>—</td>
</tr>
<tr>
<td>Composite</td>
<td>SNESCOMPOSITE</td>
<td>composite</td>
<td>—</td>
</tr>
<tr>
<td>Linear solve only</td>
<td>SNESKSPONLY</td>
<td>ksponly</td>
<td>—</td>
</tr>
<tr>
<td>Python Shell</td>
<td>SNESPYTHON</td>
<td>python</td>
<td>—</td>
</tr>
<tr>
<td>Shell (user-defined)</td>
<td>SNESHELL</td>
<td>shell</td>
<td>—</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Line Search</th>
<th>SNESLineSearchType</th>
<th>Options Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtracking</td>
<td>SNESLINESEARCHBT</td>
<td>bt</td>
</tr>
<tr>
<td>(damped) step</td>
<td>SNESLINESEARCHBASIC</td>
<td>basic</td>
</tr>
<tr>
<td>L2-norm Minimization</td>
<td>SNESLINESEARCHL2</td>
<td>l2</td>
</tr>
<tr>
<td>Critical point</td>
<td>SNESLINESEARCHCP</td>
<td>cp</td>
</tr>
<tr>
<td>Shell</td>
<td>SNESLINESEARCHSHELL</td>
<td>shell</td>
</tr>
</tbody>
</table>

Every SNES has a line search context of type `SNESLineSearch` that may be retrieved using

```c
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

```c
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

```c
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

```c
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 \( \Delta_0 \) 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

\[ x_{k+1} = x_k - \lambda F(x_k), \quad k = 0, 1, \ldots, \]

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

\[ \text{SNESNCGSetType(SNES \text{ snes}, \text{ SNESNCGType \text{ btype}});} \]

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>
<thead>
<tr>
<th>QN Method</th>
<th>SNESQNType</th>
<th>Options Name</th>
<th>Default Line Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-BFGS</td>
<td>SNES_QN_LBFGS</td>
<td>lbfgs</td>
<td>SNESLINESEARCHCP</td>
</tr>
<tr>
<td>“Good” Broyden</td>
<td>SNES_QN_BROYDEN</td>
<td>broyden</td>
<td>SNESLINESEARCHBASIC</td>
</tr>
<tr>
<td>“Bad” Broyden</td>
<td>SNES_QN_BADBROYEN</td>
<td>badbroyden</td>
<td>SNESLINESEARCHL2</td>
</tr>
</tbody>
</table>

One may also control the form of the initial Jacobian approximation with
and the restart type with

```c
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

```c
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

```c
SNESFASSetType(SNES snes, SNESFASType fastype);
```

and the cycle type, 1 for V, 2 for W, may be set with

```c
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

```c
SNESFASGetSmoothers(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmoothersUp(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmoothersDown(SNES snes, PetscInt level, SNES *smooth);
```

and the level cycles with

```c
SNESFASGetCycleSNES(SNES snes, PetscInt level, SNES *lsnes);
```

Also akin to PCMG, the restriction and prolongation at a level may be acquired with

```c
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

```c
SNESFASGetInjection(SNES snes, PetscInt level, Mat *mat);
```

The coarse solve context may be acquired with

```c
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.

```c
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 DM DA 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

```c
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:

```c
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().

2.4. SNES: Nonlinear Solvers 93
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 \texttt{-snes_trtol <trtol>} or with the routine 

\begin{verbatim}
SNESSetTrustRegionTolerance(SNES snes, PetscReal trtol);
\end{verbatim}

Users can set their own customized convergence tests in \textsc{SNES} by using the command 

\begin{verbatim}
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));
\end{verbatim}

The final argument of the convergence test routine, \texttt{cctx}, denotes an optional user-defined context for private data. When solving systems of nonlinear equations, the arguments \texttt{xnorm}, \texttt{gnorm}, and \texttt{f} are the current iterate norm, current step norm, and function norm, respectively. \texttt{SNESConvergedReason} should be set positive for convergence and negative for divergence. See \texttt{include/petscsnes.h} for a list of values for \texttt{SNESConvergedReason}.

### Convergence Monitoring

By default the \textsc{SNES} solvers run silently without displaying information about the iterations. The user can initiate monitoring with the command 

\begin{verbatim}
SNESMonitorSet(SNES snes, PetscErrorCode (*mon)(SNES snes, PetscInt its, PetscReal norm, void *, mctx), void *mctx, PetscErrorCode (*monitordestroy)(void **));
\end{verbatim}

The routine, \texttt{mon}, indicates a user-defined monitoring routine, where \texttt{its} and \texttt{mctx} respectively denote the iteration number and an optional user-defined context for private data for the monitor routine. The argument \texttt{norm} is the function norm.

The routine set by \texttt{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 \texttt{-snes_monitor} activates the default \textsc{SNES} monitor routine, \texttt{SNESMonitorDefault()}, while \texttt{-snes_monitor/lg_residualnorm} draws a simple line graph of the residual norm’s convergence.

One can cancel hardwired monitoring routines for \textsc{SNES} at runtime with \texttt{-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 \texttt{-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 \texttt{-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 

\begin{verbatim}
SNESGetSolution(SNES snes, Vec *x);
SNESGetFunction(SNES snes, Vec *r, void *ctx, int(**func)(SNES, Vec, Vec, void*));
\end{verbatim}

return the solution vector and function vector from a \textsc{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 Jacobian 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 \( \|x_k - x_l\| \) 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

\[
r_k^{(i)} = F'(x_k)\Delta x_k + F(x_k)
\]
satisfy

\[ \frac{\| r_k^{(i)} \|}{\| F(x_k) \|} \leq \eta_k \leq \eta < 1. \]

Here \( x_0 \) is an initial approximation of the solution, and \( \| \cdot \| \) denotes an arbitrary norm in \( \mathbb{R}^n \).

By default a constant relative convergence tolerance is used for solving the subsidiary linear systems within the Newton-like methods of \textbf{SNES}. When solving a system of nonlinear equations, one can instead employ the techniques of Eisenstat and Walker \cite{EW96} to compute \( \eta_k \) at each step of the nonlinear solver by using the option \texttt{-snes_ksp_ew}. In addition, by adding one's own KSP convergence test (see \textit{Convergence Tests}), one can easily create one's own, problem-dependent, inner convergence tests.

### 2.4.5 Matrix-Free Methods

The \textbf{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 \textit{only} when the linear systems are solved by an iterative method in combination with no preconditioning (\texttt{PCNONE} or \texttt{-pc_type none}), a user-provided preconditioner matrix, or a user-provided preconditioner shell (\texttt{PCSHELL}, discussed in \textit{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 \textbf{SNES} with the routine

```c
MatCreateSNESMF(SNES snes,Mat *mat);
```

This routine creates the data structures needed for the matrix-vector products that arise within Krylov space iterative methods \cite{BS90} by employing the matrix type \texttt{MAT SHELL}, discussed in \textit{Matrix-Free Matrices}. The default \textbf{SNES} matrix-free approximations can also be invoked with the command \texttt{-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 \texttt{-snes_mf_operator}.

See also

```c
MatCreateMFFD(Vec x, Mat *mat);
```

for users who need a matrix-free matrix but are not using \textbf{SNES}.

The user can set one parameter to control the Jacobian-vector product approximation with the command

```c
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 \texttt{-snes_mf_err <err>}

In addition, \textbf{SNES} provides a way to register new routines to compute the differencing parameter (\( h \)); see the manual page for \texttt{MatMFFDSetType()} and \texttt{MatMFFDRegister()}. We currently provide two default routines accessible via \texttt{-snes_mf_type <default or wp>}. For the default approach there is one “tuning” parameter, set with

```c
MatMFFDDSSetUmin(Mat mat,PetscReal umin);
```
This parameter, \( u_{\text{min}} \) (or \( u_{\text{min}} \)), is a bit involved; its default is \( 10^{-6} \). The Jacobian-vector product is approximated via the formula

\[
F'(u)a \approx \frac{F(u + h \cdot a) - F(u)}{h}
\]

where \( h \) is computed via

\[
h = e_{\text{rel}} \cdot \begin{cases} u^T a / \| a \|_2^2 & \text{if } |u^T a| > u_{\text{min}} \| a \|_1 \\ u_{\text{min}} & \text{otherwise} \end{cases}
\]

This approach is taken from Brown and Saad [BS90]. The parameter can also be set from the options database with \texttt{-snes_mf_umin <umin>}

The second approach, taken from Walker and Pernice, [PW98], computes \( h \) via

\[
h = \sqrt{1 + \| u \| e_{\text{rel}}} / \| a \|_2
\]

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

\[
\text{MatMFFDWPSetComputeNormU(Mat mat, PetscBool);} \]

or \texttt{-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

\[
\text{MatMFFDSetHHistory(Mat, PetscScalar *, int);} \\
\text{MatMFFDResetHHistory(Mat, PetscScalar *, int);} \\
\text{MatMFFDGetH(Mat, PetscScalar *);} \\
\]

We include an explicit example of using matrix-free methods in \texttt{ex3.c}. Note that by using the option \texttt{-snes_mf} one can easily convert any \texttt{SNES} code to use a matrix-free Newton-Krylov method without a preconditioner. As shown in this example, \texttt{SNESSetFromOptions()} must be called after \texttt{SNESSetJacobian()} to enable runtime switching between the user-specified Jacobian and the default \texttt{SNES} matrix-free form.

Listing: \texttt{src/snes/tutorials/ex3.c}

```c
static char help[] = "Newton methods to solve \( u'' + u^2 \) = f in parallel.
\nThis example employs a user-defined monitoring routine and optionally a user-defined
\nroutine to check candidate iterates produced by line search routines.\n\nThe command line options include:\n\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)
```

2.4. SNES: Nonlinear Solvers 97
Include "petscdm.h" so that we can use data management objects (DMs)
Include "petscdmda.h" so that we can use distributed arrays (DMDAs).
Include "petcsnes.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 <petcsnes.h>

/* User-defined routines. */
PetscErrorCode FormJacobian(SNES, Vec, Mat, Mat, void *
PetscErrorCode FormFunction(SNES, Vec, Vec, void *
PetscErrorCode FormInitialGuess(Vec);
PetscErrorCode PreCheck(SNESLineSearch, Vec, Vec, PetscBool *
PetscErrorCode PostCheck(SNESLineSearch, Vec, Vec, PetscBool *
PetscErrorCode PostSetSubKSP(SNESLineSearch, Vec, Vec, PetscBool *
PetscErrorCode MatrixFreePreconditioner(PC, Vec, Vec);

/* User-defined context for monitoring */
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 checking candidate iterates that are...
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 previous 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; /* step-checking context */
    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   *
    */
    ierr = DMDACreate1d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,N,1,1,NULL,&ctx.da); CHKERRQ(ierr);
```c
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 = SNESetFunction(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 = SNESetJacobian(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)
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 = SNESetFromOptions(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);
}
PETSc/TAO Users Manual, Release 3.15.3

(continued from previous page)

```c
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
", (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 SNESolve(). In particular,
```

(continues on next page)
to employ an initial guess of zero, the user should explicitly set this vector to zero by calling VecSet().

```c
ierr = FormInitialGuess(x);CHKERRQ(ierr);
```
```c
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 SNESetFunction()

 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);CHKERRQ(ierr);
```

(continues on next 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];
   xE++;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);

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) { /* snes */
        snes->formJacobian = FormJacobian;
        snes->ctx = ctx;
        if (user->sJerr) {
            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);
    } /* snes */
    } /* formJacobian */
    } /* PetscErrorCode FormJacobian */
ierr = SNESSetJacobianDomainError(snes); CHKERRQ(ierr);
PetscFunctionReturn(0);
}
/* Get pointer to vector data */
ierr = DMMDAVecGetArrayRead(da,x,&xx); CHKERRQ(ierr);
ierr = DMDAGetCorners(da,&xs,NULL,NULL,&xm,NULL,NULL,NULL); CHKERRQ(ierr);
/* Get range of locally owned matrix */
ierr = DMDAGetInfo(da,NULL,&M,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;
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)
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
fnorm - 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, 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);
}
/* ------------------------------------------------------------------- */
/*
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) */
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 = DMDDAVecRestoreArray(da, check->last_step, &xa_last); CHKERRQ(ierr);

ierr = VecCopy(x, check->last_step); CHKERRQ(ierr);
PetscFunctionReturn(0);

/* PostSetSubKSP - Optional user-defined routine that reset SubKSP options when a
   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 gmrres -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;

    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)
PETSc/TAO Users Manual, Release 3.15.3

(continued from previous page)

```c
ierr = KSPGetPC(ksp,&pc); CHKERRQ(ierr);
ierr = PCB JacobiGetSubKSP(pc,NULL,NULL,&sub_ksp); CHKERRQ(ierr);
sub_ksp = sub_ksp[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, subksp_it %D
                        ..."); 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 2.11: Jacobian Options

<table>
<thead>
<tr>
<th>Matrix Use</th>
<th>Conventional Matrix Formats</th>
<th>Matrix-free versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobian Matrix</td>
<td>Create matrix with MatCreate(). Assemble matrix with user-defined routine †</td>
<td>Create matrix with MatCreateShell(). Use MatShellSetOperation() to set various matrix actions, or use MatCreateMFFD() or MatCreateSNESMF().</td>
</tr>
<tr>
<td>Preconditioning Matrix</td>
<td>Create matrix with MatCreate(). Assemble matrix with user-defined routine †</td>
<td>Use SNESGetKSP() and KSPGetPC() to access the PC, then use PCSetType(pc, PCSHELL) followed by PCShellSetApply().</td>
</tr>
</tbody>
</table>

* Use either the generic MatCreate() or a format-specific variant such as MatCreateAIJ().
† Set user-defined matrix formation routine with SNESetJacobian() 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.

```c
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)
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MatColoringType</th>
<th>-mat_coloring_type</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallest-last [MoreSGH84]</td>
<td>MATCOLORINGSL</td>
<td>sl</td>
<td>No</td>
</tr>
<tr>
<td>largest-first [MoreSGH84]</td>
<td>MATCOLORINGLF</td>
<td>lf</td>
<td>No</td>
</tr>
<tr>
<td>incidence-degree [MoreSGH84]</td>
<td>MATCOLORINGID</td>
<td>id</td>
<td>No</td>
</tr>
<tr>
<td>Jones-Plassmann [JP93]</td>
<td>MATCOLORINGJP</td>
<td>jp</td>
<td>Yes</td>
</tr>
<tr>
<td>Greedy</td>
<td>MATCOLORINGGREEDY</td>
<td>greedy</td>
<td>Yes</td>
</tr>
<tr>
<td>Natural (1 color per column)</td>
<td>MATCOLORINGNATURAL</td>
<td>natural</td>
<td>Yes</td>
</tr>
<tr>
<td>Power ($A^k$ followed by 1-coloring)</td>
<td>MATCOLORINGPOWER</td>
<td>power</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

$$F'_i \approx \frac{F(u + h \cdot dx_i) - F(u)}{h}$$
where \( h \) is computed via:

\[
h = e_{rel} \cdot \begin{cases} u_i & \text{if } |u_i| > u_{\text{min}} \\ u_{\text{min}} \cdot \text{sign}(u_i) & \text{otherwise.} \end{cases}
\]

for MATMFFD_DS or:

\[
h = e_{rel} \sqrt{\|u\|}
\]

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().

```c
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: \( L u_i \leq u_i \leq H u_i \). 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

```c
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

```c
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 \( \mathbf{M} \):

\[
\mathbf{F}_M(x) = \mathbf{M}(x, b) - x.
\]
Right nonlinear preconditioning redefines the nonlinear function as the function on the action of the nonlinear preconditioner:

\[ F(M(x, b)) = b, \]

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

\[ \text{SNESCompositeSetType(SNES,SNESCompositeType);} \]

New subsolvers may be added to the composite solver with

\[ \text{SNESCompositeAddSNES(SNES,SNESType);} \]

and accessed with

\[ \text{SNESCompositeGetSNES(SNES,PetscInt,SNES *} \]; \]

### 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

\[ (u_i)_t = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \]

\[ u_t = \dot{A}u; \]

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

\[ u^n_i = \xi_i(t_n) \doteq u(x_i,t_n), \]
\[ \xi'(t_{n+1}) = \frac{u^{n+1} - u^n}{\Delta t} \]

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). 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** \( F(t, u, \dot{u}) \)

```c
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 \( \text{funP} \). 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

```c
TSSetRHSFunction(TS ts, Vec R, PetscErrorCode (*f)(TS, PetscReal, Vec, Vec, void*), void_u, void_r);
```

- **Jacobian** \( \sigma F_u(t^n, u^n, \dot{u}^n) + F_u(t^n, u^n, \dot{u}^n) \)

If using a fully implicit or semi-implicit (IMEX) method one also can provide an appropriate (approximate) Jacobian matrix of \( F() \).

```c
TSSetIJacobian(TS ts, Mat A, Mat B, PetscErrorCode (*fjac)(TS, PetscReal, Vec, Vec, Mat, Mat, void*), void *jacP);
```

The arguments for the function \( \text{jfac}() \) are the timestep context, current time, input state \( u \), input derivative \( \dot{u} \), input shift \( \sigma \), matrix \( A \), preconditioning matrix \( B \), and the (optional) user-provided context \( \text{jacP} \).

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 \( \sigma \) depends on the ODE integrator and time step but not on the function being integrated. Thus

\[ \frac{dF}{du^n} = \sigma F_u(t^n, u^n, \dot{u}^n) + F_u(t^n, u^n, \dot{u}^n). \]

This explains why the user provides Jacobian is in the given form for all integration methods. An equivalent way to derive the formula is to note that

\[ F(t^n, u^n, \dot{u}^n) = F(t^n, u^n, w + \sigma * u^n) \]

\footnote{If the matrix \( F_u(t) = \partial F/\partial \dot{u} \) 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.}

2.5. TS: Scalable ODE and DAE Solvers 115
where \( w \) is some linear combination of previous time solutions of \( u \) so that

\[
\frac{dF}{du^n} = \sigma F_u(t^n, u^n, \dot{u}^n) + F_u(t^n, u^n, \dot{u}^n)
\]

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 \( \frac{\partial u}{\partial u} u^n = 1/\delta t \) resulting in

\[
\frac{dF}{du^n} = (1/\delta t)F_u + F_u(t^n, u^n, \dot{u}^n)
\]

But \( F_u = 1 \), in this special case, resulting in the expected Jacobian \( I/\delta t - f_u(t, u^n) \).

\begin{itemize}
  \item Jacobian \( G_u \)
  \begin{itemize}
  \item If using a fully implicit method and the function \( G() \) is provided, one also can provide an appropriate (approximate) Jacobian matrix of \( G() \).
  \end{itemize}
\end{itemize}

\begin{Verbatim}
TSSetRHSJacobian(TS ts,Mat A,Mat B, 
PetscErrorCode (*fjac)(TS,PetscReal,Vec,Mat,Mat, 
void *),void *jacP);
\end{Verbatim}

The arguments for the function \( \text{fjac}() \) are the timestep context, current time, input state \( u \), matrix \( A \), preconditioning matrix \( B \), and the (optional) user-provided context \( \text{jacP} \).

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

\begin{Verbatim}
int TSCreate(MPI_Comm comm,TS *ts);
\end{Verbatim}

\begin{Verbatim}
int TSSetProblemType(TS ts,TSPreproblemType problemtype);
\end{Verbatim}

The \( \text{TSPreproblemType} \) is one of \( \text{TS_LINEAR} \) or \( \text{TS_NONLINEAR} \).

To set up \( TS \) for solving an ODE, one must set the “initial conditions” for the ODE with

\begin{Verbatim}
TSSetSolution(TS ts,Vec initialsolution);
\end{Verbatim}

One can set the solution method with the routine

\begin{Verbatim}
TSetType(TS ts,TSType type);
\end{Verbatim}

Currently supported types are \( \text{TSEULER}, \text{TSRK} \) (Runge-Kutta), \( \text{TSBEULER}, \text{TSCN} \) (Crank-Nicolson), \( \text{TSTHETA}, \text{TSGLE} \) (generalized linear), \( \text{TSPEUDO} \), and \( \text{TSSUNDIALS} \) (only if the Sundials package is installed), or the command line option

\begin{itemize}
  \item \text{-ts_type euler,rk,beuler,cn,theta,gl,peudo,sundials,eimex,arikimex,rosw.}
\end{itemize}

A list of available methods is given in the following table.
### Table 2.12: Time integration schemes

<table>
<thead>
<tr>
<th>TS Name</th>
<th>Reference</th>
<th>Class</th>
<th>Type</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler</td>
<td></td>
<td>one-step</td>
<td>explicit</td>
<td>1</td>
</tr>
<tr>
<td>ssp</td>
<td>multistage SSP [Ket08]</td>
<td>Runge-Kutta</td>
<td>explicit</td>
<td>≤ 4</td>
</tr>
<tr>
<td>rk*</td>
<td>multiscale</td>
<td>Runge-Kutta</td>
<td>explicit</td>
<td>≥ 1</td>
</tr>
<tr>
<td>beuler</td>
<td>backward Euler</td>
<td>one-step</td>
<td>implicit</td>
<td>1</td>
</tr>
<tr>
<td>cn</td>
<td>Crank-Nicolson</td>
<td>one-step</td>
<td>implicit</td>
<td>2</td>
</tr>
<tr>
<td>theta*</td>
<td>theta-method</td>
<td>one-step</td>
<td>implicit</td>
<td>≤ 2</td>
</tr>
<tr>
<td>alpha</td>
<td>alpha-method [JWH00]</td>
<td>one-step</td>
<td>implicit</td>
<td>2</td>
</tr>
<tr>
<td>gl</td>
<td>general linear [BJW07]</td>
<td>multistep-multistage</td>
<td>implicit</td>
<td>≤ 3</td>
</tr>
<tr>
<td>eimex</td>
<td>extrapolated IMEX [CS10]</td>
<td>one-step</td>
<td>≥ 1, adaptive</td>
<td></td>
</tr>
<tr>
<td>arkimex</td>
<td>See IMEX Runge-Kutta schemes</td>
<td>IMEX Runge-Kutta</td>
<td>IMEX</td>
<td>1 – 5</td>
</tr>
<tr>
<td>rosw</td>
<td>See Rosenbrock W-schemes</td>
<td>Rosenbrock-W</td>
<td>linearly implicit</td>
<td>1 – 4</td>
</tr>
<tr>
<td>glee</td>
<td>See GL schemes with global error estimation</td>
<td>GL with global error</td>
<td>explicit and implicit</td>
<td>1 – 3</td>
</tr>
</tbody>
</table>

Set the initial time with the command

```c
TSSetTime(TS ts, PetscReal time);
```

One can change the timestep with the command

```c
TSSetTimeStep(TS ts, PetscReal dt);
```

can determine the current timestep with the routine

```c
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

```c
TSSetMaxSteps(TS ts, PetscInt maxsteps);
TSSetMaxTime(TS ts, PetscReal maxtime);
```

and determines the behavior near the final time with

```c
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

```c
TSSolve(TS ts, Vec U);
```

The solve call implicitly sets up the timestep context; this can be done explicitly with

```c
TSSetUp(TS ts);
```

One destroys the context with

```c
TSDestroy(TS *ts);
```
and views it with

```c
TSView(TS ts, PetscViewer viewer);
```

