

ANL-21/39 Rev 3.25

PETSc/TAO Users Manual

Revision 3.25

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

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free at OSTI.GOV (http://www.osti.gov/), a service of the US Dept. of Energy's Office of Scientific and Technical Information.

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.

ANL-21/39 Rev 3.25

PETSc/TAO Users Manual

Argonne National Laboratory

Mathematics and Computer Science Division

Prepared by

S. Balay [1], **S. Abhyankar** [1,2], **M. F. Adams** [3], **S. Benson** [1], **J. Brown** [1,10], **P. Brune** [1], **K. Buschelman** [1], **E. M. Constantinescu** [1], **L. Dalcin** [4], **A. Dener** [1], **V. Eijkhout** [6], **J. Faibussowitsch** [1,18], **W. D. Gropp** [1,18], **V. Hapla** [8], **T. Isaac** [1,14], **P. Jolivet** [12,22], **D. Karpeev** [1], **D. Kaushik** [1], **M. G. Knepley** [1,9], **F. Kong** [1,11], **S. Kruger** [15], **D. A. May** [7,21], **L. Curfman McInnes** [1], **R. Tran Mills** [1], **L. Mitchell** [13,20], **T. Munson** [1], **J. E. Roman** [16], **K. Rupp** [1,19], **P. Sanan** [1,8], **J. Sarich** [1], **B. F. Smith** [1,17], **H. Suh** [1], **S. Zampini** [4], **H. Zhang** [1,5], **H. Zhang** [1], and **J. Zhang** [1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory

[2] Electricity Infrastructure and Buildings Division, Pacific Northwest National Laboratory

[3] Computational Research, Lawrence Berkeley National Laboratory

[4] Extreme Computing Research Center, King Abdullah University of Science and Technology

[5] Department of Computer Science, Illinois Institute of Technology

[6] Texas Advanced Computing Center, University of Texas at Austin

[7] Department of Earth Sciences, University of Oxford

[8] Institute of Geophysics, ETH Zurich

[9] Department of Computer Science and Engineering, University at Buffalo

[10] Department of Computer Science, University of Colorado, Boulder

[11] Computational Frameworks, Idaho National Laboratory

[12] Sorbonne Université, CNRS, LIP6

[13] NVIDIA Corporation

[14] College of Computing, Georgia Tech

[15] Tech-X Corporation

[16] DSIC, Universitat Politècnica de València

[17] Flatiron Institute, Simons Foundation

[18] University of Illinois, Urbana-Champaign

[19] Institute for Microelectronics, TU Wien

[20] Department of Computer Science, Durham University

[21] Scripps Institution of Oceanography, University of California, San Diego

[22] Toulouse INP, CNRS, Institute of Computer Science Research

September 28, 2026

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

CONTENTS

1 Introduction to PETSc **1**
- 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
- 1.3 Parallel and GPU Programming . . . 13
  - 1.3.1 MPI Parallelism . . . 14
  - 1.3.2 CPU SIMD parallelism . . . 18
  - 1.3.3 CPU OpenMP parallelism . . . 18
  - 1.3.4 GPU kernel parallelism . . . 19
  - 1.3.5 GPU stream parallelism . . . 20
- 1.4 Compiling and Running Programs . . . 21
- 1.5 Profiling Programs . . . 21
- 1.6 Writing C/C++ or Fortran Applications . . . 23
- 1.7 PETSc's Object-Oriented Design . . . 24
  - 1.7.1 User Callbacks . . . 28
- 1.8 Directory Structure . . . 28

2 The Solvers in PETSc/TAO **31**
- 2.1 Vectors and Parallel Data . . . 31
  - 2.1.1 Creating Vectors . . . 32
  - 2.1.2 Common vector functions and operations . . . 35
  - 2.1.3 Assembling (putting values in) vectors . . . 36
  - 2.1.4 Basic Vector Operations . . . 45
  - 2.1.5 Local/global vectors and communicating between vectors . . . 46
  - 2.1.6 Low-level Vector Communication . . . 47
  - 2.1.7 Application Orderings . . . 52
- 2.2 PetscSF - an alternative to low-level MPI calls for data communication . . . 54
  - 2.2.1 Non-contiguous storage of leafdata . . . 56
  - 2.2.2 GPU usage . . . 57
  - 2.2.3 Gathering leafdata but not reducing it . . . 57
  - 2.2.4 Optimized communication patterns . . . 57
- 2.3 Matrices . . . 57
  - 2.3.1 Creating matrices . . . 58
  - 2.3.2 Low-level matrix creation routines . . . 58
  - 2.3.3 Assembling (putting values into) matrices . . . 58
  - 2.3.4 Basic Matrix Operations . . . 69
  - 2.3.5 Application Specific Custom Matrices . . . 70

2.3.6 Transposes of Matrices . . . 71
2.3.7 Other Matrix Operations . . . 72
2.3.8 Symbolic and Numeric Stages in Sparse Matrix Operations . . . 74
2.3.9 Graph Operations . . . 74
2.3.10 Partitioning . . . 75
2.4 KSP: Linear System Solvers . . . 76
2.4.1 Using KSP . . . 77
2.4.2 Solving Successive Linear Systems . . . 78
2.4.3 Krylov Methods . . . 78
2.4.4 Preconditioners . . . 85
2.4.5 Solving Block Matrices with PCFIELDSPLIT . . . 102
2.4.6 Solving Singular Systems . . . 107
2.4.7 Using External Linear Solvers . . . 107
2.4.8 Using PETSc's MPI parallel linear solvers from a non-MPI program . . . 109
2.5 SNES: Nonlinear Solvers . . . 110
2.5.1 Basic SNES Usage . . . 111
2.5.2 Function Domain Errors and infinity or NaN . . . 118
2.5.3 The Nonlinear Solvers . . . 119
2.5.4 General Options . . . 125
2.5.5 Inexact Newton-like Methods . . . 128
2.5.6 Matrix-Free Methods . . . 128
2.5.7 Finite Difference Jacobian Approximations . . . 143
2.5.8 Variational Inequalities . . . 146
2.5.9 Nonlinear Preconditioning . . . 146
2.6 TS: Scalable ODE and DAE Solvers . . . 147
2.6.1 Basic TS Options . . . 149
2.6.2 DAE