In place of `TSSolve()`, a single step can be taken using

```c
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

$$\dot{u} = f(t, u, z)$$

$$0 = h(t, u, z)$$

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

$$\dot{u} = f(t, u, z)$$

$$0 = h(t, u)$$

Notice that the constraint equation $h$ is not a function of the constraint variable $z$. 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 an 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,

$$0 = \dot{h}(t, u)$$

$$= \frac{dh}{du} \dot{u} + \frac{\partial h}{\partial t}$$

$$= \frac{dh}{du} f(t, u, z) + \frac{\partial h}{\partial t}$$

If the Jacobian $\frac{dh}{du}$ 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 $F_u + \sigma F_u$ 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 2.13 and a list of methods with a summary of their properties is given in IMEX Runge-Kutta schemes.
Table 2.13: Use case examples for **TSARKIMEX**

| \( \dot{u} = g(t, u) \) | 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 | \( F(t, u, \dot{u}) = M\dot{u} - f(t, u) \)  
|                           |              | \( G(t, u) = M^{-1}g(t, u) \) |
| \( \dot{u} = f(t, u, z) + g(t, u, z) \) | 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 **TS-SetEquationType()** to **TS_EQ_IMPLICIT** or higher.

Table 2.14 lists 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 2.14: IMEX Runge-Kutta schemes

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Stages (IM)</th>
<th>Order (Stage)</th>
<th>IM</th>
<th>SA</th>
<th>Embed</th>
<th>DO</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>based on CN</td>
<td>2 (1)</td>
<td>2 (2)</td>
<td>A-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td></td>
</tr>
<tr>
<td>l2</td>
<td>SSP2(2,2,2) [PR05]</td>
<td>2 (2)</td>
<td>2 (1)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td>SSP SDIRK</td>
</tr>
<tr>
<td>ars122</td>
<td>ARS122 [ARS97]</td>
<td>2 (1)</td>
<td>3 (1)</td>
<td>A-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td></td>
</tr>
<tr>
<td>2c</td>
<td>[GKC13]</td>
<td>3 (2)</td>
<td>2 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td>SDIRK</td>
</tr>
<tr>
<td>2d</td>
<td>[GKC13]</td>
<td>3 (2)</td>
<td>2 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td>SDIRK</td>
</tr>
<tr>
<td>2e</td>
<td>[GKC13]</td>
<td>3 (2)</td>
<td>2 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (1)</td>
<td>yes (2)</td>
<td>SDIRK</td>
</tr>
<tr>
<td>prss2</td>
<td>PRS(3,3,2) [PR05]</td>
<td>3 (3)</td>
<td>3 (1)</td>
<td>L-Stable</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>SSP</td>
</tr>
<tr>
<td>3</td>
<td>[KC03]</td>
<td>4 (3)</td>
<td>3 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (2)</td>
<td>yes (2)</td>
<td>SDIRK</td>
</tr>
<tr>
<td>bpr3</td>
<td>[BPR11]</td>
<td>5 (4)</td>
<td>3 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>SDIRK</td>
</tr>
<tr>
<td>ars443</td>
<td>ARS97</td>
<td>5 (4)</td>
<td>3 (1)</td>
<td>L-Stable</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>SDIRK</td>
</tr>
<tr>
<td>4</td>
<td>[KC03]</td>
<td>6 (5)</td>
<td>4 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (3)</td>
<td>yes</td>
<td>SDIRK</td>
</tr>
<tr>
<td>5</td>
<td>[KC03]</td>
<td>8 (7)</td>
<td>5 (2)</td>
<td>L-Stable</td>
<td>yes</td>
<td>yes (4)</td>
<td>yes (3)</td>
<td>SDIRK</td>
</tr>
</tbody>
</table>

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 2.15 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 2.15: Rosenbrock W-schemes

<table>
<thead>
<tr>
<th>TS</th>
<th>Reference</th>
<th>Stages (IM)</th>
<th>Order (Stage)</th>
<th>IM</th>
<th>SA</th>
<th>Embed</th>
<th>DO</th>
<th>-W</th>
<th>PDAE</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>theta1</td>
<td>classical</td>
<td>1(1)</td>
<td>1(1)</td>
<td>L-Stable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>theta2</td>
<td>classical</td>
<td>1(1)</td>
<td>2(2)</td>
<td>A-Stable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2m</td>
<td>Zoltan</td>
<td>2(2)</td>
<td>2(1)</td>
<td>L-Stable</td>
<td>No</td>
<td>Yes(1)</td>
<td>Yes(2)</td>
<td>Yes</td>
<td>No</td>
<td>SSP</td>
</tr>
<tr>
<td>2p</td>
<td>Zoltan</td>
<td>2(2)</td>
<td>2(1)</td>
<td>L-Stable</td>
<td>No</td>
<td>Yes(1)</td>
<td>Yes(2)</td>
<td>Yes</td>
<td>No</td>
<td>SSP</td>
</tr>
<tr>
<td>ra3pw</td>
<td></td>
<td>3(3)</td>
<td>3(1)</td>
<td>A-Stable</td>
<td>No</td>
<td>Yes</td>
<td>Yes(2)</td>
<td>No</td>
<td>Yes(3)</td>
<td></td>
</tr>
<tr>
<td>ra34pw2</td>
<td></td>
<td>4(4)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes(3)</td>
<td>Yes</td>
<td>Yes(3)</td>
<td></td>
</tr>
<tr>
<td>rodas3</td>
<td>[SVB+97]</td>
<td>4(4)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>sandu3</td>
<td>[SVB+97]</td>
<td>3(3)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes(2)</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>assp3p3</td>
<td>un-pub.</td>
<td>3(2)</td>
<td>3(1)</td>
<td>A-Stable</td>
<td>No</td>
<td>Yes</td>
<td>Yes(2)</td>
<td>Yes</td>
<td>No</td>
<td>SSP</td>
</tr>
<tr>
<td>lassp3p3</td>
<td>un-pub.</td>
<td>4(3)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>No</td>
<td>Yes</td>
<td>Yes(3)</td>
<td>Yes</td>
<td>No</td>
<td>SSP</td>
</tr>
<tr>
<td>lassp3p4</td>
<td>un-pub.</td>
<td>4(3)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>No</td>
<td>Yes</td>
<td>Yes(3)</td>
<td>Yes</td>
<td>No</td>
<td>SSP</td>
</tr>
<tr>
<td>ark3</td>
<td>un-pub.</td>
<td>4(3)</td>
<td>3(1)</td>
<td>L-Stable</td>
<td>No</td>
<td>Yes</td>
<td>Yes(3)</td>
<td>Yes</td>
<td>No</td>
<td>IMEX-RK</td>
</tr>
</tbody>
</table>

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 \([y, \varepsilon]\), where \(y\) is the solution, \(\tilde{y}\) is the “auxiliary solution,” and \(\varepsilon\) is the error. The working vector that TSGLEE uses is \(Y = [y, \tilde{y}]\), or \([y, \varepsilon]\). A GLEE method is defined by

- \((p, r, s)\): (order, steps, and stages),
- \(\gamma\): factor representing the global error ratio,
- \(A, U, V\): method coefficients,
- \(S\): starting method to compute the working vector from the solution (say at the beginning of time integration) so that \(Y = S y\),
- \(F\): finalizing method to compute the solution from the working vector \(y = FY\).
- \(F_{\text{embed}}\): coefficients for computing the auxiliary solution \(\tilde{y}\) from the working vector \((\tilde{y} = F_{\text{embed}} Y)\),
- \(F_{\text{error}}\): coefficients to compute the estimated error vector from the working vector \((\varepsilon = F_{\text{error}} Y)\),
- \(S_{\text{error}}\): coefficients to initialize the auxiliary solution \((\tilde{y} \text{ or } \varepsilon)\) from a specified error vector \((\varepsilon)\). It is currently implemented only for \(r = 2\). We have \(y_{\text{aux}} = S_{\text{error}}[0] \ast \varepsilon + S_{\text{error}}[1] \ast y\), where \(y_{\text{aux}}\) 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 $B, U, S, F, F_{\text{embed}}$, and $F_{\text{error}}$ 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
  
  ```c
  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 $F_{\text{embed}}Y$)
  
  ```c
  ierr = TSGetAuxSolution(TS, Vec*)
  ```

- **TSGetTimeError**: Returns the estimated error vector $\varepsilon$ (computed as $F_{\text{error}}Y$ if $n = 0$ or restores the error estimate at the end of the previous step if $n = -1$)
  
  ```c
  ierr = TSGetTimeError(TS, PetscInt n, Vec*)
  ```

- **TSSetTimeError**: Initializes the auxiliary solution ($\tilde{y}$ or $\varepsilon$) for a specified initial error.
  
  ```c
  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 $\varepsilon(n) = \frac{1}{1-\gamma} \varepsilon(n) - y(n))$.

Note that $y$ and $\tilde{y}$ are reported to TSAdapt basic (TSADAPTBASIC), and thus it computes the local error as $\varepsilon_{\text{loc}} = (\tilde{y} - y)$. However, the actual local error is $\varepsilon_{\text{loc}} = \varepsilon_{n+1} - \varepsilon_{n} = \frac{1}{1-\gamma} [ (\tilde{y} - y)_{n+1} - (\tilde{y} - y)_{n}]$.

Table 2.16 lists currently available GL schemes with global error estimation [Con16].

<table>
<thead>
<tr>
<th>TS</th>
<th>Reference</th>
<th>IM/EX</th>
<th>$(p, r, s)$</th>
<th>$\gamma$</th>
<th>Form</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSGLEE11</td>
<td>BE1</td>
<td>IM</td>
<td>(1,3,2)</td>
<td>0.5</td>
<td>$y\varepsilon$</td>
<td>Based on backward Euler</td>
</tr>
<tr>
<td>TSGLEE23</td>
<td>23</td>
<td>EX</td>
<td>(2,3,2)</td>
<td>0</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEE24</td>
<td>24</td>
<td>EX</td>
<td>(2,4,2)</td>
<td>0</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEE251</td>
<td>251</td>
<td>EX</td>
<td>(2,5,2)</td>
<td>0</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEE35</td>
<td>35</td>
<td>EX</td>
<td>(3,5,2)</td>
<td>0</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEEEXRK2A</td>
<td>exrk2a</td>
<td>EX</td>
<td>(2,6,2)</td>
<td>0.25</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEEERK32G1</td>
<td>rk32g1</td>
<td>EX</td>
<td>(3,8,2)</td>
<td>0</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>TSGLEEERK285EX</td>
<td>rk285ex</td>
<td>EX</td>
<td>(2,9,2)</td>
<td>0.25</td>
<td>$y\varepsilon$</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \dot{u} + \dot{F}(t, u) \)) 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 - 4th-order, 4-stage RK scheme).

2.5.7 Special Cases

- \( \dot{u} = Au \). First compute the matrix \( A \) then call

```c
TSSetProblemType(ts,TS_LINEAR);
TSSetRHSFunction(ts,NULL,TSComputeRHSFunctionLinear,NULL);
TSSetRHSJacobian(ts,A,A,TSComputeRHSJacobianConstant,NULL);
```

or

```c
TSSetProblemType(ts,TS_LINEAR);
TSSetIFunction(ts,NULL,TSComputeIFunctionLinear,NULL);
TSSetIJacobian(ts,A,A,YourComputeIJacobian,\&appctx);
```

- \( \dot{u} = A(t)u \). Use

```c
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

```c
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, `TSMonitorLGEError()`.
- **-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 available 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>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSADAPT-</td>
<td>none</td>
<td>no adaptivity</td>
</tr>
<tr>
<td>TSADAPT-</td>
<td>basic</td>
<td>the default adaptor</td>
</tr>
<tr>
<td>TSADAPT-</td>
<td>glee</td>
<td>extension of the basic adaptor to treat $\text{Tol}_A$ and $\text{Tol}_R$ as separate criteria. It can also control global errors if the integrator (e.g., <code>TSGLEE</code>) provides this information</td>
</tr>
</tbody>
</table>

When using `TSADAPT-BASIC` (the default), the user typically provides a desired absolute $\text{Tol}_A$ or a relative $\text{Tol}_R$ 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 $\tilde{y}$ approximation, $\tilde{y}$, where $p$ is the order of the method and $\tilde{p}$ the order of $\tilde{y}$.

The adaptive controller at step $n$ computes a tolerance level

\[ Tol_n(i) = Tol_A(i) + \max(y_n(i), \tilde{y}_n(i))Tol_R(i), \]

and forms the acceptable error level

\[ \text{wlt}_{\text{e}} = \frac{1}{m} \sum_{i=1}^{m} \sqrt{\frac{\|y_n(i) - \tilde{y}_n(i)\|}{Tol(i)}}, \]
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

\[
wltn = \max_{i=1}^{m} \frac{\|y_n(i) - \hat{y}_n(i)\|}{Tol(i)}.
\]

The error tolerances are satisfied when \( wltn \leq 1.0 \).

The next step size is based on this error estimate, and determined by

\[
\Delta t_{\text{new}}(t) = \Delta t_{\text{old}} \min(\max(\alpha_{\text{min}}, \beta(1/wltn)^{-\frac{1}{\beta}})),
\]

where \( \alpha_{\text{min}} = -ts\_adapt\_clip[0] \) and \( \alpha_{\text{max}} = -ts\_adapt\_clip[1] \) keep the change in \( \Delta t \) 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 \( wltn_{k+1} \leq 1.0 \), 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 \((eq:hnew)\).

\textsc{TSAdaptGlee} is an extension of the basic adaptor to treat \( Tol_A \) and \( Tol_R \) as separate criteria. It can also control global errors if the integrator (e.g., \textsc{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

```c
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 \textsc{TStheta}, \textsc{TSARKIMEX}, and \textsc{TSROSW}.  

---

126 Chapter 2. Programming with PETSc/TAO
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

\[
\text{TSSetType(TS \, ts, TSType TSSUNDIALS);}\]

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

\[
\text{TSSundialsSetType(TS \, ts, TSSundialsLmmType [SUNDIALS_ADAMS, SUNDIALS_BDF])};
\]

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

\[
\text{TSSundialsGetPC(TS \, ts, PC *pc)};
\]

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

\[
\text{TSSundialsSetTolerance(TS \, ts, double abs, double rel)};
\]

where abs denotes the absolute tolerance and rel the relative tolerance.

Other PETSc-Sundials options include

\[
\text{TSSundialsSetGramSchmidtType(TS \, ts, TSSundialsGramSchmidtType type)};
\]

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 <modified,unmodified>.

The routine

\[
\text{TSSundialsSetMaxl(TS \, ts, PetscInt restart)};
\]

---

\(^6\) bitbucket.org/jedbrown/tchem

\(^7\) en.wikipedia.org/wiki/CHEMKIN
sets the number of vectors in the Krylov subspace used by GMRES. This may be set in the options database with `-ts_sundials_maxl maxl`.

### 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

\[ F(t, y, \dot{y}, p) = 0, \quad y(t_0) = y_0(p) \quad t_0 \leq t \leq t_F \]

and the cost function(s)

\[ \Psi_i(y_0, p) = \Phi_i(y_F, p) + \int_{t_0}^{t_F} r_i(y(t), p, t) dt \quad i = 1, \ldots, n_{\text{cost}}. \]

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

```c
TSSetSaveTrajectory(TS ts),
```

then calls `TSSolve()` in the usual manner.

One must create two arrays of \( n_{\text{cost}} \) vectors \( \lambda \) and \( \mu \) (if there are no parameters \( p \) then one can use `NULL` for the \( \mu \) array.) The \( \lambda \) 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. \( \lambda_i \) and \( \mu_i \) should be initialized with the values \( \frac{d\Phi_i}{dy}|_{t=t_F} \) and \( \frac{d\Phi_i}{dp}|_{t=t_F} \) respectively. Then one calls

```c
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

```c
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

\[
\text{TSCreateQuadratureTS(TS ts, PetscBool fwd, TS *quadts)};
\]

and provide the ODE RHS function (which evaluates the integrand \( r \)) with

\[
\text{TSSetRHSFunction(TS quadts, Vec R, PetscErrorCode (*rf)(TS, PetscReal, Vec, Vec, void*), void *ctx)};
\]

Similar to the settings for the original ODE, Jacobians of the integrand can be provided with

\[
\text{TSSetRHSJacobian(TS quadts, Vec DRDU, Vec DRDU, PetscErrorCode (*drdyf)(TS, PetscReal, Vec, Vec*, void*, void *ctx)},
\]

\[
\text{TSSetRHSJacobianP(TS quadts, Vec DRDU, Vec DRDU, PetscErrorCode (*drdpf)(TS, PetscReal, Vec*, Vec*, void*, void *ctx))};
\]

where \( drdyf = dr/dy \), \( drdpf = dr/dp \). Since the integral term is additive to the cost function, its gradient information will be included in \( \lambda \) and \( \mu \).

Lastly, one starts the backward run by calling

\[
\text{TSAdjointSolve(TS ts)};
\]

One can obtain the value of the integral term by calling

\[
\text{TSGetCostIntegral(TS ts, Vec *q)};
\]

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

\[
\begin{align*}
\phi' &= \omega_B (\omega - \omega_S) \\
2H/\omega_S \omega' &= p_m - p_{max} \sin(\phi) - D(\omega - \omega_S), & t_0 \leq t \leq t_F,
\end{align*}
\]

where \( \phi \) is the phase angle and \( \omega \) is the frequency.

The initial conditions at time \( t_0 \) are

\[
\begin{align*}
\phi(t_0) &= \arcsin (p_m/p_{max}) , \\
w(t_0) &= 1.
\end{align*}
\]

\( p_{max} \) 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

$$\Psi(p_m, \phi) = -p_m + c \int_{t_0}^{t_F} (\max(0, \phi - \phi_S))^2 \, dt$$

with respect to the parameter $p_m$. $\text{numcost}$ 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 $f_p = [0 \, 1]^T$ using TSASetRHSJacobianP(), and three routines corresponding to the integrand $r = c (\max(0, \phi - \phi_S))^2$, $r_p = [0 \, 0]^T$ and $r_y = [2c(\max(0, \phi - \phi_S)) \, 0]^T$ using TSSetCostIntegrand().

In the adjoint run, $\lambda$ and $\mu$ 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 $\lambda$ (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

$$\frac{d\Psi}{dp_m} = \mu(t_0) + \lambda(t_0) \frac{d[\phi(t_0) \, \omega(t_0)]}{dp_m}.$$

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.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,
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 workflow,
- -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

\[ u_t = \frac{u^{n+1} - u^n}{dt^n} \]

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

  \[
  \text{TSSetRHSFunction(TS ts, Vec r, PetscErrorCode (*f)(TS, PetscReal, Vec, Vec, void*), void *, *fP);}
  \]

  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

  \[
  \text{TSSetRHSJacobian(TS ts, Mat Amat, Mat Pmat, PetscErrorCode (*f)(TS, PetscReal, Vec, Mat, Mat, void*), void *fP)};
  \]

  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

  \[
  \text{TSPseudoSetTimeStep(TS ts, PetscInt(*dt)(TS, PetscReal*, void*), void* dtctx)};
  \]

  The function \( dt \) is a user-provided function that computes the next pseudo-timestep. As a default one can use \( \text{TSPseudoTimeStepDefault(TS, PetscReal*, void*}) \) 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

  \[
  \text{TSPseudoIncrementDtFromInitialDt(TS ts)};
  \]

  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

  \[
  \text{TSPseudoSetTimeStepIncrement(TS ts, PetscReal inc)};
  \]

  or the option \(-ts\_pseudo\_increment <inc>\).

- For location-dependent pseudo-timestepping, the interface function has not yet been created.
2.8 TAO: Optimization Solvers

The TAO manual has been migrated to Sphinx, and is now part of PETSc manual at.

2.8.1 Preface

The Toolkit for Advanced Optimization (TAO) focuses on the development of algorithms and software for the solution of large-scale optimization problems on high-performance architectures. Areas of interest include unconstrained and bound-constrained optimization, nonlinear least squares problems, optimization problems with partial differential equation constraints, and variational inequalities and complementarity constraints.

The development of TAO was motivated by the scattered support for parallel computations and the lack of reuse of external toolkits in current optimization software. Our aim is to produce high-quality optimization software for computing environments ranging from workstations and laptops to massively parallel high-performance architectures. Our design decisions are strongly motivated by the challenges inherent in the use of large-scale distributed memory architectures and the reality of working with large, often poorly structured legacy codes for specific applications.

2.8.2 Changes in Version 3.5

TAO is now included in the PETSc distribution and the PETSc repository, thus it versions will always match the PETSc version. The TaoSolver object is now simply Tao and there is no TaoInitialize() or TaoFinalize(). Numerous changes have been made to make the source code more PETSc-like. All future changes will be listed in the PETSc changes documentation.

2.8.3 Changes in Version 2.0

TAO version numbers will now adhere to the new PETSc standard of Major-Minor-Patch. Any patch-level changes will have an attempt to keep the application programming interface (API) untouched, but in any case backward compatibility with previous version of the minor version will be maintained.

Many new features and interface changes were introduced in TAO version 2.0 (and continue to be used in version 2.2.0). TAO applications created for previous versions will need to be updated to work with the new version. We apologize for any inconvenience this situation may cause; these changes were needed to keep the interface clean, clear, and easy to use. Some of the most important changes are highlighted below.

Elimination of the TaoApplication Object. The largest change to the TAO programming interface was the elimination of the TaoApplication data structure. In previous versions of TAO, this structure was created by the application programmer for application-specific data and routines. In order to more closely follow PETSc design principles, this information is now directly attached to a Tao object instead. See TAO Solvers for a listing of what the most common TAO routines now look like without the TaoApplication object.

New Algorithms. TAO has a new algorithm for solving derivative-free nonlinear least squares problems, POUNDerS, that can efficiently solve parameter optimization problems when no derivatives are available and function evaluations are expensive. See POUNDerS for more information on the details of the algorithm and Nonlinear Least-Squares for how to use it. TAO now also provides a new algorithm for the solution of optimization problems with partial differential equation (PDE) constraints based on a linearly constrained augmented Lagrangian (LCL) method. More information on PDE-constrained optimization and LCL can be found in PDE-Constrained Optimization.

TaoLineSearch Object. TAO has promoted the line search to a full object. Any of the available TAO line search algorithms (Armijo, Moré-Thuente, GPCG, and unit) can now be selected regardless of the
overlying TAO algorithm. Users can also create new line search algorithms that may be more suitable for their applications. More information is available in Line Searches.

Better Adherence to PETSc Style. TAO now features a tighter association with PETSc standards and practices. All TAO constructs now follow PETSc conventions and are written in C. There is no longer a separate abstract class for vectors, matrices, and linear solvers. TAO now uses these PETSc objects directly. We believe these changes make TAO applications much easier to create and maintain for users already familiar with PETSc programming. These changes also allow TAO to relax some of the previously imposed requirements on the PETSc configuration. TAO now works with PETSc configured with single-precision and quad-precision arithmetic when using GNU compilers and no longer requires a C++ compiler. However, TAO is not compatible with PETSc installations using complex data types.

2.8.4 Acknowledgments

We especially thank Jorge Moré for his leadership, vision, and effort on previous versions of TAO.

TAO relies on PETSc for the linear algebra required to solve optimization problems, and we have benefited from the PETSc team’s experience, tools, software, and advice. In many ways, TAO is a natural outcome of the PETSc development.

TAO has benefited from the work of various researchers who have provided solvers, test problems, and interfaces. In particular, we acknowledge Lisa Grignon, Elizabeth Dolan, Boyana Norris, Gabriel Lopez-Calva, Yurii Zinchenko, Michael Gertz, Jarek Nieplocha, Limin Zhang, Manojkumar Krishnan, and Evan Gawlik. We also thank all TAO users for their comments, bug reports, and encouragement.

The development of TAO is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. We also thank the Argonne Laboratory Computing Resource Center and the National Energy Research Scientific Computing Center for allowing us to test and run TAO applications on their machines.

2.8.5 Introduction

The Toolkit for Advanced Optimization (TAO) focuses on the design and implementation of optimization software for solving large-scale optimization applications on high-performance architectures. Our approach is motivated by the scattered support for parallel computations and lack of reuse of linear algebra software in currently available optimization software. The TAO design allows the reuse of toolkits that provide lower-level support (parallel sparse matrix data structures, preconditioners, solvers), and thus we are able to build on top of these toolkits instead of having to redevelop code. The advantages in terms of efficiency and development time are significant. This section provides a short introduction to our design philosophy and the importance of this design.

The TAO design philosophy place strong emphasis on the reuse of external tools where appropriate. Our design enables bidirectional connection to lower-level linear algebra support (e.g., parallel sparse matrix data structures) provided in toolkits such as PETSc as well as higher-level application frameworks. Our design decisions are strongly motivated by the challenges inherent in the use of large-scale distributed memory architectures and the reality of working with large and often poorly structured legacy codes for specific applications. Fig. 2.5 illustrates how the TAO software works with external libraries and application code.

The TAO solvers use fundamental PETSc objects to define and solve optimization problems: vectors, matrices, index sets, and linear solvers. The concepts of vectors and matrices are standard, while an index set refers to a set of integers used to identify particular elements of vectors or matrices. An optimization algorithm is a sequence of well-defined operations on these objects. These operations include vector sums, inner products, and matrix-vector multiplication.

With sufficiently flexible abstract interfaces, PETSc can support a variety of implementations of data structures and algorithms. These abstractions allow us to more easily experiment with a range of algorithmic
and data structure options for realistic problems. Such capabilities are critical for making high-performance optimization software adaptable to the continual evolution of parallel and distributed architectures and the research community’s discovery of new algorithms that exploit their features.

2.8.6 Getting Started

TAO can be used on a personal computer with a single processor or within a parallel environment. Its basic usage involves only a few commands, but fully understanding its usage requires time. Application programmers can easily begin to use TAO by working with the examples provided and then gradually learn more details according to their needs. The current version of TAO and the most recent help concerning installation and usage can be found at https://www.petsc.org

See PETSc/TAO Users Manual and /install/index for how to install and start using PETSc/TAO.

Writing Application Codes with TAO

Examples throughout the library demonstrate the software usage and can serve as templates for developing custom applications. We suggest that new TAO users examine programs in ${PETSC_DIR}/src/tao/<unconstrained,bound,..>/tutorials

The manual pages provide indices (organized by both routine names and concepts) to the tutorial examples. We suggest the following procedure for writing a new application program using TAO:

1. Install PETSc/TAO according to the instructions at /install/index.
2. Copy an example and makefile from the directories \${PETSC_DIR}/src/tao/<unconstrained, bound,..>/tutorials.
   compile the example, and run the program.

3. Select the example program matching the application most closely, and use it as a starting point for developing a customized code.

### A Simple TAO Example

To help the user start using TAO immediately, we introduce here a simple uniprocessor example. Please read *Using TAO Solvers* for a more in-depth discussion on using the TAO solvers. The code presented below minimizes the extended Rosenbrock function \( f: \mathbb{R}^n \to \mathbb{R} \) defined by

\[
  f(x) = \sum_{i=0}^{m-1} \left( \alpha (x_{2i+1} - x_{2i}^2)^2 + (1 - x_{2i})^2 \right),
\]

where \( n = 2m \) is the number of variables. Note that while we use the C language to introduce the TAO software, the package is fully usable from C++ and Fortran77/90. *PETSc for Fortran Users* discusses additional issues concerning Fortran usage.

The code in the example contains many of the components needed to write most TAO programs and thus is illustrative of the features present in complex optimization problems. Note that for display purposes we have omitted some nonessential lines of code as well as the (essential) code required for the routine `FormFunctionGradient`, which evaluates the function and gradient, and the code for `FormHessian`, which evaluates the Hessian matrix for Rosenbrock’s function. The complete code is available in \${TAO_DIR}/src/unconstrained/tutorials/rosenbrock1.c. The following sections annotate the lines of code in the example.

**Listing:** `src/tao/unconstrained/tutorials/rosenbrock1.c`

```c
#include <petsctao.h>
typedef struct {
    PetscInt n;         /* dimension */
    PetscReal alpha;    /* condition parameter */
    PetscBool chained;
} AppCtx;

/* -------------- User-defined routines ---------- */
PetscErrorCode FormFunctionGradient(Tao,Vec,PetscReal*,Vec,void*);
PetscErrorCode FormHessian(Tao,Vec,Mat,Mat,void*);

int main(int argc,char **argv)
{
    PetscErrorCode ierr;    /* used to check for functions returning nonzeros */
    PetscReal zero=0.0;
    Vec x;                   /* solution vector */
    Mat H;
    Tao tao;                 /* Tao solver context */
    PetscBool flg, test_lmvm = PETSC_FALSE;
    PetscMPIInt size;       /* number of processes running */
    AppCtx user;            /* user-defined application context */
    KSP ksp;
    PC pc;
    Mat M;
```

(continues on next page)
/* Initialize TAO and PETSc */
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_WRONG_MPI_SIZE,"Incorrect number of processors");

/* Initialize problem parameters */
user.n = 2; user.alpha = 99.0; user.chained = PETSC_FALSE;
/* Check for command line arguments to override defaults */
ierr = PetscOptionsGetInt(NULL,NULL,"-n",&user.n,&flg);CHKERRQ(ierr);
ierr = PetscOptionsGetReal(NULL,NULL,"-alpha",&user.alpha,&flg);CHKERRQ(ierr);
ierr = PetscOptionsGetBool(NULL,NULL,"-chained",&user.chained,&flg);CHKERRQ(ierr);
ierr = PetscOptionsGetBool(NULL,NULL,"-test_lmvm",&test_lmvm,&flg);CHKERRQ(ierr);

/* Allocate vectors for the solution and gradient */
ierr = VecCreateSeq(PETSC_COMM_SELF,user.n,&x);CHKERRQ(ierr);
ierr = MatCreateSeqBAIJ(PETSC_COMM_SELF,2,user.n,user.n,1,NULL,&H);CHKERRQ(ierr);

/* The TAO code begins here */

/* Create TAO solver with desired solution method */
ierr = TaoCreate(PETSC_COMM_SELF,&tao);CHKERRQ(ierr);
ierr = TaoSetType(tao,TAOLMVM);CHKERRQ(ierr);

/* Set solution vec and an initial guess */
ierr = VecSet(x,zero);CHKERRQ(ierr);
ierr = TaoSetInitialVector(tao,x);CHKERRQ(ierr);

/* Set routines for function, gradient, hessian evaluation */
ierr = TaoSetObjectiveAndGradientRoutine(tao,FormFunctionGradient,&user);
CHKERRQ(ierr);
ierr = TaoSetHessianRoutine(tao,H,H,FormHessian,&user);CHKERRQ(ierr);

/* Test the LMVM matrix */
if (test_lmvm) {
  ierr = PetscOptionsSetValue(NULL,"-tao_type","bqnktr");CHKERRQ(ierr);
}

/* Check for TAO command line options */
ierr = TaoSetFromOptions(tao);CHKERRQ(ierr);

/* SOLVE THE APPLICATION */
ierr = TaoSolve(tao);CHKERRQ(ierr);

/* Test the LMVM matrix */
if (test_lmvm) {
  ierr = TaoGetKSP(tao,&ksp);CHKERRQ(ierr);
ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = PCLMVMGetMatLMVM(pc,&M);CHKERRQ(ierr);
ierr = VecDuplicate(x,&in);CHKERRQ(ierr);
ierr = VecDuplicate(x,&out);CHKERRQ(ierr);
ierr = VecDuplicate(x,&out2);CHKERRQ(ierr);
ierr = VecSet(in,1.0);CHKERRQ(ierr);
}
ierr = MatMult(M, in, out); CHKERRQ(ierr);
ierr = MatSolve(M, out, out2); CHKERRQ(ierr);
ierr = VecAXPY(out2, -1.0, in); CHKERRQ(ierr);
ierr = VecNorm(out2, NORM_2, &mult_solve_dist); CHKERRQ(ierr);
if (mult_solve_dist < 1.0e-11) {
    ierr = PetscPrintf(PetscObjectComm((PetscObject)tao), "error between LMVM, _MatMult and MatSolve: < 1.0e-11\n"); CHKERRQ(ierr);
} else if (mult_solve_dist < 1.0e-6) {
    ierr = PetscPrintf(PetscObjectComm((PetscObject)tao), "error between LMVM, _MatMult and MatSolve: < 1.0e-6\n"); CHKERRQ(ierr);
} else {
    ierr = VecDestroy(&in); CHKERRQ(ierr);
    ierr = VecDestroy(&out); CHKERRQ(ierr);
    ierr = VecDestroy(&out2); CHKERRQ(ierr);
}

ierr = TaoDestroy(&tao); CHKERRQ(ierr);
ierr = VecDestroy(&x); CHKERRQ(ierr);
ierr = MatDestroy(&H); CHKERRQ(ierr);

ierr = PetscFinalize();
return ierr;

Include Files

The include file for TAO should be used via the statement

```c
#include <petsctao.h>
```

The required lower-level include files are automatically included within this high-level file.

TAO Solvers

Many TAO applications will follow an ordered set of procedures for solving an optimization problem: The user creates a Tao context and selects a default algorithm. Call-back routines as well as vector (Vec) and matrix (Mat) data structures are then set. These call-back routines will be used for evaluating the objective function, gradient, and perhaps the Hessian matrix. The user then invokes TAO to solve the optimization problem and finally destroys the Tao context. A list of the necessary functions for performing these steps using TAO is show below. Details of these commands are presented in Using TAO Solvers.

```c
TaoCreate(MPI_Comm comm, Tao *tao);
TaoSetType(Tao tao, TaoType type);
TaoSetInitialVector(Tao tao, Vec x);
TaoSetObjectiveAndGradientRoutine(Tao tao,
    PetscErrorCode (*FormFGradient)(Tao,Vec,PetscReal*,Vec,void*),
    void *user);
TaoSetHessianRoutine(Tao tao, Mat H, Mat Hpre,
    PetscErrorCode (*FormHessian)(Tao,Vec,Mat,Mat,
    void*), void *user);
```
Note that the solver algorithm selected through the function `TaoSetType()` can be overridden at runtime by using an options database. Through this database, the user not only can select a minimization method (e.g., limited-memory variable metric, conjugate gradient, Newton with line search or trust region) but also can prescribe the convergence tolerance, set various monitoring routines, set iterative methods and preconditions for solving the linear systems, and so forth. See *Using TAO Solvers* for more information on the solver methods available in TAO.

### Function Evaluations

Users of TAO are required to provide routines that perform function evaluations. Depending on the solver chosen, they may also have to write routines that evaluate the gradient vector and Hessian matrix.

### Programming with PETSc

TAO relies heavily on PETSc not only for its vectors, matrices, and linear solvers but also for its programming utilities such as command line option handling, error handling, and compiling system. We provide here a quick overview of some of these PETSc features. Please refer to the *PETSc/TAO Users Manual* for a more in-depth discussion of PETSc.

### Vectors

In the example above the vector data structure (`Vec`) is used to store the solution and gradient for the TAO unconstrained minimization solvers. A new parallel or sequential vector $\mathbf{x}$ of global dimension $M$ is created with the command

```c
info = VecCreate(MPI_Comm comm, int m, int M, Vec *x);
```

where `comm` denotes the MPI communicator. The type of storage for the vector may be set with calls either to `VecSetType()` or to `VecSetFromOptions()`. Additional vectors of the same type can be formed with

```c
info = VecDuplicate(Vec old, Vec *new);
```

The commands

```c
info = VecSet(Vec x, PetscScalar value);
info = VecSetValues(Vec x, int n, int *indices,
                     Scalar *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, may be found in the *PETSc/TAO Users Manual*. 

**Notes:**
- `TaoSolve(Tao tao);`
- `TaoDestroy(Tao tao);`
Matrices

Usage of matrices and vectors is similar. The user can create a new parallel or sequential matrix \( H \) with \( M \) global rows and \( N \) global columns, with the routines

\[
\text{ierr} = \text{MatCreate}(\text{MPI Comm comm,Mat } *H); \\
\text{ierr} = \text{MatSetSizes}(H,\text{PETSC_DECIDE,PETSC_DECIDE},M,N);
\]

where the matrix format can be specified at runtime. The user could alternatively specify each process's number of local rows and columns using \( m \) and \( n \) instead of \text{PETSC_DECIDE}. \( H \) can then be used to store the Hessian matrix, as indicated by the call to \text{TaoSetHessianMat()}. Matrix entries can be set with the command

\[
\text{ierr} = \text{MatSetValues(Mat } H,\text{PetscInt } m,\text{PetscInt } *im, \\
\text{PetscInt } n,\text{PetscInt } *in, \text{PetscScalar } *values,\text{INSERT_VALUES});
\]

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

\[
\text{ierr} = \text{MatAssemblyBegin(Mat } H,\text{MAT_FINAL_ASSEMBLY}); \\
\text{ierr} = \text{MatAssemblyEnd(Mat } H,\text{MAT_FINAL_ASSEMBLY});
\]

The \text{PETSc/TAO Users Manual} discusses various matrix formats as well as the details of some basic matrix manipulation routines.

The Options Database

A TAO application can access the command line options presented at runtime through the PETSc options database. This database gives the application author the ability to set and change application parameters without the need to recompile the application. For example, an application may have a grid discretization parameter \( nx \) that can be set with the command line option \text{-nx <integer>}. The application can read this option with the following line of code:

\[
\text{PetscOptionsGetInt(NULL,NULL, \\
"-nx", &nx, &flg);}\]

If the command line option is present, the variable \( nx \) is set accordingly; otherwise, \( nx \) remains unchanged. A complete description of the options database may be found in the \text{PETSc/TAO Users Manual}.

Error Checking

All TAO commands begin with the \text{Tao} prefix and return an integer indicating whether an error has occurred during the call. The error code equals zero after the successful completion of the routine and is set to a nonzero value if an error has been detected. The macro \text{CHKERRQ(ierr)} checks the value of \text{ierr} and calls an error handler upon error detection. \text{CHKERRQ()} should be used after all subroutines to enable a complete error traceback.

Below, we indicate a traceback generated by error detection within a sample program. The error occurred on line 2110 of the file \${PETSC_DIR}/src/mat/interface/matrix.c in the routine \text{MatMult()} and was caused by failure to assemble the matrix in the Hessian evaluation routine. The \text{MatMult()} routine was called from the \text{TaoSolve_NLS()} routine, which was in turn called on line 154 of \text{TaoSolve()} from the \text{main()} routine in the program \text{rosenbrock1.c}. The \text{PETSc/TAO Users Manual} provides further details regarding error checking, including information about error handling in Fortran.
When running the debugging version of the TAO software (PETSc configured with the (default) --with-debugging option), checking is performed for memory corruption (writing outside of array bounds, etc). The macros CHKMEMQ and CHKMEMA can be called anywhere in the code and, when used together with the command line option -malloc_debug, check the current status of the memory for corruption. By putting several (or many) of these macros into an application code, one can usually track down the code segment where corruption has occurred.

**Parallel Programming**

Since TAO 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. By default, however, the user is shielded from many of the details of message passing within TAO, since these are hidden within parallel objects, such as vectors, matrices, and solvers. In addition, TAO users can interface to external tools, such as the generalized vector scatters/gathers and distributed arrays within PETSc, for assistance in managing parallel data.

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

```c
ierr = MatCreate(MPI_Comm comm,Mat *H);
ierr = VecCreate(MPI_Comm comm,Vec *x);
ierr = TaoCreate(MPI_Comm comm,Tao *tao);
```

In most cases, the value for `comm` will be either `PETSC_COMM_SELF` for single-process objects or `PETSC_COMM_WORLD` for objects distributed over all processors. 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, the routines must be called in the same order on each processor.
2.8.7 Using TAO Solvers

TAO contains unconstrained minimization, bound-constrained minimization, nonlinear complementarity, nonlinear least squares solvers, and solvers for optimization problems with partial differential equation constraints. The structure of these problems can differ significantly, but TAO has a similar interface to all its solvers. Routines that most solvers have in common are discussed in this section. A complete list of options can be found by consulting the manual pages. Many of the options can also be set at the command line. These options can also be found by running a program with the `-help` option.

Header File

TAO applications written in C/C++ should have the statement

```
#include <petsctao.h>
```

in each file that uses a routine in the TAO libraries.

Creation and Destruction

A TAO solver can be created by calling the

```
TaoCreate(MPI_Comm comm, Tao *newsolver);
```

routine. Much like creating PETSc vector and matrix objects, the first argument is an MPI `communicator`. An MPI\(^1\) communicator indicates a collection of processors that will be used to evaluate the objective function, compute constraints, and provide derivative information. When only one processor is being used, the communicator `PETSC_COMM_SELF` can be used with no understanding of MPI. Even parallel users need to be familiar with only the basic concepts of message passing and distributed-memory computing. Most applications running TAO in parallel environments can employ the communicator `PETSC_COMM_WORLD` to indicate all processes known to PETSc in a given run.

The routine

```
TaoSetType(Tao tao, TaoType type);
```

can be used to set the algorithm TAO uses to solve the application. The various types of TAO solvers and the flags that identify them will be discussed in the following sections. The solution method should be carefully chosen depending on the problem being solved. Some solvers, for instance, are meant for problems with no constraints, whereas other solvers acknowledge constraints in the problem and handle them accordingly. The user must also be aware of the derivative information that is available. Some solvers require second-order information, while other solvers require only gradient or function information. The command line option `-tao_method` (or equivalently `-tao_type`) followed by a TAO method will override any method specified by the second argument. The command line option `-tao_method tao_lmmv`, for instance, will specify the limited-memory, variable metric method for unconstrained optimization. Note that the `TaoType` variable is a string that requires quotation marks in an application program, but quotation marks are not required at the command line.

Each TAO solver that has been created should also be destroyed by using the

```
TaoDestroy(Tao tao);
```

command. This routine frees the internal data structures used by the solver.

\(^1\) For more on MPI and PETSc, see *Running PETSc Programs*. 

142 Chapter 2. Programming with PETSc/TAO
TAO Applications

The solvers in TAO address applications that have a set of variables, an objective function, and possibly constraints on the variables. Many solvers also require derivatives of the objective and constraint functions. To use the TAO solvers, the application developer must define a set of variables, implement routines that evaluate the objective function and constraint functions, and pass this information to a TAO application object.

TAO uses vector and matrix objects to pass this information from the application to the solver. The set of variables, for instance, is represented in a vector. The gradient of an objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, evaluated at a point, is also represented as a vector. Matrices, on the other hand, can be used to represent the Hessian of $f$ or the Jacobian of a constraint function $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$. The TAO solvers use these objects to compute a solution to the application.

Defining Variables

In all the optimization solvers, the application must provide a Vec object of appropriate dimension to represent the variables. This vector will be cloned by the solvers to create additional work space within the solver. If this vector is distributed over multiple processors, it should have a parallel distribution that allows for efficient scaling, inner products, and function evaluations. This vector can be passed to the application object by using the

```c
TaoSetInitialVector(Tao, Vec);
```

routine. When using this routine, the application should initialize the vector with an approximate solution of the optimization problem before calling the TAO solver. This vector will be used by the TAO solver to store the solution. Elsewhere in the application, this solution vector can be retrieved from the application object by using the

```c
TaoGetSolutionVector(Tao, Vec *);
```

routine. This routine takes the address of a Vec in the second argument and sets it to the solution vector used in the application.

Application Context

Writing a TAO application may require use of an application context. An application context is a structure or object defined by an application developer, passed into a routine also written by the application developer, and used within the routine to perform its stated task.

For example, a routine that evaluates an objective function may need parameters, work vectors, and other information. This information, which may be specific to an application and necessary to evaluate the objective, can be collected in a single structure and used as one of the arguments in the routine. The address of this structure will be cast as type (void*) and passed to the routine in the final argument. Many examples of these structures are included in the TAO distribution.

This technique offers several advantages. In particular, it allows for a uniform interface between TAO and the applications. The fundamental information needed by TAO appears in the arguments of the routine, while data specific to an application and its implementation is confined to an opaque pointer. The routines can access information created outside the local scope without the use of global variables. The TAO solvers and application objects will never access this structure, so the application developer has complete freedom to define it. If no such structure or needed by the application then a NULL pointer can be used.

2.8. TAO: Optimization Solvers
Objective Function and Gradient Routines

TAO solvers that minimize an objective function require the application to evaluate the objective function. Some solvers may also require the application to evaluate derivatives of the objective function. Routines that perform these computations must be identified to the application object and must follow a strict calling sequence.

Routines should follow the form

```c
PetscErrorCode EvaluateObjective(Tao, Vec, PetscReal*, void*);
```

in order to evaluate an objective function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \). The first argument is the TAO Solver object, the second argument is the \( n \)-dimensional vector that identifies where the objective should be evaluated, and the fourth argument is an application context. This routine should use the third argument to return the objective value evaluated at the point specified by the vector in the second argument.

This routine, and the application context, should be passed to the application object by using the

```c
TaoSetObjectiveRoutine(Tao,
    PetscErrorCode(*)(Tao, Vec, PetscReal*, void*),
    void*);
```

routine. The first argument in this routine is the TAO solver object, the second argument is a function pointer to the routine that evaluates the objective, and the third argument is the pointer to an appropriate application context. Although the final argument may point to anything, it must be cast as a \((\text{void}*)\) type. This pointer will be passed back to the developer in the fourth argument of the routine that evaluates the objective. In this routine, the pointer can be cast back to the appropriate type. Examples of these structures and their usage are provided in the distribution.

Many TAO solvers also require gradient information from the application. The gradient of the objective function is specified in a similar manner. Routines that evaluate the gradient should have the calling sequence

```c
PetscErrorCode EvaluateGradient(Tao, Vec, Vec, void*);
```

where the first argument is the TAO solver object, the second argument is the variable vector, the third argument is the gradient vector, and the fourth argument is the user-defined application context. Only the third argument in this routine is different from the arguments in the routine for evaluating the objective function. The numbers in the gradient vector have no meaning when passed into this routine, but they should represent the gradient of the objective at the specified point at the end of the routine. This routine, and the user-defined pointer, can be passed to the application object by using the

```c
TaoSetGradientRoutine(Tao,
    PetscErrorCode(*)(Tao, Vec, Vec, void*),
    void*);
```

routine. In this routine, the first argument is the Tao object, the second argument is the function pointer, and the third object is the application context, cast to \((\text{void}*)\).

Instead of evaluating the objective and its gradient in separate routines, TAO also allows the user to evaluate the function and the gradient in the same routine. In fact, some solvers are more efficient when both function and gradient information can be computed in the same routine. These routines should follow the form

```c
PetscErrorCode EvaluateFunctionAndGradient(Tao, Vec,
    PetscReal*, Vec, void*);
```

where the first argument is the TAO solver and the second argument points to the input vector for use in evaluating the function and gradient. The third argument should return the function value, while the fourth
argument should return the gradient vector. The fifth argument is a pointer to a user-defined context. This context and the name of the routine should be set with the call

```c
TaoSetObjectiveAndGradientRoutine(Tao, 
    PetscErrorCode (*)(Tao, Vec, PetscReal*, Vec, void*),
    void *
);
```

where the arguments are the TAO application, a function name, and a pointer to a user-defined context.

The TAO example problems demonstrate the use of these application contexts as well as specific instances of function, gradient, and Hessian evaluation routines. All these routines should return the integer 0 after successful completion and a nonzero integer if the function is undefined at that point or an error occurred.

### Hessian Evaluation

Some optimization routines also require a Hessian matrix from the user. The routine that evaluates the Hessian should have the form

```c
PetscErrorCode EvaluateHessian(Tao, Vec, Mat, Mat, 
    void *
);
```

where the first argument of this routine is a TAO solver object. The second argument is the point at which the Hessian should be evaluated. The third argument is the Hessian matrix, and the sixth argument is a user-defined context. Since the Hessian matrix is usually used in solving a system of linear equations, a preconditioner for the matrix is often needed. The fourth argument is the matrix that will be used for preconditioning the linear system; in most cases, this matrix will be the same as the Hessian matrix. The fifth argument is the flag used to set the Hessian matrix and linear solver in the routine `KSPSetOperators()`.

One can set the Hessian evaluation routine by calling the

```c
TaoSetHessianRoutine(Tao, Mat H, Mat Hpre, 
    PetscErrorCode (*)(Tao, Vec, Mat, Mat, 
        void*), void *
);
```

routine. The first argument is the TAO Solver object. The second and third arguments are, respectively, the Mat object where the Hessian will be stored and the Mat object that will be used for the preconditioning (they may be the same). The fourth argument is the function that evaluates the Hessian, and the fifth argument is a pointer to a user-defined context, cast to `(void*)`.

### Finite Differences

Finite-difference approximations can be used to compute the gradient and the Hessian of an objective function. These approximations will slow the solve considerably and are recommended primarily for checking the accuracy of hand-coded gradients and Hessians. These routines are

```c
TaoDefaultComputeGradient(Tao, Vec, Vec, void*);
```

and

```c
TaoDefaultComputeHessian(Tao, Vec, Mat*, Mat*, void*);
```

respectively. They can be set by using `TaoSetGradientRoutine()` and `TaoSetHessianRoutine()` or through the options database with the options `-tao_fdgrad` and `-tao_fd`, respectively.

The efficiency of the finite-difference Hessian can be improved if the coloring of the matrix is known. If the application programmer creates a PETSc `MatFDColoring` object, it can be applied to the finite-difference approximation by setting the Hessian evaluation routine to
TaoDefaultComputeHessianColor(Tao, Vec, Mat*, Mat*, void*);

and using the MatFDColoring object as the last (void *) argument to TaoSetHessianRoutine().

One also can use finite-difference approximations to directly check the correctness of the gradient and/or Hessian evaluation routines. This process can be initiated from the command line by using the special TAO solver tao_fd_test together with the option -tao_test_gradient or -tao_test_hessian.

Matrix-Free Methods

TAO fully supports matrix-free methods. The matrices specified in the Hessian 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 (PCHELL). In other words, matrix-free methods cannot be used if a direct solver is to be employed. Details about using matrix-free methods are provided in the PETSc/TAO Users Manual.

Bounds on Variables

Some optimization problems also impose constraints on the variables. The constraints may impose simple bounds on the variables or require that the variables satisfy a set of linear or nonlinear equations.

The simplest type of constraint on an optimization problem puts lower or upper bounds on the variables. Vectors that represent lower and upper bounds for each variable can be set with the

TaoSetVariableBounds(Tao, Vec, Vec);

command. The first vector and second vector should contain the lower and upper bounds, respectively. When no upper or lower bound exists for a variable, the bound may be set to TAO_INFINITY or TAO_NINFINITY. After the two bound vectors have been set, they may be accessed with the command TaoGetVariableBounds().

Alternatively, it may be more convenient for the user to designate a routine for computing these bounds that the solver will call before starting its algorithm. This routine will have the form

PetscErrorCode EvaluateBounds(Tao, Vec, Vec, void*);

where the two vectors, representing the lower and upper bounds respectfully, will be computed.

This routine can be set with the

TaoSetVariableBoundsRoutine(Tao
  PetscErrorCode *(Tao, Vec, Vec, void*), void*);

command.

Since not all solvers recognize the presence of bound constraints on variables, the user must be careful to select a solver that acknowledges these bounds.
Solving

Once the application and solver have been set up, the solver can be

TaoSolve(Tao);

routine. We discuss several universal options below.

Convergence

Although TAO and its solvers set default parameters that are useful for many problems, the user may need
to modify these parameters in order to change the behavior and convergence of various algorithms.

One convergence criterion for most algorithms concerns the number of digits of accuracy needed in the
solution. In particular, the convergence test employed by TAO attempts to stop when the error in the
constraints is less than $\epsilon_{crtol}$ and either

$$
\|g(X)\| \leq \epsilon_{gatol},
$$

$$
\|g(X)\|/|f(X)| \leq \epsilon_{grtol}, \text{ or}
$$

$$
\|g(X)\|/|g(X_0)| \leq \epsilon_{gttol},
$$

where $X$ is the current approximation to the true solution $X^*$ and $X_0$ is the initial guess. $X^*$ is unknown, so
TAO estimates $f(X) - f(X^*)$ with either the square of the norm of the gradient or the duality gap. A relative
tolerance of $\epsilon_{frtol} = 0.01$ indicates that two significant digits are desired in the objective function. Each solver
sets its own convergence tolerances, but they can be changed by using the routine TaoSetTolerances().

Another set of convergence tolerances terminates the solver when the norm of the gradient function (or
Lagrangian function for bound-constrained problems) is sufficiently close to zero.

Other stopping criteria include a minimum trust-region radius or a maximum number of iterations. These
parameters can be set with the routines TaoSetTrustRegionTolerance() and TaoSetMaximumIterations().
Similarly, a maximum number of function evaluations can be set with the command TaoSet-
MaximumFunctionEvaluations(). -tao_max_it, and -tao_max_funcs.

Viewing Status

To see parameters and performance statistics for the solver, the routine

TaoView(Tao tao)

can be used. This routine will display to standard output the number of function evaluations need by the
solver and other information specific to the solver. This same output can be produced by using the command
line option -tao_view.

The progress of the optimization solver can be monitored with the runtime option -tao_monitor. Although
monitoring routines can be customized, the default monitoring routine will print out several relevant statistics
to the screen.

The user also has access to information about the current solution. The current iteration number, objective
function value, gradient norm, infeasibility norm, and step length can be retrieved with the following
command.

TaoGetSolutionStatus(Tao tao, PetscInt *iterate, PetscReal *f,
PetscReal *gnorm, PetscReal *cnorm, PetscReal *xdiff,
TaoConvergedReason *reason)
The last argument returns a code that indicates the reason that the solver terminated. Positive numbers indicate that a solution has been found, while negative numbers indicate a failure. A list of reasons can be found in the manual page for `TaoGetConvergedReason()`.

### Obtaining a Solution

After exiting the `TaoSolve()` function, the solution, gradient, and with the following routines.

```c
TaoGetSolutionVector(Tao, Vec *X);
TaoGetGradientVector(Tao, Vec *G);
TaoComputeDualVariables(Tao, Vec X, Vec Duals);
```

Note that the `Vec` returned by `TaoGetSolutionVector` will be the same vector passed to `TaoSetInitialVector`. This information can be obtained during user-defined routines such as a function evaluation and customized monitoring routine or after the solver has terminated.

### Additional Options

Additional options for the TAO solver can be be set from the command line by using the

```c
TaoSetFromOptions(Tao)
```

routine. This command also provides information about runtime options when the user includes the `-help` option on the command line.

### Special Problem Structures

Below we discuss how to exploit the special structures for three classes of problems that TAO solves.

#### PDE-Constrained Optimization

TAO can solve PDE-constrained optimization applications of the form

\[
\min_{u,v} f(u,v) \quad \text{subject to} \quad g(u,v) = 0,
\]

where the state variable \(u\) is the solution to the discretized partial differential equation defined by \(g\) and parametrized by the design variable \(v\), and \(f\) is an objective function. In this case, the user needs to set routines for computing the objective function and its gradient, the constraints, and the Jacobian of the constraints with respect to the state and design variables. TAO also needs to know which variables in the solution vector correspond to state variables and which to design variables.

The objective and gradient routines are set as for other TAO applications, with `TaoSetObjectiveRoutine()` and `TaoSetGradientRoutine()`. The user can also provide a fused objective function and gradient evaluation with `TaoSetObjectiveAndGradientRoutine()`. The input and output vectors include the combined state and design variables. Index sets for the state and design variables must be passed to TAO by using the function

```c
TaosetStateDesignIS(Tao, IS, IS);
```
where the first IS is a PETSc IndexSet containing the indices of the state variables and the second IS the design variables.

Nonlinear equation constraints have the general form $c(x) = 0$, where $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$. These constraints should be specified in a routine, written by the user, that evaluates $c(x)$. The routine that evaluates the constraint equations should have the form

```c
PetscErrorCode EvaluateConstraints(Tao, Vec, Vec, void);
```

The first argument of this routine is a TAO solver object. The second argument is the variable vector at which the constraint function should be evaluated. The third argument is the vector of function values $c(x)$, and the fourth argument is a pointer to a user-defined context. This routine and the user-defined context should be set in the TAO solver with the

```c
TaoSetConstraintsRoutine(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec, void*), void*);
```

command. In this function, the first argument is the TAO solver object, the second argument a vector in which to store the constraints, the third argument is a function point to the routine for evaluating the constraints, and the fourth argument is a pointer to a user-defined context.

The Jacobian of $c(x)$ is the matrix in $\mathbb{R}^{m \times n}$ such that each column contains the partial derivatives of $c(x)$ with respect to one variable. The evaluation of the Jacobian of $c$ should be performed by calling the

```c
PetscErrorCode JacobianState(Tao, Vec, Mat, Mat, Mat, void*);
PetscErrorCode JacobianDesign(Tao, Vec, Mat*, void*);
```

routines. In these functions, the first argument is the TAO solver object. The second argument is the variable vector at which to evaluate the Jacobian matrix, the third argument is the Jacobian matrix, and the last argument is a pointer to a user-defined context. The fourth and fifth arguments of the Jacobian evaluation with respect to the state variables are for providing PETSc matrix objects for the preconditioner and for applying the inverse of the state Jacobian, respectively. This inverse matrix may be `PETSC_NULL`, in which case TAO will use a PETSc Krylov subspace solver to solve the state system. These evaluation routines should be registered with TAO by using the

```c
TaoSetJacobianStateRoutine(Tao, Mat, Mat, Mat, PetscErrorCode (*)(Tao, Vec, Mat, Mat, void*), void*);
TaoSetJacobianDesignRoutine(Tao, Mat, PetscErrorCode (*)(Tao, Vec, Mat*, void*), void*);
```

routines. The first argument is the TAO solver object, and the second argument is the matrix in which the Jacobian information can be stored. For the state Jacobian, the third argument is the matrix that will be used for preconditioning, and the fourth argument is an optional matrix for the inverse of the state Jacobian. One can use `PETSC_NULL` for this inverse argument and let PETSc apply the inverse using a KSP method, but faster results may be obtained by manipulating the structure of the Jacobian and providing an inverse. The fifth argument is the function pointer, and the sixth argument is an optional user-defined context. Since no solve is performed with the design Jacobian, there is no need to provide preconditioner or inverse matrices.
Nonlinear Least Squares

For nonlinear least squares applications, we are solving the optimization problem

$$\min_x \frac{1}{2} \| r(x) \|_2^2.$$ 

For these problems, the objective function value should be computed as a vector of residuals, $r(x)$, computed with a function of the form

```c
PetscErrorCode EvaluateResidual(Tao, Vec, Vec, void*);
```

and set with the

```c
TaoSetResidualRoutine(Tao, PetscErrorCode (*)(Tao, Vec, Vec, void*), void*);
```

routine. If required by the algorithm, the Jacobian of the residual, $J = \partial r(x) / \partial x$, should be computed with a function of the form

```c
PetscErrorCode EvaluateJacobian(Tao, Vec, Mat, void*);
```

and set with the

```c
TaoSetJacobianResidualRoutine(Tao, PetscErrorCode (*)(Tao, Vec, Mat, void*), void*);
```

routine.

Complementarity

Complementarity applications have equality constraints in the form of nonlinear equations $C(X) = 0$, where $C : \mathbb{R}^n \to \mathbb{R}^m$. These constraints should be specified in a routine written by the user with the form

```c
PetscErrorCode EqualityConstraints(Tao, Vec, Vec, void*);
```

that evaluates $C(X)$. The first argument of this routine is a TAO Solver object. The second argument is the variable vector $X$ at which the constraint function should be evaluated. The third argument is the output vector of function values $C(X)$, and the fourth argument is a pointer to a user-defined context.

This routine and the user-defined context must be registered with TAO by using the

```c
TaoSetConstraintRoutine(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec, void*), void*);
```

command. In this command, the first argument is TAO Solver object, the second argument is vector in which to store the function values, the third argument is the user-defined routine that evaluates $C(X)$, and the fourth argument is a pointer to a user-defined context that will be passed back to the user.

The Jacobian of the function is the matrix in $\mathbb{R}^{m \times n}$ such that each column contains the partial derivatives of $f$ with respect to one variable. The evaluation of the Jacobian of $C$ should be performed in a routine of the form

```c
PetscErrorCode EvaluateJacobian(Tao, Vec, Mat, Mat, void*);
```
In this function, the first argument is the TAO Solver object and the second argument is the variable vector at which to evaluate the Jacobian matrix. The third argument is the Jacobian matrix, and the sixth argument is a pointer to a user-defined context. Since the Jacobian matrix may be used in solving a system of linear equations, a preconditioner for the matrix may be needed. The fourth argument is the matrix that will be used for preconditioning the linear system; in most cases, this matrix will be the same as the Hessian matrix. The fifth argument is the flag used to set the Jacobian matrix and linear solver in the routine KSPSetOperators().

This routine should be specified to TAO by using the

```c
TaoSetJacobianRoutine(Tao, Mat J, Mat Jpre,
PETSCErrorCode (*)(Tao, Vec, Mat, Mat,
void*), void*);
```

command. The first argument is the TAO Solver object; the second and third arguments are the Mat objects in which the Jacobian will be stored and the Mat object that will be used for the preconditioning (they may be the same), respectively. The fourth argument is the function pointer; and the fifth argument is an optional user-defined context. The Jacobian matrix should be created in a way such that the product of it and the variable vector can be stored in the constraint vector.

### 2.8.8 TAO Solvers

TAO includes a variety of optimization algorithms for several classes of problems (unconstrained, bound-constrained, and PDE-constrained minimization, nonlinear least-squares, and complementarity). The TAO algorithms for solving these problems are detailed in this section, a particular algorithm can chosen by using the TaoSetType() function or using the command line arguments -tao_type <name>. For those interested in extending these algorithms or using new ones, please see Adding a Solver for more information.

**Unconstrained Minimization**

Unconstrained minimization is used to minimize a function of many variables without any constraints on the variables, such as bounds. The methods available in TAO for solving these problems can be classified according to the amount of derivative information required:

1. Function evaluation only – Nelder-Mead method (tao_nm)
2. Function and gradient evaluations – limited-memory, variable-metric method (tao_lmvn) and nonlinear conjugate gradient method (tao_cg)
3. Function, gradient, and Hessian evaluations – Newton line search method (tao_nls) and Newton trust-region method (tao_ntr)

The best method to use depends on the particular problem being solved and the accuracy required in the solution. If a Hessian evaluation routine is available, then the Newton line search and Newton trust-region methods will likely perform best. When a Hessian evaluation routine is not available, then the limited-memory, variable-metric method is likely to perform best. The Nelder-Mead method should be used only as a last resort when no gradient information is available.

Each solver has a set of options associated with it that can be set with command line arguments. These algorithms and the associated options are briefly discussed in this section.
Nelder-Mead Method

The Nelder-Mead algorithm \[\text{NM65}\] is a direct search method for finding a local minimum of a function \(f(x)\). This algorithm does not require any gradient or Hessian information of \(f\) and therefore has some expected advantages and disadvantages compared to the other TAO solvers. The obvious advantage is that it is easier to write an application when no derivatives need to be calculated. The downside is that this algorithm can be slow to converge or can even stagnate, and it performs poorly for large numbers of variables.

This solver keeps a set of \(N + 1\) sorted vectors \(x_1, x_2, \ldots, x_{N+1}\) and their corresponding objective function values \(f_1 \leq f_2 \leq \ldots \leq f_{N+1}\). At each iteration, \(x_{N+1}\) is removed from the set and replaced with

\[
x(\mu) = (1 + \mu) \frac{1}{N} \sum_{i=1}^{N} x_i - \mu x_{N+1},
\]

where \(\mu\) can be one of \(\mu_0, 2\mu_0, \frac{1}{2}\mu_0, -\frac{1}{2}\mu_0\) depending on the values of each possible \(f(x(\mu))\).

The algorithm terminates when the residual \(f_{N+1} - f_1\) becomes sufficiently small. Because of the way new vectors can be added to the sorted set, the minimum function value and/or the residual may not be impacted at each iteration.

Two options can be set specifically for the Nelder-Mead algorithm:

- **-tao_nm_lamda <value>** sets the initial set of vectors \((x_0\) plus \text{value} in each coordinate direction); the default value is 1.

- **-tao_nm_mu <value>** sets the value of \(\mu_0\); the default is \(\mu_0 = 1\).

Limited-Memory, Variable-Metric Method

The limited-memory, variable-metric method computes a positive definite approximation to the Hessian matrix from a limited number of previous iterates and gradient evaluations. A direction is then obtained by solving the system of equations

\[
H_k d_k = -\nabla f(x_k),
\]

where \(H_k\) is the Hessian approximation obtained by using the BFGS update formula. The inverse of \(H_k\) can readily be applied to obtain the direction \(d_k\). Having obtained the direction, a Moré-Thuente line search is applied to compute a step length, \(\tau_k\), that approximately solves the one-dimensional optimization problem

\[
\min_{\tau} f(x_k + \tau d_k).
\]

The current iterate and Hessian approximation are updated, and the process is repeated until the method converges. This algorithm is the default unconstrained minimization solver and can be selected by using the TAO solver tao_lmvm. For best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

The primary factors determining the behavior of this algorithm are the type of Hessian approximation used, the number of vectors stored for the approximation and the initialization/scaling of the approximation. These options can be configured using the **-tao_lmvm_mat_lmvm** prefix. For further detail, we refer the reader to the MATLMVM matrix type definitions in the PETSc Manual.

The LMVM algorithm also allows the user to define a custom initial Hessian matrix \(H_{0,k}\) through the interface function TaoLMVMSetH0(). This user-provided initialization overrides any other scalar or diagonal initialization inherent to the LMVM approximation. The provided \(H_{0,k}\) must be a PETSc Mat type object that represents a positive-definite matrix. The approximation prefers MatSolve() if the provided matrix has MATOP_SOLVE implemented. Otherwise, MatMult() is used in a KSP solve to perform the inversion of the user-provided initial Hessian.
In applications where `TaoSolve()` on the LMVM algorithm is repeatedly called to solve similar or related problems, `-tao_lmvm_recycle` flag can be used to prevent resetting the LMVM approximation between subsequent solutions. This recycling also avoids one extra function and gradient evaluation, instead re-using the values already computed at the end of the previous solution.

This algorithm will be deprecated in the next version and replaced by the bounded quasi-Newton Line Search (BQNLS) algorithm that can solve both bound constrained and unconstrained problems.

**Nonlinear Conjugate Gradient Method**

The nonlinear conjugate gradient method can be viewed as an extension of the conjugate gradient method for solving symmetric, positive-definite linear systems of equations. This algorithm requires only function and gradient evaluations as well as a line search. The TAO implementation uses a Moré-Thuente line search to obtain the step length. The nonlinear conjugate gradient method can be selected by using the TAO solver `tao_cg`. For the best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

Five variations are currently supported by the TAO implementation: the Fletcher-Reeves method, the Polak-Ribière method, the Polak-Ribière-Plus method [NW99], the Hestenes-Stiefel method, and the Dai-Yuan method. These conjugate gradient methods can be specified by using the command line argument `-tao_cg_type <fr,pr,prp,hs,dy>`, respectively. The default value is `prp`.

The conjugate gradient method incorporates automatic restarts when successive gradients are not sufficiently orthogonal. TAO measures the orthogonality by dividing the inner product of the gradient at the current point and the gradient at the previous point by the square of the Euclidean norm of the gradient at the current point. When the absolute value of this ratio is greater than `eta`, the algorithm restarts using the gradient direction. The parameter `eta` can be set by using the command line argument `-tao_cg_eta <real>`, 0.1 is the default value.

**Newton Line Search Method**

The Newton line search method solves the symmetric system of equations

\[ H_k d_k = -g_k \]

to obtain a step \( d_k \), where \( H_k \) is the Hessian of the objective function at \( x_k \) and \( g_k \) is the gradient of the objective function at \( x_k \). For problems where the Hessian matrix is indefinite, the perturbed system of equations

\[ (H_k + \rho_k I) d_k = -g_k \]

is solved to obtain the direction, where \( \rho_k \) is a positive constant. If the direction computed is not a descent direction, the (scaled) steepest descent direction is used instead. Having obtained the direction, a Moré-Thuente line search is applied to obtain a step length, \( \tau_k \), that approximately solves the one-dimensional optimization problem

\[ \min_{\tau} f(x_k + \tau d_k). \]

The Newton line search method can be selected by using the TAO solver `tao_nls`. The options available for this solver are listed in Table 2.18. For the best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.
The system of equations is approximately solved by applying the conjugate gradient method, Nash conjugate...
With the command line argument -tao_nls_ksp_type <cg,nash,stcg,gltr,gmres,...>: stcg is the default. See the PETSc manual for further information on changing the behavior of the linear system solvers.

A good preconditioner reduces the number of iterations required to solve the linear system of equations. For the conjugate gradient methods and generalized Lanczos method, this preconditioner must be symmetric and positive definite. The available options are to use no preconditioner, the absolute value of the diagonal of the Hessian matrix, a limited-memory BFGS approximation to the Hessian matrix, or one of the other preconditioners provided by the PETSc package. These preconditioners are specified by the command line arguments -tao_nls_pc_type <none,jacobi,icc,ilu,lmvm>, respectively. The default is the lmvm preconditioner, which uses a BFGS approximation of the inverse Hessian. See the PETSc manual for further information on changing the behavior of the preconditioners.

The perturbation $\rho_k$ is added when the direction returned by the Krylov subspace method is not a descent direction, the Krylov method diverged due to an indefinite preconditioner or matrix, or a direction of negative curvature was found. In the last two cases, if the step returned is a descent direction, it is used during the line search. Otherwise, a steepest descent direction is used during the line search. The perturbation is decreased as long as the Krylov subspace method reports success and increased if further problems are encountered. There are three cases: initializing, increasing, and decreasing the perturbation. These cases are described below.

1. If $\rho_k$ is zero and a problem was detected with either the direction or the Krylov subspace method, the perturbation is initialized to

$$\rho_{k+1} = \text{median} \{ \text{imin}, \text{imac} \ast \|g(x_k)\|, \text{imax} \},$$

where $g(x_k)$ is the gradient of the objective function and imin is set with the command line argument -tao_nls_imin <real> with a default value of $10^{-4}$, imac by -tao_nls_imac with a default value of 0.1, and imax by -tao_nls_imax with a default value of 100. When using the gltr method to solve the system of equations, an estimate of the minimum eigenvalue $\lambda_1$ of the Hessian matrix is available. This value is used to initialize the perturbation to $\rho_{k+1} = \max \{ \rho_k, -\lambda_1 \}$ in this case.

2. If $\rho_k$ is nonzero and a problem was detected with either the direction or Krylov subspace method, the perturbation is increased to

$$\rho_{k+1} = \min \{ \text{pmax}, \max \{ \text{pgfac} \ast \rho_k, \text{pmgfac} \ast \|g(x_k)\| \} \},$$

where $g(x_k)$ is the gradient of the objective function and pgfac is set with the command line argument -tao_nls_pgfac with a default value of 10, pmgfac by -tao_nls_pmgfac with a default value of 0.1, and pmax by -tao_nls_pmax with a default value of 100.

3. If $\rho_k$ is nonzero and no problems were detected with either the direction or Krylov subspace method, the perturbation is decreased to

$$\rho_{k+1} = \min \{ \text{psfac} \ast \rho_k, \text{pmsfac} \ast \|g(x_k)\| \},$$

where $g(x_k)$ is the gradient of the objective function, psfac is set with the command line argument -tao_nls_psfac with a default value of 0.4, and pmsfac is set by -tao_nls_pmsfac with a default value of 0.1. Moreover, if $\rho_{k+1} < \text{pmin}$, then $\rho_{k+1} = 0$, where pmin is set with the command line argument -tao_nls_pmin and has a default value of $10^{-12}$.

Near a local minimizer to the unconstrained optimization problem, the Hessian matrix will be positive-semidefinite; the perturbation will shrink toward zero, and one would eventually observe a superlinear convergence rate.

When using nash, stcg, or gltr to solve the linear systems of equation, a trust-region radius needs to be initialized and updated. This trust-region radius simultaneously limits the size of the step computed and
reduces the number of iterations of the conjugate gradient method. The method for initializing the trust-region radius is set with the command line argument -tao_nls_init_type <constant,direction, interpolation>; interpolation, which chooses an initial value based on the interpolation scheme found in [CGT00], is the default. This scheme performs a number of function and gradient evaluations to determine a radius such that the reduction predicted by the quadratic model along the gradient direction coincides with the actual reduction in the nonlinear function. The iterate obtaining the best objective function value is used as the starting point for the main line search algorithm. The constant method initializes the trust-region radius by using the value specified with the -tao_trust0 <real> command line argument, where the default value is 100. The direction technique solves the first quadratic optimization problem by using a standard conjugate gradient method and initializes the trust region to \( ||s_0|| \).

The method for updating the trust-region radius is set with the command line argument -tao_nls_update_type <step, reduction, interpolation>; step is the default. The step method updates the trust-region radius based on the value of \( \tau_k \). In particular,

\[
\Delta_{k+1} = \begin{cases} 
\omega_1 \min(\Delta_k, ||d_k||) & \text{if } \tau_k \in [0, \nu_1) \\
\omega_2 \min(\Delta_k, ||d_k||) & \text{if } \tau_k \in [\nu_1, \nu_2) \\
\omega_3 \Delta_k & \text{if } \tau_k \in [\nu_2, \nu_3) \\
\max(\Delta_k, \omega_4 ||d_k||) & \text{if } \tau_k \in [\nu_3, \nu_4) \\
\max(\Delta_k, \omega_5 ||d_k||) & \text{if } \tau_k \in [\nu_4, \infty),
\end{cases}
\]

where \( 0 < \omega_1 < \omega_2 < \omega_3 = 1 < \omega_4 < \omega_5 \) and \( 0 < \nu_1 < \nu_2 < \nu_3 < \nu_4 \) are constants. The reduction method computes the ratio of the actual reduction in the objective function to the reduction predicted by the quadratic model for the full step, \( \kappa_k = \frac{f(x_k) - f(x_k + d_k)}{q(x_k) - q(x_k + d_k)} \), where \( q_k \) is the quadratic model. The radius is then updated as

\[
\Delta_{k+1} = \begin{cases} 
\alpha_1 \min(\Delta_k, ||d_k||) & \text{if } \kappa_k \in (-\infty, \eta_1) \\
\alpha_2 \min(\Delta_k, ||d_k||) & \text{if } \kappa_k \in [\eta_1, \eta_2) \\
\alpha_3 \Delta_k & \text{if } \kappa_k \in [\eta_2, \eta_3) \\
\max(\Delta_k, \alpha_4 ||d_k||) & \text{if } \kappa_k \in [\eta_3, \eta_4) \\
\max(\Delta_k, \alpha_5 ||d_k||) & \text{if } \kappa_k \in [\eta_4, \infty),
\end{cases}
\]

where \( 0 < \alpha_1 < \alpha_2 < \alpha_3 = 1 < \alpha_4 < \alpha_5 \) and \( 0 < \eta_1 < \eta_2 < \eta_3 < \eta_4 \) are constants. The interpolation method uses the same interpolation mechanism as in the initialization to compute a new value for the trust-region radius.

This algorithm will be deprecated in the next version and replaced by the bounded Newton Line Search (BNLS) algorithm that can solve both bound constrained and unconstrained problems.

**Newton Trust-Region Method**

The Newton trust-region method solves the constrained quadratic programming problem

\[
\min_d \quad \frac{1}{2} d^T H_k d + g_k^T d \\
\text{subject to} \quad ||d|| \leq \Delta_k
\]

to obtain a direction \( d_k \), where \( H_k \) is the Hessian of the objective function at \( x_k \), \( g_k \) is the gradient of the objective function at \( x_k \), and \( \Delta_k \) is the trust-region radius. If \( x_k + d_k \) sufficiently reduces the nonlinear objective function, then the step is accepted, and the trust-region radius is updated. However, if \( x_k + d_k \) does not sufficiently reduce the nonlinear objective function, then the step is rejected, the trust-region radius is reduced, and the quadratic program is re-solved by using the updated trust-region radius. The Newton trust-region method can be set by using the TAO solver tao_ntr. The options available for this solver are listed in Table 2.19. For the best efficiency, function, gradient and Jacobian evaluations should be performed separately when using this algorithm.
### Table 2.19: Summary of ntr options

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- tao_ntr_ksp_type</td>
<td>nash, stcg, gltr</td>
<td>stcg</td>
<td>Type of Krylov subspace method to use when solving linear system</td>
</tr>
<tr>
<td>- tao_ntr_pc_type</td>
<td>none, jacobi, icc, lu, ilu, lmvm</td>
<td>lmvm</td>
<td>Type of preconditioner to use when solving linear system</td>
</tr>
<tr>
<td>- tao_ntr_init_type</td>
<td>constant, direction, interpolation</td>
<td>interpolation</td>
<td>Method used to initialize trust-region radius</td>
</tr>
<tr>
<td>- tao_ntr_mu1_i</td>
<td>real</td>
<td>0.35</td>
<td>$\mu_1$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_mu2_i</td>
<td>real</td>
<td>0.50</td>
<td>$\mu_2$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_gamma1_i</td>
<td>real</td>
<td>0.0625</td>
<td>$\gamma_1$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_gamma2_i</td>
<td>real</td>
<td>0.50</td>
<td>$\gamma_2$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_gamma3_i</td>
<td>real</td>
<td>2.00</td>
<td>$\gamma_3$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_gamma4_i</td>
<td>real</td>
<td>5.00</td>
<td>$\gamma_4$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_theta_i</td>
<td>real</td>
<td>0.25</td>
<td>$\theta$ in interpolation init</td>
</tr>
<tr>
<td>- tao_ntr_update_type</td>
<td>reduction, interpolation</td>
<td>reduction</td>
<td>Method used to update trust-region radius</td>
</tr>
<tr>
<td>- tao_ntr_eta1</td>
<td>real</td>
<td>:</td>
<td>$\eta_1$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_eta2</td>
<td>real</td>
<td>0.25</td>
<td>$\eta_2$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_eta3</td>
<td>real</td>
<td>0.50</td>
<td>$\eta_3$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_eta4</td>
<td>real</td>
<td>0.90</td>
<td>$\eta_4$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_alpha1</td>
<td>real</td>
<td>0.25</td>
<td>$\alpha_1$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_alpha2</td>
<td>real</td>
<td>0.50</td>
<td>$\alpha_2$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_alpha3</td>
<td>real</td>
<td>1.00</td>
<td>$\alpha_3$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_alpha4</td>
<td>real</td>
<td>2.00</td>
<td>$\alpha_4$ in reduction update</td>
</tr>
<tr>
<td>- tao_ntr_alpha5</td>
<td>real</td>
<td>4.00</td>
<td>$\alpha_5$ in reduction update</td>
</tr>
<tr>
<td>-tao_ntr_mu1</td>
<td>real</td>
<td>0.10</td>
<td>$\mu_1$ in interpolation update</td>
</tr>
<tr>
<td>-tao_ntr_mu2</td>
<td>real</td>
<td>0.50</td>
<td>$\mu_2$ in interpolation update</td>
</tr>
<tr>
<td>- tao_ntr_gamma1</td>
<td>real</td>
<td>0.25</td>
<td>$\gamma_1$ in interpolation update</td>
</tr>
<tr>
<td>- tao_ntr_gamma2</td>
<td>real</td>
<td>0.50</td>
<td>$\gamma_2$ in interpolation update</td>
</tr>
<tr>
<td>- tao_ntr_gamma3</td>
<td>real</td>
<td>2.00</td>
<td>$\gamma_3$ in interpolation update</td>
</tr>
<tr>
<td>- tao_ntr_gamma4</td>
<td>real</td>
<td>4.00</td>
<td>$\gamma_4$ in interpolation update</td>
</tr>
</tbody>
</table>

2.8. TAO: Optimization Solvers

| tao_ntr_theta | 0.05 | $\theta$ in interpolation update |
The quadratic optimization problem is approximately solved by applying the Nash or Steihaug-Toint conjugate gradient methods or the generalized Lanczos method to the symmetric system of equations $H_k d = -g_k$. The method used to solve the system of equations is specified with the command line argument -tao_ntr_ksp_type <nash, stcg, gltr>; stcg is the default. See the PETSc manual for further information on changing the behavior of these linear system solvers.

A good preconditioner reduces the number of iterations required to compute the direction. For the Nash and Steihaug-Toint conjugate gradient methods and generalized Lanczos method, this preconditioner must be symmetric and positive definite. The available options are to use no preconditioner, the absolute value of the diagonal of the Hessian matrix, a limited-memory BFGS approximation to the Hessian matrix, or one of the other preconditioners provided by the PETSc package. These preconditioners are specified by the command line argument -tao_ntr_pc_type <none, jaccobi, icc, ilu, lmvn>, respectively. The default is the lmvn preconditioner. See the PETSc manual for further information on changing the behavior of the preconditioners.

The method for computing an initial trust-region radius is set with the command line arguments -tao_ntr_init_type <constant, direction, interpolation>; interpolation, which chooses an initial value based on the interpolation scheme found in [CGT00], is the default. This scheme performs a number of function and gradient evaluations to determine a radius such that the reduction predicted by the quadratic model along the gradient direction coincides with the actual reduction in the nonlinear function. The iterate obtaining the best objective function value is used as the starting point for the main trust-region algorithm. The constant method initializes the trust-region radius by using the value specified with the -tao_trust0 <real> command line argument, where the default value is 100. The direction technique solves the first quadratic optimization problem by using a standard conjugate gradient method and initializes the trust region to $\|s_0\|$.

The method for updating the trust-region radius is set with the command line arguments -tao_ntr_update_type <reduction, interpolation>; reduction is the default. The reduction method computes the ratio of the actual reduction in the objective function to the reduction predicted by the quadratic model for the full step, $\kappa_k = \frac{f(x_k) - f(x_k + d_k)}{q(x_k) - q(x_k + d_k)}$, where $q_k$ is the quadratic model. The radius is then updated as

$$
\Delta_{k+1} = \begin{cases} 
\alpha_1 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in (-\infty, \eta_1) \\
\alpha_2 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in [\eta_1, \eta_2) \\
\alpha_3 \Delta_k & \text{if } \kappa_k \in [\eta_2, \eta_3) \\
\max(\Delta_k, \alpha_4 \|d_k\|) & \text{if } \kappa_k \in [\eta_3, \eta_4) \\
\max(\Delta_k, \alpha_5 \|d_k\|) & \text{if } \kappa_k \in [\eta_4, \infty), 
\end{cases}
$$

where $0 < \alpha_1 < \alpha_2 < \alpha_3 = 1 < \alpha_4 < \alpha_5$ and $0 < \eta_1 < \eta_2 < \eta_3 < \eta_4$ are constants. The interpolation method uses the same interpolation mechanism as in the initialization to compute a new value for the trust-region radius.

This algorithm will be deprecated in the next version and replaced by the bounded Newton Trust Region (BNTR) algorithm that can solve both bound constrained and unconstrained problems.

**BMRM**

The Bundle Method for Regularized Risk Minimization (BMRM) is a numerical approach to optimizing an unconstrained objective in the form of $f(x) + 0.5 \lambda \|x\|^2$. Here $f$ is a convex function that is finite on the whole space. $\lambda$ is a positive weight parameter, and $\|x\|$ is the Euclidean norm of $x$. The algorithm only requires a routine which, given an $x$, returns the value of $f(x)$ and the gradient of $f$ at $x$. 
OWL-QN

The Orthant-Wise Limited-memory Quasi-Newton algorithm (OWL-QN) [AG07] is a numerical approach to optimizing an unconstrained objective in the form of \( f(x) + \lambda \|x\|_1 \). Here \( f \) is a convex and differentiable function, \( \lambda \) is a positive weight parameter, and \( \|x\|_1 \) is the L1 norm of \( x \): \( \sum_i |x_i| \). The algorithm only requires evaluating the value of \( f \) and its gradient.

Bound-Constrained Optimization

Bound-constrained optimization algorithms solve optimization problems of the form

\[
\begin{align*}
\min_{x} & \quad f(x) \\
\text{subject to} & \quad l \leq x \leq u.
\end{align*}
\]

These solvers use the bounds on the variables as well as objective function, gradient, and possibly Hessian information.

For any unbounded variables, the bound value for the associated index can be set to `PETSC_INFINITY` for the upper bound and `PETSC_NINFINITY` for the lower bound. If all bounds are set to infinity, then the bounded algorithms are equivalent to their unconstrained counterparts.

Before introducing specific methods, we will first define two projection operations used by all bound constrained algorithms.

- Gradient projection:
  \[
  \mathcal{P}(g) = \begin{cases} 
  0 & \text{if } (x \leq l_i \land g_i > 0) \lor (x \geq u_i \land g_i < 0) \\
  g_i & \text{otherwise}
  \end{cases}
  \]

- Bound projection:
  \[
  \mathcal{B}(x) = \begin{cases} 
  l_i & \text{if } x_i < l_i \\
  u_i & \text{if } x_i > u_i \\
  x_i & \text{otherwise}
  \end{cases}
  \]

Bounded Newton-Krylov Methods

TAO features three bounded Newton-Krylov (BNK) class of algorithms, separated by their globalization methods: projected line search (BNLS), trust region (BNTR), and trust region with a projected line search fall-back (BNTL). They are available via the TAO solvers `TAOBNLS`, `TAOBNTR`, and `TAOBNTL`, respectively, or the `-tao_type bnls/bntr/bntl` flag.

The BNK class of methods use an active-set approach to solve the symmetric system of equations, 

\[
H_k p_k = -g_k,
\]

only for inactive variables in the interior of the bounds. The active-set estimation is based on Bertsekas [Ber82] with the following variable index categories:

- lower bounded: \( \mathcal{L}(x) = \{ i : x_i \leq l_i + \epsilon \land g(x)_i > 0 \} \)
- upper bounded: \( \mathcal{U}(x) = \{ i : x_i \geq u_i + \epsilon \land g(x)_i < 0 \} \)
- fixed: \( \mathcal{F}(x) = \{ i : l_i = u_i \} \)
- active-set: \( \mathcal{A}(x) = \mathcal{L}(x) \cup \mathcal{U}(x) \cup \mathcal{F}(x) \)
- inactive-set: \( \mathcal{I}(x) = \{ 1, 2, \ldots, n \} \setminus \mathcal{A}(x) \)
At each iteration, the bound tolerance is estimated as $\epsilon_{k+1} = \min(\epsilon_k, \|w_k\|_2)$ with $w_k = x_k - \nabla(x_k - \beta D_k g_k)$, where the diagonal matrix $D_k$ is an approximation of the Hessian inverse $H_k^{-1}$. The initial bound tolerance $\epsilon_0$ and the step length $\beta$ have default values of 0.001 and can be adjusted using -tao_bnk_as_tol and -tao_bnk_as_step flags, respectively. The active-set estimation can be disabled using the option -tao_bnk_as_type none, in which case the algorithm simply uses the current iterate with no bound tolerances to determine which variables are actively bounded and which are free.

BNK algorithms invert the reduced Hessian using a Krylov iterative method. Trust-region conjugate gradient methods (KSPNASH, KSPSTCG, and KSPGLTR) are required for the BNTR and BNTL algorithms, and recommended for the BNLS algorithm. The preconditioner type can be changed using the -tao_bnk_pc_type none/ilu/jacobi/lmvm option, which is also the default, preconditions the Krylov solution with a MATLMVM matrix. The remaining supported preconditioner types are default PETSc types. If Jacobi is selected, the diagonal values are safeguarded to be positive. Ilu and Jacobi options produce good results for problems with dense Hessians. The LMVM and Jacobi preconditioners are also used as the approximate inverse-Hessian in the active-set estimation. If neither are available, or if the Hessian matrix does not have MATOP_GET_DIAGONAL defined, then the active-set estimation falls back onto using an identity matrix in place of $D_k$ (this is equivalent to estimating the active-set using a gradient descent step).

A special option is available to accelerate the convergence of the BNK algorithms by taking a finite number of BNCG iterations at each Newton iteration. By default, the number of BNCG iterations is set to zero and the algorithms do not take any BNCG steps. This can be changed using the option flag -tao_bnk_max_cg_its <i>. While this reduces the number of Newton iterations, in practice it simply trades off the Hessian evaluations in the BNK solver for more function and gradient evaluations in the BNCG solver. However, it may be useful for certain types of problems where the Hessian evaluation is disproportionately more expensive than the objective function or its gradient.

**Bounded Newton Line Search (BNLS)**

BNLS safeguards the Newton step by falling back onto a BFGS, scaled gradient, or gradient steps based on descent direction verifications. For problems with indefinite Hessian matrices, the step direction is calculated using a perturbed system of equations,

$$(H_k + \rho_k I)p_k = -g_k,$$

where $\rho_k$ is a dynamically adjusted positive constant. The step is globalized using a projected More-Thuente line search. If a trust-region conjugate gradient method is used for the Hessian inversion, the trust radius is modified based on the line search step length.
Bounded Newton Trust Region (BNTR)

BNTR globalizes the Newton step using a trust region method based on the predicted versus actual reduction in the cost function. The trust radius is increased only if the accepted step is at the trust region boundary. The reduction check features a safeguard for numerical values below machine epsilon, scaled by the latest function value, where the full Newton step is accepted without modification.

Bounded Newton Trust Region with Line Search Fall-back (BNTL)

BNTL safeguards the trust-region globalization such that a line search is used in the event that the step is initially rejected by the predicted versus actual decrease comparison. If the line search fails to find a viable step length for the Newton step, it falls back onto a scaled gradient or a gradient descent step. The trust radius is then modified based on the line search step length.

Bounded Nonlinear Conjugate Gradient

BNCG extends the unconstrained nonlinear conjugate gradient algorithm to bound constraints via gradient projections and a bounded More-Thuente line search.

Like its unconstrained counterpart, BNCG offers gradient descent and a variety of CG updates: Fletcher-Reeves, Polak-Ribiére, Polak-Ribiére-Plus, Hestenes-Stiefel, Dai-Yuan, Hager-Zhang, Dai-Kou, Kou-Dai, and the Self-Scaling Memoryless (SSML) BFGS, DFP, and Broyden methods. These methods can be specified by using the command line argument `-tao_bncg_type <gd,fr,pr,prp,hs,dy,hz,dk,kd,ssml_bfgs,ssml_dfp,ssml_brddn>`, respectively. The default value is `ssml_bfgs`. We have scalar preconditioning for these methods, and it is controlled by the flag `tao_bncg_alpha`. To disable rescaling, use $\alpha = -1.0$, otherwise $\alpha \in [0,1]$. BNCG is available via the TAO solver TAOBNCG or the `-tao_type bncg` flag.

Some individual methods also contain their own parameters. The Hager-Zhang and Dou-Kai methods have a parameter that determines the minimum amount of contribution the previous search direction gives to the next search direction. The flags are `-tao_bncg_hz_eta` and `-tao_bncg_dk_eta`, and by default are set to 0.4 and 0.5 respectively. The Kou-Dai method has multiple parameters. `-tao_bncg_zeta` serves the same purpose as the previous two; set to 0.1 by default. There is also a parameter to scale the contribution of $y_k \equiv \nabla f(x_k) - \nabla f(x_{k-1})$ in the search direction update. It is controlled by `-tao_bncg_xi`, and is equal to 1.0 by default. There are also times where we want to maximize the descent as measured by $\nabla f(x_k)^T d_k$, and that may be done by using a negative value of $\zeta$; this achieves better performance when not using the diagonal preconditioner described next. This is enabled by default, and is controlled by `-tao_bncg_neg_xi`. Finally, the Broyden method has its convex combination parameter, set with `-tao_bncg_theta`. We have this as 1.0 by default, i.e. it is by default the BFGS method. One can also individually tweak the BFGS and DFP contributions using the multiplicative constants `-tao_bncg_scale`; both are set to 1 by default.

All methods can be scaled using the parameter `-tao_bncg_alpha`, which continuously varies in $[0,1]$. The default value is set depending on the method from initial testing.

BNCG also offers a special type of method scaling. It employs Broyden diagonal scaling as an option for its CG methods, turned on with the flag `-tao_bncg_diag_scaling`. Formulations for both the forward (regular) and inverse Broyden methods are developed, controlled by the flag `-tao_bncg_mat_lmvm_forward`. It is set to True by default. Whether one uses the forward or inverse formulations depends on the method being used. For example, in our preliminary computations, the forward formulation works better for the SSML_BFGS method, but the inverse formulation works better for the Hestenes-Stiefel method. The convex combination parameter for the Broyden scaling is controlled by `-tao_bncg_mat_lmvm_theta`, and is 0 by default. We also employ rescaling of the Broyden diagonal, which aids the linesearch immensely. The rescaling parameter is controlled by `-tao_bncg_mat_lmvm_alpha`, and should be $\in [0,1]$. One can disable rescaling of the Broyden diagonal entirely by setting `-tao_bncg_mat_lmvm_sigma_hist 0`.

2.8. TAO: Optimization Solvers
One can also supply their own preconditioner, serving as a Hessian initialization to the above diagonal scaling. The appropriate user function in the code is `TaoBNCGSetH0(tao, H0)` where `H0` is the user-defined `Mat` object that serves as a preconditioner. For an example of similar usage, see `tao/tutorials/ex3.c`.

The active set estimation uses the Bertsekas-based method described in `Bounded Newton-Krylov Methods`, which can be deactivated using `-tao_bncg_as_type none`, in which case the algorithm will use the current iterate to determine the bounded variables with no tolerancing and no look-ahead step. As in the BNK algorithm, the initial bound tolerance and estimator step length used in the Bertsekas method can be set via `-tao_bncg_as_tol` and `-tao_bncg_as_step`, respectively.

In addition to automatic scaled gradient descent restarts under certain local curvature conditions, we also employ restarts based on a check on descent direction such that $\nabla f(x_k)^T d_k \in [-10^{11}, -1^{-9}]$. Furthermore, we allow for a variety of alternative restart strategies, all disabled by default. The `-tao_bncg_unscaled_restart` flag allows one to disable rescaling of the gradient for gradient descent steps. The `-tao_bncg_spaced_restart` flag tells the solver to restart every $Mn$ iterations, where $n$ is the problem dimension and $M$ is a constant determined by `-tao_bncg_min_restart_num` and is 6 by default. We also have dynamic restart strategies based on checking if a function is locally quadratic; if so, go do a gradient descent step. The flag is `-tao_bncg_dynamic_restart`, disabled by default since the CG solver usually does better in those cases anyway. The minimum number of quadratic-like steps before a restart is set using `-tao_bncg_min_quad` and is 6 by default.

**Trust-Region Newton Method**

The TRON [LMore99] algorithm is an active-set method that uses a combination of gradient projections and a preconditioned conjugate gradient method to minimize an objective function. Each iteration of the TRON algorithm requires function, gradient, and Hessian evaluations. In each iteration, the algorithm first applies several conjugate gradient iterations. After these iterates, the TRON solver momentarily ignores the variables that equal one of its bounds and applies a preconditioned conjugate gradient method to a quadratic model of the remaining set of free variables.

The TRON algorithm solves a reduced linear system defined by the rows and columns corresponding to the variables that lie between the upper and lower bounds. The TRON algorithm applies a trust region to the conjugate gradients to ensure convergence. The initial trust-region radius can be set by using the command `TaoSetInitialTrustRegionRadius()`, and the current trust region size can be found by using the command `TaoGetCurrentTrustRegionRadius()`. The initial trust region can significantly alter the rate of convergence for the algorithm and should be tuned and adjusted for optimal performance.

**Bound-constrained Limited-Memory Variable-Metric Method**

BLMVM is a limited-memory, variable-metric method and is the bound-constrained variant of the LMVM method for unconstrained optimization. It uses projected gradients to approximate the Hessian, eliminating the need for Hessian evaluations. The method can be set by using the TAO solver `tao_blmvm`. For more details, please see the LMVM section in the unconstrained algorithms as well as the LMVM matrix documentation in the PETSc manual.

This algorithm will be deprecated in the next version in favor of the bounded quasi-Newton line search (BQNLS) algorithm.
Bounded Quasi-Newton-Krylov

BQNK algorithms use the BNK infrastructure, but replace the exact Hessian with a quasi-Newton approximation. The matrix-free forward product operation based on quasi-Newton update formulas are used in conjunction with Krylov solvers to compute step directions. The quasi-Newton inverse application is used to precondition the Krylov solution, and typically helps converge to a step direction in $O(10)$ iterations. This approach is most useful with quasi-Newton update types such as Symmetric Rank-1 that cannot strictly guarantee positive-definiteness. The BNLS framework with Hessian shifting, or the BNTR framework with trust region safeguards, can successfully compensate for the Hessian approximation becoming indefinite.

Similar to the full Newton-Krylov counterpart, BQNK algorithms come in three forms separated by the globalization technique: line search (BQNKLS), trust region (BQNKTR) and trust region w/ line search fall-back (BQNKTL). These algorithms are available via `tao_type <bqnkls, bqnktr, bqnktl>`.

Bounded Quasi-Newton Line Search (BQNLBS)

BQNLS algorithm uses the BNLS infrastructure, but replaces the step calculation with a direct inverse application of the approximate Hessian based on quasi-Newton update formulas. No Krylov solver is used in the solution, and therefore the quasi-Newton method chosen must guarantee a positive-definite Hessian approximation. This algorithm is available via `tao_type bqnls`.

PDE-Constrained Optimization

TAO solves PDE-constrained optimization problems of the form

$$\min_{u,v} \quad f(u,v)$$
subject to \quad $g(u,v) = 0$,

where the state variable $u$ is the solution to the discretized partial differential equation defined by $g$ and parametrized by the design variable $v$, and $f$ is an objective function. The Lagrange multipliers on the constraint are denoted by $y$. This method is set by using the linearly constrained augmented Lagrangian TAO solver `tao_lcl`.

We make two main assumptions when solving these problems: the objective function and PDE constraints have been discretized so that we can treat the optimization problem as finite dimensional and $\nabla_u g(u,v)$ is invertible for all $u$ and $v$.

Linearly-Constrained Augmented Lagrangian Method

Given the current iterate $(u_k, v_k, y_k)$, the linearly constrained augmented Lagrangian method approximately solves the optimization problem

$$\min_{u,v} \quad \hat{f}_k(u,v)$$
subject to \quad $A_k(u - u_k) + B_k(v - v_k) + g_k = 0$,

where $A_k = \nabla_u g(u_k, v_k), B_k = \nabla_v g(u_k, v_k)$, and $g_k = g(u_k, v_k)$ and

$$\hat{f}_k(u,v) = f(u,v) - g(u,v)^T y_k + \frac{\rho_k}{2} \|g(u,v)\|^2$$

is the augmented Lagrangian function. This optimization problem is solved in two stages. The first computes the Newton direction and finds a feasible point for the linear constraints. The second computes a reduced-space direction that maintains feasibility with respect to the linearized constraints and improves the augmented Lagrangian merit function.
Newton Step

The Newton direction is obtained by fixing the design variables at their current value and solving the linearized constraint for the state variables. In particular, we solve the system of equations

\[ A_k du = -g_k \]

to obtain a direction \( du \). We need a direction that provides sufficient descent for the merit function

\[ \frac{1}{2} \| g(u, v) \|^2. \]

That is, we require \( g_k^T A_k du < 0 \).

If the Newton direction is a descent direction, then we choose a penalty parameter \( \rho_k \) so that \( du \) is also a sufficient descent direction for the augmented Lagrangian merit function. We then find \( \alpha \) to approximately minimize the augmented Lagrangian merit function along the Newton direction.

\[ \min_{\alpha \geq 0} \tilde{f}_k(u_k + \alpha du, v_k). \]

We can enforce either the sufficient decrease condition or the Wolfe conditions during the search procedure. The new point,

\[ u_{k+\frac{1}{2}} = u_k + \alpha_k du \]
\[ v_{k+\frac{1}{2}} = v_k, \]

satisfies the linear constraint

\[ A_k(u_{k+\frac{1}{2}} - u_k) + B_k(v_{k+\frac{1}{2}} - v_k) + \alpha_k g_k = 0. \]

If the Newton direction computed does not provide descent for the merit function, then we can use the steepest descent direction \( du = -A_k^T g_k \) during the search procedure. However, the implication that the intermediate point approximately satisfies the linear constraint is no longer true.

Modified Reduced-Space Step

We are now ready to compute a reduced-space step for the modified optimization problem:

\[ \min_{u, v} \tilde{f}_k(u, v) \]
\[ \text{subject to } A_k(u - u_k) + B_k(v - v_k) + \alpha_k g_k = 0. \]

We begin with the change of variables

\[ \min_{du, dv} \tilde{f}_k(u_k + du, v_k + dv) \]
\[ \text{subject to } A_k du + B_k dv + \alpha_k g_k = 0 \]

and make the substitution

\[ du = -A_k^{-1}(B_k dv + \alpha_k g_k). \]

Hence, the unconstrained optimization problem we need to solve is

\[ \min_{dv} \tilde{f}_k(u_k - A_k^{-1}(B_k dv + \alpha_k g_k), v_k + dv), \]

which is equivalent to

\[ \min_{dv} \tilde{f}_k(u_{k+\frac{1}{2}} - A_k^{-1} B_k dv, v_{k+\frac{1}{2}} + dv). \]
We apply one step of a limited-memory quasi-Newton method to this problem. The direction is obtained by solving the quadratic problem
\[
\min_{dv} \frac{1}{2} dv^T \tilde{H}_k dv + \tilde{g}_{k+\frac{1}{2}}^T dv,
\]
where \( \tilde{H}_k \) is the limited-memory quasi-Newton approximation to the reduced Hessian matrix, a positive-definite matrix, and \( \tilde{g}_{k+\frac{1}{2}} \) is the reduced gradient.

The reduced gradient is obtained from one linearized adjoint solve
\[
y_{k+\frac{1}{2}} = A_k^T c_{k+\frac{1}{2}}
\]
and some linear algebra
\[
\tilde{g}_{k+\frac{1}{2}} = d_{k+\frac{1}{2}} + y_{k+\frac{1}{2}}^T B_k.
\]
Because the Hessian approximation is positive definite and we know its inverse, we obtain the direction
\[
dv = -H_k^{-1} \tilde{g}_{k+\frac{1}{2}}
\]
and recover the full-space direction from one linearized forward solve,
\[
du = -A_k^{-1} B_k dv.
\]
Having the full-space direction, which satisfies the linear constraint, we now approximately minimize the augmented Lagrangian merit function along the direction.
\[
\min_{\beta \geq 0} \tilde{f}_k(u_{k+\frac{1}{2}} + \beta du, v_{k+\frac{1}{2}} + \beta dv)
\]
We enforce the Wolfe conditions during the search procedure. The new point is
\[
\begin{align*}
u_{k+1} &= u_{k+\frac{1}{2}} + \beta_k du \\
v_{k+1} &= v_{k+\frac{1}{2}} + \beta_k dv.
\end{align*}
\]
The reduced gradient at the new point is computed from
\[
\begin{align*}
y_{k+1} &= A_{k+1}^T c_{k+1} \\
\tilde{g}_{k+1} &= d_{k+1} - y_{k+1}^T B_k,
\end{align*}
\]
where \( c_{k+1} = \nabla u \tilde{f}_k(u_{k+1}, v_{k+1}) \) and \( d_{k+1} = \nabla v \tilde{f}_k(u_{k+1}, v_{k+1}) \). The multipliers \( y_{k+1} \) become the multipliers used in the next iteration of the code. The quantities \( v_{k+\frac{1}{2}}, v_{k+1}, \tilde{g}_{k+\frac{1}{2}}, \) and \( \tilde{g}_{k+1} \) are used to update \( H_k \) to obtain the limited-memory quasi-Newton approximation to the reduced Hessian matrix used in the next iteration of the code. The update is skipped if it cannot be performed.

**Nonlinear Least-Squares**

Given a function \( F : \mathbb{R}^n \to \mathbb{R}^m \), the nonlinear least-squares problem minimizes
\[
f(x) = \| F(x) \|^2 = \sum_{i=1}^{m} F_i(x)^2.
\]

The nonlinear equations \( F \) should be specified with the function \text{TaoSetResidual()}.
POUNDerS

One algorithm for solving the least squares problem ((2.5)) when the Jacobian of the residual vector \( F \) is unavailable is the model-based POUNDerS (Practical Optimization Using No Derivatives for sums of Squares) algorithm (\texttt{tao_pounders}). POUNDerS employs a derivative-free trust-region framework as described in [CSV09] in order to converge to local minimizers. An example of this version of POUNDerS applied to a practical least-squares problem can be found in [KortelainenLesinskiMore+10].

Derivative-Free Trust-Region Algorithm

In each iteration \( k \), the algorithm maintains a model \( m_k(x) \), described below, of the nonlinear least squares function \( f \) centered about the current iterate \( x_k \).

If one assumes that the maximum number of function evaluations has not been reached and that \( \| \nabla m_k(x_k) \|_2 > \text{gtol} \), the next point \( x_+ \) to be evaluated is obtained by solving the trust-region subproblem

\[
\min \{ m_k(x) : \| x - x_k \|_p \leq \Delta_k, \}
\]

where \( \Delta_k \) is the current trust-region radius. By default we use a trust-region norm with \( p = \infty \) and solve ((2.6)) with the BLMVM method described in Bound-constrained Limited-Memory Variable-Metric Method. While the subproblem is a bound-constrained quadratic program, it may not be convex and the BQPIP and GPCG methods may not solve the subproblem. Therefore, either BLMVM (the default) or TRON should be used. Note: TRON uses its own internal trust region that may interfere with the infinity-norm trust region used in the model problem ((2.6)).

The residual vector is then evaluated to obtain \( F(x_+) \) and hence \( f(x_+) \). The ratio of actual decrease to predicted decrease,

\[
\rho_k = \frac{f(x_k) - f(x_+)}{m_k(x_k) - m_k(x_+)},
\]

as well as an indicator, \text{valid}, on the model’s quality of approximation on the trust region is then used to update the iterate,

\[
x_{k+1} = \begin{cases} 
  x_+ & \text{if } \rho_k \geq \eta_1 \\
  x_+ & \text{if } 0 < \rho_k < \eta_1 \text{ and valid=true} \\
  x_k & \text{else,}
\end{cases}
\]

and trust-region radius,

\[
\Delta_{k+1} = \begin{cases} 
  \min(\gamma_1 \Delta_k, \Delta_{\text{max}}) & \text{if } \rho_k \geq \eta_1 \text{ and } \| x_+ - x_k \|_p \geq \omega_1 \Delta_k \\
  \gamma_0 \Delta_k & \text{if } \rho_k < \eta_1 \text{ and valid=true} \\
  \Delta_k & \text{else,}
\end{cases}
\]

where \( 0 < \eta_1 < 1, 0 < \gamma_0 < 1 < \gamma_1, 0 < \omega_1 < 1, \) and \( \Delta_{\text{max}} \) are constants.

If \( \rho_k \leq 0 \) and \text{valid} is \text{false}, the iterate and trust-region radius remain unchanged after the above updates, and the algorithm tests whether the direction \( x_+ - x_k \) improves the model. If not, the algorithm performs an additional evaluation to obtain \( F(x_k + d_k) \), where \( d_k \) is a model-improving direction.

The iteration counter is then updated, and the next model \( m_k \) is obtained as described next.
Forming the Trust-Region Model

In each iteration, POUNDerS uses a subset of the available evaluated residual vectors \( \{ F(y_1), F(y_2), \cdots \} \) to form an interpolatory quadratic model of each residual component. The \( m \) quadratic models

\[
q_k^{(i)}(x) = F_i(x_k) + (x - x_k)^T g_k^{(i)} + \frac{1}{2} (x - x_k)^T H_k^{(i)} (x - x_k), \quad i = 1, \ldots, m
\]  

(2.7)

thus satisfy the interpolation conditions

\[
q_k^{(i)}(y_j) = F_i(y_j), \quad i = 1, \ldots, m; \quad j = 1, \ldots, l_k
\]

on a common interpolation set \( \{ y_1, \cdots, y_{l_k} \} \) of size \( l_k \in [n + 1, npmax] \).

The gradients and Hessians of the models in \((eq:models)\) are then used to construct the main model,

\[
m_k(x) = f(x_k) + 2(x - x_k)^T \sum_{i=1}^{m} F_i(x_k) g_k^{(i)} + (x - x_k)^T \sum_{i=1}^{m} \left( g_k^{(i)} (g_k^{(i)})^T + F_i(x_k) H_k^{(i)} \right) (x - x_k).
\]  

(2.8)

The process of forming these models also computes the indicator \texttt{valid} of the model’s local quality.

Parameters

POUNDerS supports the following parameters that can be set from the command line or PETSc options file:

- \texttt{-tao_pounders_delta <delta>} The initial trust-region radius (> 0, real). This is used to determine the size of the initial neighborhood within which the algorithm should look.

- \texttt{-tao_pounders_npmax <npmax>} The maximum number of interpolation points used (\( n + 2 \leq npmax \leq 0.5(n + 1)(n + 2) \)). This input is made available to advanced users. We recommend the default value (\( npmax = 2n + 1 \)) be used by others.

- \texttt{-tao_pounders_gqt} Use the gqt algorithm to solve the subproblem \((2.6)\) (uses \( p = 2 \)) instead of BQPIP.

- \texttt{-pounders_subsolver} If the default BQPIP algorithm is used to solve the subproblem \((2.6)\), the parameters of the subproblem solver can be accessed using the command line options prefix \texttt{-pounders_subsolver_}. For example,

\[
\texttt{-pounders_subsolver_tao_gatol 1.0e-5}
\]

sets the gradient tolerance of the subproblem solver to \( 10^{-5} \).

Additionally, the user provides an initial solution vector, a vector for storing the separable objective function, and a routine for evaluating the residual vector \( F \). These are described in detail in \textit{Objective Function and Gradient Routines} and \textit{Nonlinear Least Squares}. Here we remark that because gradient information is not available for scaling purposes, it can be useful to ensure that the problem is reasonably well scaled. A simple way to do so is to rescale the decision variables \( x \) so that their typical values are expected to lie within the unit hypercube \([0, 1]^n\).
Convergence Notes

Because the gradient function is not provided to POUNDerS, the norm of the gradient of the objective function is not available. Therefore, for convergence criteria, this norm is approximated by the norm of the model gradient and used only when the model gradient is deemed to be a reasonable approximation of the gradient of the objective. In practice, the typical grounds for termination for expensive derivative-free problems is the maximum number of function evaluations allowed.

Bound-constrained Regularized Gauss-Newton

The Gauss-Newton method is used to iteratively solve nonlinear least squares problem with the iterations

\[ x_{k+1} = x_k - \alpha_k (J_k^T J_k)^{-1} J_k^T r(x_k) \]

where \( r(x) \) is the least-squares residual vector, \( J_k = \partial r(x_k)/\partial x \) is the Jacobian of the residual, and \( \alpha_k \) is the step length parameter. In other words, the Gauss-Newton method approximates the Hessian of the objective as \( H_k \approx (J_k^T J_k) \) and the gradient of the objective as \( g_k \approx -J_k r(x_k) \). The least-squares Jacobian, \( J \), should be provided to Tao using TaoSetJacobianResidual() routine.

The BRGN implementation adds a regularization term \( \beta(x) \) such that

\[ \min_{x} \frac{1}{2} ||R(x)||^2 + \lambda \beta(x), \]

where \( \lambda \) is the scalar weight of the regularizer. BRGN provides two default implementations for \( \beta(x) \):

- **L2-norm** - \( \beta(x) = \frac{1}{2}||x||^2 \)
- **L2-norm Proximal Point** - \( \beta(x) = \frac{1}{2}||x_k - x_{k-1}||^2 \)
- **L1-norm with Dictionary** - \( \beta(x) = ||Dx||_1 \approx \sum_i \sqrt{y_i^2 + \epsilon^2} - \epsilon \) where \( y = Dx \) and \( \epsilon \) is the smooth approximation parameter.

The regularizer weight can be controlled with either TaoBRGNSetRegularizerWeight() or -tao_brgn_regularizer_weight command line option, while the smooth approximation parameter can be set with either TaoBRGNSetL1SmoothEpsilon() or -tao_brgn_l1_smooth_epsilon. For the L1-norm term, the user can supply a dictionary matrix with TaoBRGNSetDictionaryMatrix(). If no dictionary is provided, the dictionary is assumed to be an identity matrix and the regularizer reduces to a sparse solution term.

The regularization selection can be made using the command line option -tao_brgn_regularization_type <l2pure, l2prox, l1dict, user> where the user option allows the user to define a custom \( C^2 \)-continuous regularization term. This custom term can be defined by using the interface functions:

- TaoBRGNSetRegularizerObjectiveAndGradientRoutine() - Provide user-call back for evaluating the function value and gradient evaluation for the regularization term.
- TaoBRGNSetRegularizerHessianRoutine() - Provide user call-back for evaluating the Hessian of the regularization term.
Complementarity

Mixed complementarity problems, or box-constrained variational inequalities, are related to nonlinear systems of equations. They are defined by a continuously differentiable function, \( F : \mathbb{R}^n \to \mathbb{R}^n \), and bounds, \( \ell \in \{ \mathbb{R} \cup \{-\infty\}\}^n \) and \( u \in \{ \mathbb{R} \cup \{\infty\}\}^n \), on the variables such that \( \ell \leq u \). Given this information, \( x^* \in [\ell, u] \) is a solution to \( MCP(F, \ell, u) \) if for each \( i \in \{1, \ldots, n\} \) we have at least one of the following:

\[
F_i(x^*) \geq 0 \quad \text{if} \quad x^*_i = \ell_i
\]
\[
F_i(x^*) = 0 \quad \text{if} \quad \ell_i < x^*_i < u_i
\]
\[
F_i(x^*) \leq 0 \quad \text{if} \quad x^*_i = u_i.
\]

Note that when \( \ell = \{-\infty\}^n \) and \( u = \{\infty\}^n \), we have a nonlinear system of equations, and \( \ell = \{0\}^n \) and \( u = \{\infty\}^n \) correspond to the nonlinear complementarity problem [Cot64].

Simple complementarity conditions arise from the first-order optimality conditions from optimization [Kar39] [KT51]. In the simple bound-constrained optimization case, these conditions correspond to \( MCP(\nabla f, \ell, u) \), where \( f : \mathbb{R}^n \to \mathbb{R} \) is the objective function. In a one-dimensional setting these conditions are intuitive. If the solution is at the lower bound, then the function must be increasing and \( \nabla f \geq 0 \). If the solution is at the upper bound, then the function must be decreasing and \( \nabla f \leq 0 \). If the solution is strictly between the bounds, we must be at a stationary point and \( \nabla f = 0 \). Other complementarity problems arise in economics and engineering [FP97], game theory [Nas50], and finance [HP98].

Evaluation routines for \( F \) and its Jacobian must be supplied prior to solving the application. The bounds, \([\ell, u]\), on the variables must also be provided. If no starting point is supplied, a default starting point of all zeros is used.

Semismooth Methods

TAO has two implementations of semismooth algorithms [MFF+01] [DeLucaFK96] [FFK97] for solving mixed complementarity problems. Both are based on a reformulation of the mixed complementarity problem as a nonsmooth system of equations using the Fischer-Burmeister function [Fis92]. A nonsmooth Newton method is applied to the reformulated system to calculate a solution. The theoretical properties of such methods are detailed in the aforementioned references.

The Fischer-Burmeister function, \( \phi : \mathbb{R}^2 \to \mathbb{R} \), is defined as

\[
\phi(a, b) := \sqrt{a^2 + b^2} - a - b.
\]

This function has the following key property,

\[
\phi(a, b) = 0 \iff a \geq 0, \quad b \geq 0, \quad ab = 0,
\]

used when reformulating the mixed complementarity problem as the system of equations \( \Phi(x) = 0 \), where \( \Phi : \mathbb{R}^n \to \mathbb{R}^n \). The reformulation is defined componentwise as

\[
\Phi_i(x) := \begin{cases}
\phi(x_i - l_i, F_i(x)) & \text{if } -\infty < l_i < u_i = \infty, \\
-\phi(u_i - x_i, -F_i(x)) & \text{if } -\infty = l_i < u_i < \infty, \\
\phi(x_i - l_i, \phi(u_i - x_i, -F_i(x))) & \text{if } -\infty < l_i < u_i < \infty, \\
-F_i(x) & \text{if } -\infty = l_i < u_i = \infty, \\
l_i - x_i & \text{if } -\infty < l_i < u_i < \infty.
\end{cases}
\]

We note that \( \Phi \) is not differentiable everywhere but satisfies a semismoothness property [Mif77] [Qi93] [QS93]. Furthermore, the natural merit function, \( \Psi(x) := \frac{1}{2} \|\Phi(x)\|^2 \), is continuously differentiable.

The two semismooth TAO solvers both solve the system \( \Phi(x) = 0 \) by applying a nonsmooth Newton method with a line search. We calculate a direction, \( d^k \), by solving the system \( H^k d^k = -\Phi(x^k) \), where \( H^k \) is an
element of the \( B \)-subdifferential [QS93] of \( \Phi \) at \( x^k \). If the direction calculated does not satisfy a suitable descent condition, then we use the negative gradient of the merit function, \(-\nabla \Psi(x^k)\), as the search direction. A standard Armijo search [Arm66] is used to find the new iteration. Nonmonotone searches [GLL86] are also available by setting appropriate runtime options. See Line Searches for further details.

The first semismooth algorithm available in TAO is not guaranteed to remain feasible with respect to the bounds, \([\ell, u]\), and is termed an infeasible semismooth method. This method can be specified by using the `tao_ssils` solver. In this case, the descent test used is that

\[
\nabla \Psi(x^k)^T d^k \leq -\delta \|d^k\|^\rho.
\]

Both \( \delta > 0 \) and \( \rho > 2 \) can be modified by using the runtime options `-tao_ssils_delta <delta>` and `-tao_ssils_rho <rho>`, respectively. By default, \( \delta = 10^{-10} \) and \( \rho = 2.1 \).

An alternative is to remain feasible with respect to the bounds by using a projected Armijo line search. This method can be specified by using the `tao_ssfls` solver. The descent test used is the same as above where the direction in this case corresponds to the first part of the piecewise linear arc searched by the projected line search. Both \( \delta > 0 \) and \( \rho > 2 \) can be modified by using the runtime options `-tao_ssfls_delta <delta>` and `-tao_ssfls_rho <rho>` respectively. By default, \( \delta = 10^{-10} \) and \( \rho = 2.1 \).

The recommended algorithm is the infeasible semismooth method, `tao_ssils`, because of its strong global and local convergence properties. However, if it is known that \( F \) is not defined outside of the box, \([\ell, u]\), perhaps because of the presence of log functions, the feasibility-enforcing version of the algorithm, `tao_ssfls`, is a reasonable alternative.

### Quadratic Solvers

Quadratic solvers solve optimization problems of the form

\[
\begin{align*}
\min_x & \quad \frac{1}{2}x^T Q x + c^T x \\
\text{subject to} & \quad l \geq x \geq u
\end{align*}
\]

where the gradient and the Hessian of the objective are both constant.

### Gradient Projection Conjugate Gradient Method

The GPCG [MoreT91] algorithm is much like the TRON algorithm, discussed in Section Trust-Region Newton Method, except that it assumes that the objective function is quadratic and convex. Therefore, it evaluates the function, gradient, and Hessian only once. Since the objective function is quadratic, the algorithm does not use a trust region. All the options that apply to TRON except for trust-region options also apply to GPCG. It can be set by using the TAO solver `tao_gpcg` or via the option flag `-tao_type gpcg`.

### Interior-Point Newton’s Method

The BQPIP algorithm is an interior-point method for bound constrained quadratic optimization. It can be set by using the TAO solver of `tao_bqpip` or via the option flag `-tao_type bqpip`. Since it assumes the objective function is quadratic, it evaluates the function, gradient, and Hessian only once. This method also requires the solution of systems of linear equations, whose solver can be accessed and modified with the command `TaoGetKSP()`.
Constrained Solvers

Constrained solvers solve optimization problems of the form

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{subject to} & \quad g(x) = c \\
& \quad l \leq x \leq u
\end{align*}
\]

where \(g_i(x) = c_i\) for \(i = 1, \ldots, n\) are equality constraints.

Where applicable, TAO algorithms may convert inequality constraints to equality constraints via slack variables or penalty functions.

Augmented Lagrangian Multiplier Method

The TAOALMM method solves problems of the form

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{subject to} & \quad g(x) = 0 \\
& \quad h(x) \geq 0 \\
& \quad l \leq x \leq u
\end{align*}
\]

where \(g(x)\) are equality constraints, \(h(x)\) are inequality constraints and \(l\) and \(u\) are lower and upper bounds on the optimization variables, respectively.

TAOALMM converts the above general constrained problem into a sequence of bound constrained problems at each outer iteration \(k = 1, 2, \ldots\)

\[
\begin{align*}
\min_x & \quad L(x, \lambda_k) \\
\text{subject to} & \quad l \leq x \leq u
\end{align*}
\]

where \(L(x, \lambda_k)\) is the augmented Lagrangian merit function and \(\lambda_k\) is the Lagrange multiplier estimates at outer iteration \(k\).

TAOALMM offers two versions of the augmented Lagrangian formulation: the canonical Hestenes-Powell augmented Lagrangian \([\text{Hes69}]\ [\text{Pow69}]\) with inequality constrained converted to equality constraints via slack variables, and the slack-less Powell-Hestenes-Rockafellar formulation \([\text{Roc74}]\) that utilizes a pointwise \(\max()\) on the inequality constraints. For most applications, the canonical Hestenes-Powell formulation is likely to perform better. However, the PHR formulation may be desirable for problems featuring very large numbers of inequality constraints as it avoids inflating the dimension of the subproblem with slack variables.

The inner subproblem is solved using a nested bound-constrained first-order TAO solver. By default, TAOALM uses a quasi-Newton-Krylov trust-region method (TAOBQNKTR). Other first-order methods such as TAOBNCG and TAOBQNLS are also appropriate, but a trust-region globalization is strongly recommended for most applications.

Alternating Direction Method of Multipliers

The ADMM is an algorithm that is intended to blend the decomposability of dual ascent with the superior convergence properties of the method of multipliers. \([\text{BPC+11}]\) The algorithm solves problems in the form

\[
\begin{align*}
\min_x & \quad f(x) + g(z) \\
\text{subject to} & \quad Ax + Bz = c
\end{align*}
\]
where \( x \in \mathbb{R}^n, z \in \mathbb{R}^m, A \in \mathbb{R}^{p \times n}, B \in \mathbb{R}^{p \times m}, \) and \( c \in \mathbb{R}^p. \) Essentially, ADMM is a wrapper over two TAO solver, one for \( f(x), \) and one for \( g(z). \) With method of multipliers, one can form the augmented Lagrangian

\[
L_\rho(x, z, y) = f(x) + g(z) + y^T(Ax + Bz - c) + (\rho/2)\|Ax + Bz - c\|_2^2
\]

Then, ADMM consists of the iterations

\[
x^{k+1} := \text{argmin} L_\rho(x, z^k, y^k)
\]

\[
z^{k+1} := \text{argmin} L_\rho(x^{k+1}, z, y^k)
\]

\[
y^{k+1} := y^k + \rho(Ax^{k+1} + Bz^{k+1} - c)
\]

In certain formulation of ADMM, solution of \( z^{k+1} \) may have closed-form solution. Currently ADMM provides one default implementation for \( z^{k+1}, \) which is soft-threshold. It can be used with either `TaoADMMSetRegularizerType_ADMM()` or `-tao_admm_regularizer_type <regularizer_soft_thresh>`. User can also pass spectral penalty value, \( \rho, \) with either `TaoADMMSetSpectralPenalty()` or `-tao_admm_spectral_penalty`. Currently, user can use

- `TaoADMMSetMisfitObjectiveAndGradientRoutine()`
- `TaoADMMSetRegularizerObjectiveAndGradientRoutine()`
- `TaoADMMSetMisfitHessianRoutine()`
- `TaoADMMSetRegularizerHessianRoutine()`

Any other combination of routines is currently not supported. Hessian matrices can either be constant or non-constant, of which fact can be set via `TaoADMMSetMisfitHessianChangeStatus()`, and `TaoADMMSetRegularizerHessianChangeStatus()`. Also, it may appear in certain cases where augmented Lagrangian’s Hessian may become nearly singular depending on the \( \rho, \) which may change in the case of `-tao_admm_dual_update <update_adaptive>, <update_adaptive_relaxed>`. This issue can be prevented by `TaoADMMSetMinimumSpectralPenalty()`.

### General Nonlinear Programming problems

General nonlinear programming problems are of the form

\[
\min_x \quad f(x)
\]

subject to

\[
g(x) = 0
\]

\[
h(x) \geq 0
\]

\[
x^- \leq x \leq x^+
\]

(2.9)

Here, \( f(x) \) is the nonlinear objective function, \( g(x), h(x) \) are the equality and inequality constraints, and \( x^- \) and \( x^+ \) are the lower and upper bounds on decision variables \( x. \)

### Interior-point Newton’s method

The PDIPM (`-tao_type pdipm`) algorithm implements a primal-dual interior point method for solving general nonlinear programming problems. Using slack variables \( z \) and a log-barrier term, (2.9) is transformed to

\[
\min f(x) - \mu \sum_{i=1}^{nci} \ln z_i
\]

s.t.

\[
ce(x) = 0
\]

\[
ci(x) - z = 0
\]

(2.10)
Here, \( ce(x) \) is set of equality constraints that include \( g(x) \) and fixed decision variables, i.e., \( x^- = x = x^+ \). Similarly, \( ci(x) \) are inequality constraints including \( h(x) \) and lower/upper/box-constraints on \( x \). \( \mu \) is a parameter that is driven to zero as the optimization progresses.

The Lagrangian for (2.10) is

\[
L_\mu(x, \lambda_{ce}, \lambda_{ci}, z) = f(x) + \lambda_{ce}^T ce(x) - \lambda_{ci}^T ci(x) - z - \mu \sum_{i=1}^{nci} \ln z_i
\]

(2.11)

where, \( \lambda_{ce} \) and \( \lambda_{ci} \) are the lagrangian multipliers for the equality and inequality constraints, respectively.

The first order KKT conditions for optimality are as follows

\[
\nabla L_\mu(x, \lambda_{ce}, \lambda_{ci}, z) = \begin{bmatrix} \nabla f(x) + \nabla ce(x)^T \lambda_{ce} - \nabla ci(x)^T \lambda_{ci} \\ ce(x) \\ ci(x) - z \\ Z\lambda_{ci} - \mu e \end{bmatrix} = 0
\]

(2.12)

(2.12) is solved iteratively using Newton’s method using PETSC’s SNES object. After each Newton iteration, a line-search is performed to update \( x \) and enforce \( z, \lambda_{ci} \geq 0 \). The barrier parameter \( \mu \) is also updated after each Newton iteration. The Newton update is obtained by solving the second-order KKT system \( H\delta = -\nabla L_\mu \). Here, \( H \) is the Hessian matrix of the KKT system. For interior-point methods such as PDIPM, the Hessian matrix tends to be ill-conditioned, thus necessitating the use of a direct solver. We recommend using LU preconditioner `-pc_type lu` and using direct linear solver packages such `SuperLU_DIST` or `MUMPS`.

### 2.8.9 Advanced Options

This section discusses options and routines that apply to most TAO solvers and problem classes. In particular, we focus on linear solvers, convergence tests, and line searches.

#### Linear Solvers

One of the most computationally intensive phases of many optimization algorithms involves the solution of linear systems of equations. The performance of the linear solver may be critical to an efficient computation of the solution. Since linear equation solvers often have a wide variety of options associated with them, TAO allows the user to access the linear solver with the

```c
TaoGetKSP(Tao, KSP *
```

command. With access to the KSP object, users can customize it for their application to achieve improved performance. Additional details on the KSP options in PETSc can be found in the `PETSc/TAO Users Manual`.

#### Monitors

By default the TAO solvers run silently without displaying information about the iterations. The user can initiate monitoring with the command

```c
TaoSetMonitor(Tao,
```

```c
PetscErrorCode (*mon)(Tao, void*),
```

```c
void *
```

command.
The routine `mon` indicates a user-defined monitoring routine, and `void*` denotes an optional user-defined context for private data for the monitor routine.

The routine set by `TaoSetMonitor()` is called once during each iteration of the optimization solver. Hence, the user can employ this routine for any application-specific computations that should be done after the solution update.

**Convergence Tests**

Convergence of a solver can be defined in many ways. The methods TAO uses by default are mentioned in *Convergence*. These methods include absolute and relative convergence tolerances as well as a maximum number of iterations of function evaluations. If these choices are not sufficient, the user can specify a customized test.

Users can set their own customized convergence tests of the form

```
PetscErrorCode conv(Tao, void*);
```

The second argument is a pointer to a structure defined by the user. Within this routine, the solver can be queried for the solution vector, gradient vector, or other statistic at the current iteration through routines such as `TaoGetSolutionStatus()` and `TaoGetTolerances()`.

To use this convergence test within a TAO solver, one uses the command

```
TaoSetConvergenceTest(Tao, PetscErrorCode (*conv)(Tao, void*), void*);
```

The second argument of this command is the convergence routine, and the final argument of the convergence test routine denotes an optional user-defined context for private data. The convergence routine receives the TAO solver and this private data structure. The termination flag can be set by using the routine

```
TaoSetConvergedReason(Tao, TaoConvergedReason);
```

**Line Searches**

By using the command line option `-tao_ls_type`. Available line searches include Moré-Thuente [MoréT92], Armijo, gpcg, and unit.

The line search routines involve several parameters, which are set to defaults that are reasonable for many applications. The user can override the defaults by using the following options

- `-tao_ls_maxfev <max>`
- `-tao_ls_stepmin <min>`
- `-tao_ls_stepmax <max>`
- `-tao_ls_ftol <ftol>`
- `-tao_ls_gtol <gtol>`
- `-tao_ls_rtol <rtol>`

One should run a TAO program with the option `-help` for details. Users may write their own customized line search codes by modeling them after one of the defaults provided.
Recycling History

Some TAO algorithms can re-use information accumulated in the previous TaoSolve() call to hot-start the new solution. This can be enabled using the `-tao_recycle_history` flag, or in code via the `TaoSetRecycleHistory()` interface.

For the nonlinear conjugate gradient solver (TAOBNCG), this option re-uses the latest search direction from the previous TaoSolve() call to compute the initial search direction of a new TaoSolve(). By default, the feature is disabled and the algorithm sets the initial direction as the negative gradient.

For the quasi-Newton family of methods (TAOBQNLS, TAOBQNKL, TAOBQNKR, TAOBQNKL), this option re-uses the accumulated quasi-Newton Hessian approximation from the previous TaoSolve() call. By default, the feature is disabled and the algorithm will reset the quasi-Newton approximation to the identity matrix at the beginning of every new TaoSolve().

The option flag has no effect on other TAO solvers.

2.8.10 Adding a Solver

One of the strengths of both TAO and PETSc is the ability to allow users to extend the built-in solvers with new user-defined algorithms. It is certainly possible to develop new optimization algorithms outside of TAO framework, but using TAO to implement a solver has many advantages,

1. TAO includes other optimization solvers with an identical interface, so application problems may conveniently switch solvers to compare their effectiveness.

2. TAO provides support for function evaluations and derivative information. It allows for the direct evaluation of this information by the application developer, contains limited support for finite difference approximations, and allows the use of matrix-free methods. The solvers can obtain this function and derivative information through a simple interface while the details of its computation are handled within the toolkit.

3. TAO provides line searches, convergence tests, monitoring routines, and other tools that are helpful in an optimization algorithm. The availability of these tools means that the developers of the optimization solver do not have to write these utilities.

4. PETSc offers vectors, matrices, index sets, and linear solvers that can be used by the solver. These objects are standard mathematical constructions that have many different implementations. The objects may be distributed over multiple processors, restricted to a single processor, have a dense representation, use a sparse data structure, or vary in many other ways. TAO solvers do not need to know how these objects are represented or how the operations defined on them have been implemented. Instead, the solvers apply these operations through an abstract interface that leaves the details to PETSc and external libraries. This abstraction allows solvers to work seamlessly with a variety of data structures while allowing application developers to select data structures tailored for their purposes.

5. PETSc provides the user a convenient method for setting options at runtime, performance profiling, and debugging.
Header File

TAO solver implementation files must include the TAO implementation file \texttt{taoimpl.h}:

\begin{verbatim}
#include "petsc/private/taoimpl.h"
\end{verbatim}

This file contains data elements that are generally kept hidden from application programmers, but may be necessary for solver implementations to access.

TAO Interface with Solvers

TAO solvers must be written in C or C++ and include several routines with a particular calling sequence. Two of these routines are mandatory: one that initializes the TAO structure with the appropriate information and one that applies the algorithm to a problem instance. Additional routines may be written to set options within the solver, view the solver, setup appropriate data structures, and destroy these data structures. In order to implement the conjugate gradient algorithm, for example, the following structure is useful.

\begin{verbatim}
typedef struct{
    PetscReal beta;
    PetscReal eta;
    PetscInt ngradtseps;
    PetscInt nresetsteps;
    Vec X_old;
    Vec G_old;
} TAO_CG;
\end{verbatim}

This structure contains two parameters, two counters, and two work vectors. Vectors for the solution and gradient are not needed here because the TAO structure has pointers to them.

Solver Routine

All TAO solvers have a routine that accepts a TAO structure and computes a solution. TAO will call this routine when the application program uses the routine \texttt{TaoSolve()} and will pass to the solver information about the objective function and constraints, pointers to the variable vector and gradient vector, and support for line searches, linear solvers, and convergence monitoring. As an example, consider the following code that solves an unconstrained minimization problem using the conjugate gradient method.

\begin{verbatim}
PetscErrorCode TaoSolve_CG(Tao tao){
    TAO_CG *cg = (TAO_CG *) tao->data;
    Vec x = tao->solution;
    Vec g = tao->gradient;
    Vec s = tao->stepdirection;
    PetscInt iter=0;
    PetscReal gnormPrev, gdx, f, gnorm, steplength=0;
    TaoLineSearchConvergedReason lsflag=TAO_LINESEARCH_CONTINUE_ITERATING;
    TaoConvergedReason reason=TAO_CONTINUE_ITERATING;
    PetscErrorCode ierr;

    PetscFunctionBegin;
    ierr = TaoComputeObjectiveAndGradient(tao,x,&f,g); CHKERRQ(ierr);

    (continues on next page)
\end{verbatim}
The first line of this routine casts the second argument to a pointer to a TAO_CG data structure. This structure contains pointers to three vectors and a scalar that will be needed in the algorithm.

After declaring and initializing several variables, the solver lets TAO evaluate the function and gradient at the current point in the using the routine TaoComputeObjectiveAndGradient(). Other routines may be used to evaluate the Hessian matrix or evaluate constraints. TAO may obtain this information using direct evaluation or other means, but these details do not affect our implementation of the algorithm.

The norm of the gradient is a standard measure used by unconstrained minimization solvers to define convergence. This quantity is always nonnegative and equals zero at the solution. The solver will pass this quantity, the current function value, the current iteration number, and a measure of infeasibility to TAO with the routine

```c
PetscErrorCode TaoMonitor(Tao tao, PetscInt iter, PetscReal f, 
PetscReal res, PetscReal cnorm, PetscReal steplength, 
TaoConvergedReason *reason);
```

Most optimization algorithms are iterative, and solvers should include this command somewhere in each iteration. This routine records this information, and applies any monitoring routines and convergence tests set by default or the user. In this routine, the second argument is the current iteration number, and the
third argument is the current function value. The fourth argument is a nonnegative error measure associated with the distance between the current solution and the optimal solution. Examples of this measure are the norm of the gradient or the square root of a duality gap. The fifth argument is a nonnegative error that usually represents a measure of the infeasibility such as the norm of the constraints or violation of bounds. This number should be zero for unconstrained solvers. The sixth argument is a nonnegative steplength, or the multiple of the step direction added to the previous iterate. The results of the convergence test are returned in the last argument. If the termination reason is TAO_CONTINUE_ITERATING, the algorithm should continue.

After this monitoring routine, the solver computes a step direction using the conjugate gradient algorithm and computations using Vec objects. These methods include adding vectors together and computing an inner product. A full list of these methods can be found in the manual pages.

Nonlinear conjugate gradient algorithms also require a line search. TAO provides several line searches and support for using them. The routine

```c
TaoLineSearchApply(TaoLineSearch ls, Vec x, PetscReal *f, Vec g, 
    TaoVec *s, PetscReal *steplength, 
    TaoLineSearchConvergedReason *lsflag)
```

passes the current solution, gradient, and objective value to the line search and returns a new solution, gradient, and objective value. More details on line searches can be found in Line Searches. The details of the line search applied are specified elsewhere, when the line search is created.

TAO also includes support for linear solvers using PETSc KSP objects. Although this algorithm does not require one, linear solvers are an important part of many algorithms. Details on the use of these solvers can be found in the PETSc users manual.

**Creation Routine**

The TAO solver is initialized for a particular algorithm in a separate routine. This routine sets default convergence tolerances, creates a line search or linear solver if needed, and creates structures needed by this solver. For example, the routine that creates the nonlinear conjugate gradient algorithm shown above can be implemented as follows.

```c
PETSC_EXTERN PetscErrorCode TaoCreate_CG(Tao tao) 
{
    TAO_CG *cg = (TAO_CG*)tao->data;
    const char *morethuente_type = TAOLINESEARCH_MT;
    PetscErrorCode ierr;

    PetscFunctionBegin;

    ierr = PetscNewLog(tao,&cg); CHKERRQ(ierr);
    tao->data = (void*)cg;
    cg->eta = 0.1;
    cg->delta_min = 1e-7;
    cg->delta_max = 100;
    cg->cg_type = CG_PolakRibierePlus;

    tao->max_it = 2000;
    tao->max_funcs = 4000;
    tao->ops->setup = TaoSetUp_CG;
    tao->ops->solve = TaoSolve_CG;
    tao->ops->view = TaoView_CG;

    (continues on next page)
```
This routine declares some variables and then allocates memory for the TAO_CG data structure. Notice that the Tao object now has a pointer to this data structure (tao->data) so it can be accessed by the other functions written for this solver implementation.

This routine also sets some default parameters particular to the conjugate gradient algorithm, sets default convergence tolerances, and creates a particular line search. These defaults could be specified in the routine that solves the problem, but specifying them here gives the user the opportunity to modify these parameters either by using direct calls setting parameters or by using options.

Finally, this solver passes to TAO the names of all the other routines used by the solver.

Note that the lines EXTERN_C_BEGIN and EXTERN_C_END surround this routine. These macros are required to preserve the name of this function without any name-mangling from the C++ compiler (if used).

### Destroy Routine

Another routine needed by most solvers destroys the data structures created by earlier routines. For the nonlinear conjugate gradient method discussed earlier, the following routine destroys the two work vectors and the TAO_CG structure.

This routine is called from within the TaoDestroy() routine. Only algorithm-specific data objects are destroyed in this routine; any objects indexed by TAO (tao->linesearch, tao->ksp, tao->gradient, etc.) will be destroyed by TAO immediately after the algorithm-specific destroy routine completes.
**SetUp Routine**

If the SetUp routine has been set by the initialization routine, TAO will call it during the execution of `TaoSolve()`. While this routine is optional, it is often provided to allocate the gradient vector, work vectors, and other data structures required by the solver. It should have the following form.

```c
PetscErrorCode TaoSetUp_CG(Tao tao)
{
    PetscErrorCode ierr;
    TAO_CG *cg = (TAO_CG*)tao->data;
    PetscFunctionBegin;
    ierr = VecDuplicate(tao->solution,&tao->gradient); CHKERRQ(ierr);
    ierr = VecDuplicate(tao->solution,&tao->stepdirection); CHKERRQ(ierr);
    ierr = VecDuplicate(tao->solution,&cg->X_old); CHKERRQ(ierr);
    ierr = VecDuplicate(tao->solution,&cg->G_old); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}
```

**SetFromOptions Routine**

The SetFromOptions routine should be used to check for any algorithm-specific options set by the user and will be called when the application makes a call to `TaoSetFromOptions()`. It should have the following form.

```c
PetscErrorCode TaoSetFromOptions_CG(Tao tao, void *solver);
{
    PetscErrorCode ierr;
    TAO_CG *cg = (TAO_CG*)solver;
    PetscFunctionBegin;
    ierr = PetscOptionsReal("-tao_cg_eta","restart tolerance","",cg->eta,&cg->eta,0); CHKERRQ(ierr);
    ierr = PetscOptionsReal("-tao_cg_delta_min","minimum delta value","",cg->delta_min,&cg->delta_min,0); CHKERRQ(ierr);
    ierr = PetscOptionsReal("-tao_cg_delta_max","maximum delta value","",cg->delta_max,&cg->delta_max,0); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}
```

**View Routine**

The View routine should be used to output any algorithm-specific information or statistics at the end of a solve. This routine will be called when the application makes a call to `TaoView()` or when the command line option `-tao_view` is used. It should have the following form.

```c
PetscErrorCode TaoView_CG(Tao tao, PetscViewer viewer)
{
    TAO_CG *cg = (TAO_CG*)tao->data;
    PetscErrorCode ierr;
    PetscFunctionBegin;
    ierr = PetscViewerASCIIPushTab(viewer);
    PetscFunctionReturn(ierr);
}
```
Registering the Solver

Once a new solver is implemented, TAO needs to know the name of the solver and what function to use to create the solver. To this end, one can use the routine

```c
TaoRegister(const char *name,
            const char *path,
            const char *cname,
            PetscErrorCode (*create)(Tao));
```

where `name` is the name of the solver (i.e., `tao_blmvm`), `path` is the path to the library containing the solver, `cname` is the name of the routine that creates the solver (in our case, `TaoCreate_CG`), and `create` is a pointer to that creation routine. If one is using dynamic loading, then the fourth argument will be ignored.

Once the solver has been registered, the new solver can be selected either by using the `TaoSetType()` function or by using the `-tao_type` command line option.

2.9 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

\[ Ax = \lambda M x \]

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.

### 2.9.1 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

$$\phi_i^F f_i^F - \phi_i^F Pf^C = f_i^F - \sum_j P_{ij} f_j^C$$

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,

\[
\min_{P_{ij}} \sum_k w_k \| f_i^{F,k} - \sum_j P_{ij} f_j^{C,k} \|_2
\]

\[
= \min_{P_{ij}} \sum_k \| \sqrt{w_k} f_i^{F,k} - \sqrt{w_k} \sum_j P_{ij} f_j^{C,k} \|_2
\]

\[
= \min_{P_{ij}} \| W^{1/2} f_i - W^{1/2} f C p_i \|_2
\]

where

\[
W = \begin{pmatrix}
w_0 \\
\vdots \\
w_K 
\end{pmatrix}
\]

\[
f_i^F = \begin{pmatrix}
f_i^{F,0} \\
\vdots \\
f_i^{F,K} 
\end{pmatrix}
\]

\[
f^C = \begin{pmatrix}
f_0^{C,0} & \cdots & f_n^{C,0} \\
\vdots & \ddots & \vdots \\
f_0^{C,K} & \cdots & f_n^{C,K} 
\end{pmatrix}
\]

\[
p_i = \begin{pmatrix}
P_{i0} \\
\vdots \\
P_{in} 
\end{pmatrix}
\]

or alternatively

\[
k_k = w_k
\]

\[
[f_i^F]_k = f_i^{F,k}
\]

\[
[f_i^C]_j = f_j^{C,k}
\]

\[
[p_i]_j = P_{ij}
\]

We thus have a standard least-squares problem

\[
\min_{P_{ij}} \| b - Ax \|_2
\]

where

\[
A = W^{1/2} f^C
\]

\[
b = W^{1/2} f_i^F
\]

\[
x = p_i
\]

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 $P_{l+1}$, and the interpolator should be recomputed.

### 2.10 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.

#### 2.10.1 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 PetscInts. 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, 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. 2.6,

![Fig. 2.6: A 2D doublet mesh, two triangles sharing an edge.](image)

which can also be represented as the DAG in Fig. 2.7.

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. 2.6, these numberings are

$f_0 \mapsto 0, f_1 \mapsto 1, \quad v_0 \mapsto 2, v_1 \mapsto 3, v_2 \mapsto 4, v_3 \mapsto 5, \quad e_0 \mapsto 6, e_1 \mapsto 7, e_2 \mapsto 8, e_3 \mapsto 9, e_4 \mapsto 10$
First, we declare the set of points present in a mesh,

```c
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,

```c
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,

```c
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

```c
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

```c
DMPlexStratify(dm);
```
2.10.2 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 \([pStart, pEnd)\).

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. 2.6, we can lay out data for a continuous Galerkin \( P_3 \) finite element method,

```c
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(dm, &depth)` routines the depth of the DAG, hence `DMPlexGetHeightStratum(dm, depth-1-h,)` returns the same values as `DMPlexGetHeightStratum(dm, h,)`.

For \( P_3 \) elements there is one degree of freedom at each vertex, 2 along each edge (resulting in a total of 4 degrees of freedom along 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,
though it is usually easier to use the DM directly, which also provides global vectors,

```c
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 contrast, 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`.

### 2.10.3 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`. 

---

**2.10. DMPlex: Unstructured Grids in PETSc**
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.

```c
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:

```c
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:

```c
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.

## 2.10.4 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.

   ```c
   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, or generator/transmission line data for power grids. Components are registered by calling

   ```c
   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 36 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 shared vertices of the physical subnetworks. The topological sizes of the network are set by calling

   ```c
   DMNetworkSetNumSubNetworks(DM dm, PetscInt nsubnet, PetscInt Nsubnet);
   ```

   Here, `nsubnet` and `Nsubnet` are the local and global number of subnetworks.

4. A subnetwork is added to the network by calling

   ```c
   DMNetworkAddSubnetwork(DM dm, const char *name, PetscInt nv, PetscInt ne, PetscInt *edgelist, PetscInt *netnum);
   ```

   Here `name` is the subnetwork name, `nv` and `ne` are the numbers of local vertices and local edges on the subnetwork, and `edgelist` is the connectivity for the subnetwork. The output `netnum` is the global numbering of the subnetwork in the network. Each element of `edgelist` is an integer array of size `2*ne` containing the edge connectivity for the subnetwork.

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

The two subnetworks are coupled by merging vertex 0 from subnetwork 0 with vertex 2 from subnetwork 1.

The `edgelist` of this network is
edgelist[0] = {0,1,2,2,3}
edgelist[1] = {1,2,0}

The coupling is done by calling

```c
DMNetworkAddSharedVertices(DM dm, PetscInt anet, PetscInt bnet, PetscInt nsv, __PetscInt asv[], PetscInt bsv[]);
```

Here `anet` and `bnet` are the first and second subnetwork global numberings returned by `DMNetworkAddSubnetwork()`, `nsv` is the number of vertices shared by the two subnetworks, `asv` and `bsv` are the vertex indices in the subnetwork `anet` and `bnet`.

5. The next step is to have DMNetwork create a bare layout (graph) of the network by calling

```c
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 to the vertices and edges.

A component and number of variables are added to a vertex/edge by calling

```c
DMNetworkAddComponent(DM dm, PetscInt p, PetscInt compkey, void* compdata, __PetscInt nvar)
```

where `p` is the network vertex/edge point in the range obtained by either `DMNetworkGetVertexRange()`, `DMNetworkGetEdgeRange()`, `DMNetworkGetSubnetwork()`, or `DMNetworkGetSharedVertices()`; `compkey` is the component key returned when registering the component (`DMNetworkRegisterComponent()`); `compdata` holds the data for the component; and `nvar` is the number of variables associated to the added component at this network point. DMNetwork supports setting multiple components (max. 36) at a vertex/edge. At a shared vertex, DMNetwork currently requires the owner process of the vertex adds all the components and number of variables.

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.

7. Set up network internal data structures.

```c
DMSetUp(DM dm);
```

8. Distribute the network (also moves components attached with vertices/edges) to multiple processors.

```c
DMNetworkDistribute(DM dm, const char partitioner[], PetscInt overlap, DM __distDM);
```

9. Associate the `DM` with a PETSc solver:

```c
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,

```c
DMNetworkGetEdgeRange(DM dm, PetscInt *eStart, PetscInt *eEnd);
```

```c
DMNetworkGetVertexRange(DM dm, PetscInt *vStart, PetscInt *vEnd);
```

```c
DMNetworkGetSubnetwork(DM dm, PetscInt netnum, PetscInt *nv, PetscInt *ne, const PetscInt **vtx, const PetscInt **edge);
```

checking the status of a vertex,

```c
DMNetworkIsGhostVertex(DM dm, PetscInt p, PetscBool *isghost);
```

```c
DMNetworkIsSharedVertex(DM dm, PetscInt p, PetscBool *isshared);
```

and retrieving local/global indices of vertex/edge component variables for inserting elements in vectors/matrices,

```c
DMNetworkGetLocalVecOffset(DM dm, PetscInt p, PetscInt compnum, PetscInt *offset);
```

```c
DMNetworkGetGlobalVecOffset(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.

```c
DMNetworkGetConnectedVertices(DM dm, PetscInt edge, const PetscInt *vertices[]);
```

```c
DMNetworkGetSupportingEdges(DM dm, PetscInt vertex, PetscInt *nedges, const PetscInt *edges[]).
```

Retrieving components and number of variables

The components and the corresponding number of variables set at a vertex/edge can be accessed by

```c
DMNetworkGetComponent(DM dm, PetscInt p, PetscInt compnum, PetscInt *compkey, void *component, PetscInt *nvar)
```

input **compnum** is the component number, output **compkey** is the key set by **DMNetworkRegisterComponent**. An example of accessing and retrieving the components and number of variables at vertices is:

```c
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, &nvar);
        compdata = (UserCompDataType)(component);
    }
}
```

(continues on next page)
The above example does not explicitly use the component key. It is used when different component types are set at different vertices. In this case, `compkey` is used to differentiate the component type.
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 Synopsis

To use PETSc with Fortran you must use both PETSc include files and modules. At the beginning of every function and module definition you need something like

```plaintext
#include "petsc/finclude/petscXXX.h"
use petscXXX
```

You can declare PETSc object variables using either of the following:

```plaintext
XXX variablename
```

```plaintext
type(tXXX) variablename
```

For example,

```plaintext
#include "petsc/finclude/petscvec.h"
use petscvec

Vec b
```

```plaintext
Vype(tVec) x
```

PETSc types like PetscInt and PetscReal are simply aliases for basic Fortran types and cannot be written as type(tPetscInt)

3.1.2 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 `dabc_()`. 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:
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:

```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, \( v_{\text{new}} \) of size \( n \). Then, after VecDuplicateVecs() has been called, \( v_{\text{new}} \) 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 \( v_{\text{old}} \) to form two new vectors, \( v_{\text{new}(1)} \) and \( v_{\text{new}(2)} \).

```fortran
    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:

```fortran
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:

```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:

```fortran
VecGetArrayF90(), VecRestoreArrayF90()
VecGetArrayReadF90(), VecRestoreArrayReadF90()
VecDuplicateVecsF90(), VecDestroyVecsF90()
DMADVecGetArrayF90(), DMADVecGetArrayReadF90(), ISLocalToGlobalMappingGetIndicesF90()
MatDenseGetArrayF90(), MatDenseRestoreArrayF90()
ISGetIndicesF90(), ISRestoreIndicesF90()
```

See the manual pages for details and pointers to example programs.
3.1.3 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`

```fortran
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
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)
```

(continues on next page)
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

!/*TEST
|
|   test:
|!TEST*/

Listing: src/vec/vec/tutorials/ex4f.F

<p>| |
| |
| 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 |</p>
<table>
<thead>
<tr>
<th>T*/</th>
</tr>
</thead>
</table>

program main
#include <petsc/hinclude/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
        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)

(continues on next page)
! 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

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*/
This example demonstrates basic use of the Fortran interface for PetscDraw routines.

```fortran
implicit none
!
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
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, &
    '-n',n,flg,ierr)
call PetscDrawCreate(PETSC_COMM_WORLD,PETSC_NULL_CHARACTER, &
  & PETSC_NULL_CHARACTER,x,y,width,height,draw,ierr)
call PetscDrawLGCreate(draw,one,lg,ierr)
call PetscLGGetAxis(lg,axis,ierr)
call PetscDrawAxisSetColors(axis,PETSC_DRAW_BLACK,PETSC_DRAW_RED, &
  & PETSC_DRAW_BLUE,ierr)
call PetscDrawAxisSetLabels(axis,'toplabel','xlabel','ylabel', &
  & ierr)
! loop  
do 10, i=0,n-1
  xd = real(i) - 5.0
  yd = xd*xd
  call PetscDrawLGAddPoint(lg,xd,yd,ierr)
10  continue
```

(continues on next page)
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/exl_1.out
|!
!TEST*/

Listing: src/snes/tutorials/exlf.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

 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

(continues on next page)
Mat J
SNESLineSearch linesearch
PetscErrorCode ierr
PetscInt its,i2,i20
PetscMPIInt size,rank
PetscScalar pfive
PetscReal tol
PetscBool setls

#define lx_a(ib) lx_v(lx_i + (ib))
#define lf_a(ib) lf_v(lf_i + (ib))

! Note: Any user-defined Fortran routines (such as FormJacobian)
! MUST be declared as external.

external FormFunction, FormJacobian, MyLineSearch

#begin

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! 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
! FormFunction() and FormJacobian().
! - The element lx_a(ib) is element ib in the vector x
!
! dummy test of logging a reduction

! if defined(PETSC_USE_LOG)
! ierr = PetscAReduce()
! endif

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! 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

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_moniter -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 SNESolve(). 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 SNESolve(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
   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)
#endif

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:
! 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.

  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
!   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

(continues on next page)
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, 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
end

/*TEST
 |
 | test:
 |   args: -ksp_gmres_cgs_refinement_type refine_always -snes_monitor_short
 | requires: -single
 |
!TEST*/

Array Arguments

This material is no longer relevant 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:

PetcScalar xx_v(1), aa_v(1)
PetcErrorcode ierr
PetcInt ss_v(1), dd_v(1), nloc
PetcOffset ss_i, xx_i, aa_i, dd_i
Vec x
Mat A
IS s
DM d

(continues on next page)
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 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,

```c
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

```c
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

```c
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:

```matlab
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

```c
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

```c
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 **);
```

### 3.2. Using MATLAB with PETSc

211
There is a short-cut to starting the MATLAB engine with `PETSC_MATLAB.Engine (MPI_Comm)`. If you are running PETSc on a cluster (or machine) that does not have a license for MATLAB, you might able to run MATLAB on the head node of the cluster or some other machine accessible to the cluster using the `-matlab_engine_host hostname` option.

### 3.2.4 Licensing the MATLAB Compute Engine on a cluster

To activate MATLAB on head node which does not have access to the internet. (see also: [https://www.mathworks.com/matlabcentral/answers/259627-how-do-i-activate-matlab-or-other-mathworks-products-without-an-internet-connection](https://www.mathworks.com/matlabcentral/answers/259627-how-do-i-activate-matlab-or-other-mathworks-products-without-an-internet-connection))

First ssh into the head node using the command: `ssh node_name`


Login to your MathWorks Account from a computer which has internet access. You will see the available license that your account has. Select a license from the list.

Then, select Install and Activate option and select the Activate to Retrieve License File option.

Enter the information and click Continue.

An option to download the License file will appear. Download it and copy the license file to the cluster (your home directory). Now, launch MATLAB where you have sshed into your head node.

Select the Activate manually without the internet option and click Next >. Browse and locate the license file.

MATLAB is activated and ready to use.
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.
Manually Activate Software on a Computer

Select software release.

**Release**  
R2020b

Describe computer on which software will be activated.

**Operating System**  
Linux

**Host ID**

How do I find my computer's host id?

**Computer Login Name**

How do I find my Computer Login Name?

Provide a label so you can easily identify this computer later.

**Activation Label**
Activate MathWorks Software

Activation is a process that verifies licensed use of MathWorks products. This process validates the license and ensures that it is not used on more systems than allowed by the license option you have acquired.

- Activate automatically using the Internet (recommended)
- Activate manually without the Internet

Activate without an Internet connection

- Enter the full path to your license file, including the file name:
- I do not have a license file. Help me with the next steps.
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 \( y = ax + y \) 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
      - MatILUFactorNumeric - Numeric factorization phase
  - 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.
The output produced with -log_view is flat, meaning that the hierarchy of PETSc operations is not completely clear. 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. If this is problematic then it is also possible to examine the profiling information in a nested format. For more information see Profiling Nested Events.

**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.

```bash
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, 2018 13:36:54
Using Petsc Development GIT revision: v3.9-163-gbe3efd42 GIT Date: 2018-04-16 10:45:40

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>1.849e-01</td>
<td>1.00002</td>
<td>1.849e-01</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.060e+02</td>
<td>1.00000</td>
<td>1.060e+02</td>
<td></td>
</tr>
<tr>
<td>Flop:</td>
<td>2.361e+08</td>
<td>1.00684</td>
<td>2.353e+08</td>
<td></td>
</tr>
<tr>
<td>Flop/sec:</td>
<td>1.277e+09</td>
<td>1.00685</td>
<td>1.273e+09</td>
<td></td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>2.360e+02</td>
<td>1.34857</td>
<td>2.061e+02</td>
<td></td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>1.256e+07</td>
<td>2.24620</td>
<td>4.071e+04</td>
<td></td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>2.160e+02</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary of Stages: --- Time ------ --- Flop ------ --- Messages --- ---
--- Message Lengths --- --- Reductions --- ---
    | %Total | counts | %Total | Avg | %Total | counts | %Total | Avg |
0: Main Stage: | 5.9897e-04 | 0.3%  | 0.0000e+00 | 0.0% | 0.0000e+00 | 0.0% | 0.0000e+00 | 0.0% |
1: Load System: | 2.9113e-03 | 1.6%  | 0.0000e+00 | 0.0% | 3.550e+01 | 4.3% | 3.550e+01 | 4.3% |
984e+02 | 0.1% | 2.200e+01 | 10.2% |
```
(continues on next page)
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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.

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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.

\[
\text{mpiexec -n 4 ./ex10 -f0 medium -f1 arco6 -ksp_gmres_classical_gramschmidt -log_view -mat_type baij -matload_block_size 3 -pc_type bjacobi -options_left}
\]

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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.

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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.

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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.

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.

\[
\text{Total Mflop/sec} = 10^{-6} \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}
\]

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.
### 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
  
  ```c
  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**: \(10^6 \times \frac{\text{sum of flop over all processors}}{\text{max time over all processors}}\)

<table>
<thead>
<tr>
<th>Phase</th>
<th>Count</th>
<th>Time (sec)</th>
<th>Flop/sec</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Max</td>
<td>Ratio</td>
<td>Max</td>
<td>Ratio</td>
<td>Mess</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**--- Event Stage 5: KSPSetUp 1**

| MatFactorNum          | 1 1.0 3.6440e-03 1.1 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00 2 |
| MatFactorSym          | 1 1.0 1.7111e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 1 |
| MatGetRowIJ           | 1 1.0 1.1921e-06 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 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 |
| KSPSetUp              | 2 1.0 6.6495e-04 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 2 |
| PCSetUp               | 2 1.0 5.4271e-03 1.2 5.30e+06 1.0 0.00e+00 0.0e+00 0.0e+00 1 |
| PCSetUpOnBlocks       | 1 1.0 5.3999e-03 1.2 5.30e+06 1.0 0.00e+00 0.0e+00 0.0e+00 3 |

---

**--- 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 |
| MatSolve             | 61 1.0 1.9177e-02 1.0 5.99e+07 1.0 0.0e+00 0.0e+00 0.0e+00 10 |
| VecMDot              | 59 1.0 1.4741e-02 1.3 4.86e+07 1.0 0.0e+00 0.0e+00 5.9e+01 7 |
| VecNorm              | 61 1.0 3.0417e-03 1.4 3.29e+06 1.0 0.0e+00 0.0e+00 6.1e+01 1 |
| VecScale             | 61 1.0 9.9802e-04 1.0 1.65e+06 1.0 0.0e+00 0.0e+00 0.0e+00 1 |
| VecCopy              | 2 1.0 5.9128e-05 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 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 |

(continues on next page)
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`. 

... [Conclusion of overall summary, see part I] ...
Profiling Nested Events

It is possible to output the PETSc logging information in a nested format where the hierarchy of events is explicit. This output can be generated either as an XML file or as a text file in a format suitable for viewing as a flame graph.

One can generate the XML output by passing the option `-log_view:[logfile]:ascii_xml`. It can be generated by copying `${PETSC_DIR}/share/petsc/xml/performance_xml2html.xsl` into the current directory, then opening the logfile in your browser.

The flame graph output can be generated with the option `-log_view:[logfile]:ascii_flamegraph`. It can then be visualised with either FlameGraph or speedscope.

Legacy XML profiles may be converted into this format using the script `${PETSC_DIR}/lib/petsc/bin/xml2flamegraph.py`.

Note that user-defined stages (see Profiling Multiple Sections of Code) will be ignored when using this nested format.

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:

```c
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:

```c
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

```c
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.

3.3. Profiling 221
The user can log the number of floating-point operations for this segment of code by calling

```c
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

```c
PetscLogStagePush(PetscLogStage stage);
PetscLogStagePop();
```

where `stage` is an integer (0-9); see the manual pages for details. The command

```c
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.

```c
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*. 

222 Chapter 3. Additional Information
### 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

```c
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):

```c
PetscLogEventDeactivateClass(MAT_CLASSID);
PetscLogEventDeactivateClass(KSP_CLASSID); /* includes PC and KSP */
PetscLogEventDeactivateClass(VEC_CLASSID);
PetscLogEventDeactivateClass(SNES_CLASSID);
```

and

```c
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

```c
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

```c
PetscInfo(snes,"Cubically determined step, lambda %.5g\n",lambda);
```

One can selectively turn off informative messages about any of the basic PETSc objects (e.g., `Mat`, `SNES`) with the command

```c
PetscInfoDeactivateClass(int object_classid)
```

where `object_classid` is one of `MAT_CLASSID`, `SNES_CLASSID`, etc. Messages can be reactivated with the command

---

**3.3. Profiling** 223
3.3.6 Time

PETSc application programmers can access the wall clock time directly with the command

```c
PetscLogDouble time;
PetcTime(&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 page fault 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

```c
PetscPreLoadBegin(PetscBool flag,char* stagename)
PetcPreLoadStage(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 embarrassingly 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
```

(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.

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 (P#1 63GB)
```

(continues on next 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: 010000000000010000000000
process 2 binding: 001000000000001000000000
process 3 binding: 000100000000000100000000
process 4 binding: 000010000000000010000000
process 5 binding: 000001000000000001000000
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: 000000000001000000000000
process 2 binding: 001000000000000010000000
process 3 binding: 000100000000000001000000
process 4 binding: 000010000000000000100000
process 5 binding: 000001000000000000010000
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)
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 documentation 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 implementation 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 facilitate 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 `SNES-JacobianEval`. 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
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 \texttt{MatCreateSeqAIJ()} and \texttt{MatCreateAIJ()} for compressed, sparse row formats, instead of the generic \texttt{MatCreate()} routine. For problems with multiple degrees of freedom per node, the block, compressed, sparse row formats, created by \texttt{MatCreateSeqBAIJ()} and \texttt{MatCreateBAIJ()}, can significantly enhance performance. \textit{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 \texttt{MatILUInfo} structure when calling \texttt{MatILUFactorSymbolic()} or \texttt{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 \texttt{-info}. The symbolic factorization phase will then print information such as

\begin{verbatim}
Info:MatILUFactorSymbolic_AIJ:Realloc 12 Fill ratio:given 1 needed 2.16423
\end{verbatim}

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

\begin{verbatim}
-pc_ilu_fill 2.17
\end{verbatim}

will cause PETSc to preallocate the correct amount of space for incomplete (ILU) factorization. The corresponding option for direct (LU) factorization is \texttt{-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 \texttt{--with-debugging=1}). PETSc memory allocation can be activated for optimized-mode (configured with \texttt{--with-debugging=0}) using the option \texttt{-malloc}. The option \texttt{-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 \texttt{-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 \texttt{-malloc_dump} merely activates a call to \texttt{PetscMallocDump()} during \texttt{PetscFinalize()} the user can also call \texttt{PetscMallocDump()} elsewhere in a program.

- Another useful option for use with PETSc memory allocation routines is \texttt{-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 hard-wired 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_restart <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 `getusage` 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 The Use of BLAS and LAPACK in PETSc and external libraries

1. BLAS 1 operations - vector operations such as VecNorm(), VecAXPY(), and VecScale(). Depending on the simulation the size of the vectors may be from hundreds of entries to many millions.
2. BLAS 2 operations - dense matrix with vector operations, generally the dense matrices are very small.
3. Eigenvalue and SVD computations, generally for very small matrices
4. External packages such as MUMPS and SuperLU_DIST use BLAS 3 operations (and possibly BLAS 1 and 2). The dense matrices may be of modest size, going up to thousands of rows and columns.

For most PETSc simulations (that is not using certain external packages) using an optimized set of BLAS/LAPACK routines only provides a modest improvement in performance. For some external packages using optimized BLAS/LAPACK can make a dramatic improvement in performance.

3.5.1 32 or 64 bit BLAS/LAPACK integers

BLAS/LAPACK libraries may use 32 bit integers or 64 bit integers. PETSc configure and compile handles this automatically so long at the arguments to the BLAS/LAPACK routines are set to the type PetscBLASInt. The routine PetscBLASIntCast(PetscInt,PetscBLASInt*) casts a PetscInt to the BLAS/LAPACK size. If the BLAS/LAPACK size is not large enough it generates an error. For the vast majority of simulations even very large ones 64 bit BLAS/LAPACK integers are not needed, even if 64 bit PETSc integers are used. The configure option -with-64-bit-blas-indices attempts to locate and use a 64 bit integer BLAS/LAPACK library.

3.5.2 Shared memory BLAS/LAPACK parallelism

Some BLAS/LAPACK libraries make use of shared memory parallelism within the function calls, generally using OpenMP, or possibly PThreads. If this feature is on, it is in addition to the MPI based parallelism that PETSc is using. Thus it can result in over-subscription of hardware resources. For example, if a system has 16 cores and PETSc is run with an MPI size of 16 then each core is assigned an MPI process. But if the BLAS/LAPACK is running with OpenMP and 4 threads per process this results 64 threads competing to use 16 cores which generally will perform poorly.

If one elects to use both MPI parallelism and OpenMP BLAS/LAPACK parallelism one should insure they do not over subscribe the hardware resources. Since PETSc does not natively using OpenMP this means that phases of the computation that do not use BLAS/LAPACK will be under-subscribed, thus under-utilizing the system. For PETSc simulations which do not use external packages there is generally no benefit to using parallel BLAS/LAPACK. The environmental variable OMP_NUM_THREADS can be used to set the number of threads used by parallel BLAS/LAPACK. The additional environmental variables OMP_PROC_BIND and OMP_PLACES may also need to be set appropriate for the system to obtain good parallel performance with BLAS/LAPACK. The configure option -with-openmp will trigger PETSc to try to locate and use a parallel BLAS/LAPACK library.

Certain external packages such as MUMPS may benefit from using parallel BLAS/LAPACK operations. See the manual page MATSOLVERMUMPS for details on how one can restrict the number of MPI processes while running MUMPS to utilize parallel BLAS/LAPACK.

236 Chapter 3. Additional Information
3.5.3 Available BLAS/LAPACK libraries

Most systems (besides Microsoft Windows) come with pre-installed BLAS/LAPACK which are satisfactory for many PETSc simulations.

The freely available Intel MKL mathematics libraries provide BLAS/LAPACK that are generally better performing than the system provided libraries and are generally fine for most users.

For systems that do not provide BLAS/LAPACK, such as Microsoft Windows, PETSc provides the Fortran reference version `--download-fblaslapack` and a `f2c` generated C version `--download-f2cblaslapack` (which also supports 128 bit real number computations). These libraries are generally low performing but useful to get started with PETSc easily.

PETSc also provides access to OpenBLAS via the `--download-openblas` configure option. OpenBLAS uses some highly optimized operations but falls back on reference routines for many other operations. See the OpenBLAS manual for more information. The configure option `--download-openblas` provides a full BLAS/LAPACK implementation.

BLIS does not bundle LAPACK with it so PETSc’s configure attempts to locate a compatible system LAPACK library to use if `--download-blis` is selected. One can use `--download-f2cblaslapack --download-blis` to build netlib LAPACK with BLIS. This is recommended as a portable high-performance option.

3.6 Other PETSc Features

3.6.1 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 “manager” process to coordinate the work of “worker” 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()`.

3.6.2 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:

```c
PetscInitialize(int *argc, char ***args, const char *file, const char *help); /* C */
```

```fortran
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 `.petsrc` file in the format

```
alias -newname -oldname
```

For example,

```
alias -kspt -ksp_type
alias -sd -start_in_debugger
```

Comments can be placed in the `.petsrc` 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.

Adding options from a file

PETSc can load additional options from a file using `PetscOptionsInsertFile()`, which can also be used from the command line, e.g. `-options_file my_options.opts`.

One can also use YAML files this way (relying on `PetscOptionsInsertFileYAML()`). For example, the following file:

```yaml
$ans: &ans 42
$eu: &eu 2.72
$pi: &pi 3.14

opt:
  bool: true
  int: *ans
  real: *pi
  imag: 2.72i
  cmplx: -3.14+2.72i
  str: petsc

$seq-bool: [true, false]
$seq-int: [123, 456, 789]
$seq-real: [*pi, *eu]
$seq-str: [abc, ijk, fgh]
```

(continues on next page)
corresponds to the following PETSc options:

- map key0,key1,key2
- map_key0 0
- map_key1 1
- map_key2 2
- opt_bool true
- opt_cmplx -3.14+2.72i
- opt_imag 2.72i
- opt_int 42
- opt_real 3.14
- opt_str petsc
- seq1_bool true,false
- seq1_int 123,456,789
- seq1_real 3.14,2.72
- seq1_str abc,ijk,fgh
- seq2_bool true,false
- seq2_int 123,456,789
- seq2_real 3.14,2.72
- seq2_str rst,uvw,xyz

With -options_file, PETSc will parse the file as YAML if it ends in a standard YAML or JSON extension or if one uses a :yaml postfix, e.g. -options_file my_options.yaml or -options_file my_options.txt:yaml

PETSc will also check the first line of the options file itself and parse the file as YAML if it matches certain criteria, for example.

```yaml
%YAML 1.2
---
name: value
```

and

```yaml
---
name: value
```

both correspond to options

- name value

---

4. JSON is a subset of YAML
User-Defined PetscOptions

Any subroutine in a PETSc program can add entries to the database with the command

```c
PetscOptionsSetValue(PetscOptions options, char *name, char *value);
```

though this is rarely done. To locate options in the database, one should use the commands

```c
PetscOptionsHasName(PetscOptions options, char *pre, char *name, PetscBool *flg);
PetcOptionsGetInt(PetscOptions options, char *pre, char *name, PetscInt *value, _PetscBool *flg);
PetcOptionsGetReal(PetscOptions options, char *pre, char *name, PetscReal *value, _PetscBool *flg);
PetcOptionsGetString(PetscOptions options, char *pre, char *name, char *value, int *maxlen, PetscBool *flg);
PetcOptionsGetStringArray(PetscOptions options, char *pre, char *name, char **values, int *nmax, PetscBool *flg);
PetcOptionsGetIntArray(PetscOptions options, char *pre, char *name, int *value, PetscInt *nmax, PetscBool *flg);
PetcOptionsGetRealArray(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.

3.6.3 Viewers: Looking at PETSc Objects

PETSc employs a consistent scheme for examining, printing, and saving objects through commands of the form

```c
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.

3.6. Other PETSc Features
PETSc/TAO Users Manual, Release 3.15.3

- **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 PetscOptions-GetViewer() 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.

3.6.4 Using SAWs with PETSc

The Scientific Application Web server, SAWs\(^1\), 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.
- **-sawsHttps 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.

\(^1\) Saws wiki on Bitbucket

3.6. Other PETSc Features 243
• -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.

3.6.5 Debugging

PETSc programs may be debugged using one of the two options below.

• -start_in_debugger [noxterm,dbx,xxgdb,xdb,lddb] [-display name] - start all processes in debugger

• -on_error_attach_debugger [noxterm,dbx,xxgdb,xdb,lddb] [-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. Retracing 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.

3.6.6 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 theHandlerContext. The routine

PetscPopErrorHandler()

removes the last error handler and discards it.

PETSc provides two additional error handlers besides PetscTraceBackErrorHandler():

244 Chapter 3. Additional Information
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 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.

3.6. Other PETSc Features 245
PETSc/TAO Users Manual, Release 3.15.3

```c
PetscErrorCode MyRoutine1() {
    /* Declarations Here */
    PetscFunctionBeginUser;
    /* code here */
    PetscFunctionReturn(0);
}
```

### 3.6.7 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.

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.

### 3.6.8 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.

### 3.6.9 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.

---

2 Note that this option is not required to use PETSc with C++
Windows as PetscViewers

For drawing predefined PETSc objects such as matrices and vectors, one may first create a viewer using the command

```c
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

```c
PetscViewerDrawGetDraw(PetscViewer viewer, PetscDraw *draw);
```

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

```c
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

```c
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

```c
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 $0 \leq xl \leq xr \leq 1$ and $0 \leq yl \leq yr \leq 1$.

To draw a line, one uses the command
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

```c
PetscDrawFlush(PetscDraw win);
```

When displaying by using double buffering, which is set with the command

```c
PetscDrawSetDoubleBuffer(PetscDraw win);
```

all processes must call

```c
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

```c
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

```c
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

```c
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

```c
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
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`

```c
static char help[] = "Plots a simple line graph.\n";

#if defined(PETSC_APPLE_FRAMEWORK)
#include <PETSc/petscsys.h>
#include <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;
    (continues on next page)
```

3.6. Other PETSc Features 249
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";

draw = NULL;

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;

ierr = PetscOptionsHasName(NULL, NULL, "-notoplabel", &flg); CHKERRQ(ierr);

if (flg) topplabel = 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);

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);

for (i=0; i<n; i++) {
  xd = (PetscReal)(i - 5); yd = xd*xd;
  ierr = PetscDrawLGAAddPoint(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);

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 draw::draw lg` and `-snes_monitor draw::draw lg`. See also Convergence Monitoring and Convergence Monitoring.

Disabling Graphics at Compile Time

To disable all X-window-based graphics, run `configure` with the additional option `-with-x=0`

3.6.10 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.

3.6. Other PETSc Features
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

\[ \text{M-x visit-tags-table} \]

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.

### 3.6.11 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.

### 3.6.12 Vi and Vim Users

This section lists helpful Vim commands for PETSc. Ones that configure Vim can be placed in a `.vimrc` file in the top of the PETSc directory and will be loaded automatically.

Vi has configurable keymaps: all of the “command mode” commands given that start with a colon (such as `:help`) can be assigned to short sequences in “normal mode,” which is how most Vim users use their most frequently used commands.

See the Emacs Users discussion above for configuration of clangd, which provides integrated development environment.

**Tags**

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

\[ \text{:set tags=CTAGS} \]

from the `$PETSC_DIR` directory and enter the name of the `CTAGS` file. The command `:tag functionname` will cause Vi/Vim to open the file and line number where a desired PETSc function is defined in the current window. `<Ctrl-o>` will return the screen to your previous location.

The command `:stag functionname` will split the current window and then open the file and line number for that function in one half. Some prefer this because it is easier to compare the file you are editing to the function definition this way.
Cscope and gtags

Vim can also use the cscope utility to navigate source code. One useful thing it can do that the basic tags feature can’t is search for references to a symbol, rather than its definition, which can be useful for refactoring. The command

```
:cs find s functionname
```

opens a list of all of the places the function is called in PETSc, and opens the file and line that you choose. The variant `:scs find s functionname` does the same but splits the window like `stag`.

The PETSc makefile does not have a command for building a cscope database, but GNU Global is cross-compatible with cscope: call `make allgtags` to make the gtags database, and run the commands

```
:set csprg=gtags-cscope
:cs add GTAGS
```

Quickfix

Rather than exiting editing a file to build the library and check for errors or warnings, calling `:make` runs the make command without leaving Vim and collects the errors and warnings in a “quickfix” window. Move the cursor to one of the errors or warnings in the quickfix window and press `<Enter>` and the main window will jump to the file and line with the error. The following commands filter lines of out PETSc’s make output that can clutter the quickfix window:

```
:set efm^=%-GStarting\ make\ run\ on\ %.%#
:set efm^=%-GMachine\ characteristics:\ %.%#
:set efm^=%-G#define\ PETSC%.%#
```

Autocompletion and snippets

Autocompletion of long function names can be helpful when working with PETSc. If you have a tags file, you can press `<Ctrl-N>` when you have partially typed a word to bring up a list of potential completions that you can choose from with `<Tab>.

More powerful autocompletion, such as completing the fieldname of a struct, is available from external plugins that can be added to Vim, such as SuperTab, VimCompletesMe, or YouCompleteMe.

Along the same lines, plugins can be added that fill in the boilerplate associated with PETSc programming with code snippets. One such tool is UltiSnips.

LSP for Vim

Several plugins provide the equivalent of emacs’ lsp-mode: YouCompleteMe, mentioned above, is one; another popular one is ale. These can check for syntax errors, check for compilation errors in the background, and provide sophisticated tools for refactoring. Like lsp-mode, they also rely on a compilation database, so `bear -- make -B` should be used as well to generate the file `compile_commands.json`. 

3.6. Other PETSc Features 253
Etc.

See online tutorials for additional Vi/Vim options.

### 3.6.13 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”.

   ```plaintext
   $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`<https://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,

   ```plaintext
   $PETSC_DIR/include
   $PETSC_DIR/$PETSC_ARCH/include
   ```

1. Select “Libraries” under the header Linker and set the library search path:

   ```plaintext
   $PETSC_DIR/$PETSC_ARCH/lib
   ```

   and the libraries, for example
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 “Excluded 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.

### 3.6.14 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.

#### 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:

```plaintext
# 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 functionality
# NOTE: petsc.pc must be in the pkgconfig path. You might need to adjust PKG_CONFIG_PATH
CONFIG += link_pkgconfig
PKGCONFIG += PETSc
```

(continues on next page)
# 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` envirement 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](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/](https://www.qt.io/qt-features-libraries-apis-tools-and-ide/)
3.6.15 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.

3.6.16 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.

3.7 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.

3.7.1 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 columns 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.
3.7.2 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.

```c
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

```c
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

```c
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:

```c
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 info);
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 an 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.
3.7.3 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

\[
\text{KSPCreate(MPI_Comm comm, KSP *kps);}\]

Before using the Krylov context, one must set the matrix-vector multiplication routine and the preconditioner with the commands

\[
\text{PCSetOperators(PC pc, Mat Amat, Mat Pmat);} \\
\text{KSPSetPC(KSP ksp, PC pc);}\]

In addition, the KSP solver must be initialized with

\[
\text{KSPSetUp(KSP ksp);}\]

Solving a linear system is done with the command

\[
\text{KSPSolve(KSP ksp, Vec b, Vec x);}\]

Finally, the KSP context should be destroyed with

\[
\text{KSPDestroy(KSP *ksp);}\]

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.

3.7.4 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

\[
\text{PCCreate(MPI_Comm comm, PC *pc);}\]

The routine
PETSc/TAO Users Manual, Release 3.15.3

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 w, PetscReal dtol,PetscInt maxits,PetscBool zeroguess,PetscInt *outits, PetscInt *outits); 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);

3.8 Running PETSc Tests

3.8.1 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
3.8.2 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.

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.

\(^{11}\) See [https://testanything.org/tap-specification.html](https://testanything.org/tap-specification.html)
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
> 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 of installed PETSc, use

```
make -f ${PETSC_DIR}/share/petsc/examples/gmakefile.test test
```

which will create/use `tests/*` in the current directory for generated test files. You may pass an additional argument `TESTDIR=mytests` to place these generated files somewhere else.

To learn more about the test system details, one can look at the the PETSc developers documentation.

### 3.9 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](https://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 (`KSPLMGRES`) 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`);
PETSc/TAO Users Manual, Release 3.15.3

- Paul Mullowney, enhancements to portions of the NVIDIA GPU interface;
- Dave May - the GCR implementation (KSPGCR);
- Peter Mell - portions of the DM DA 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;
- 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;
- 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,
• 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://cscss.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;

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


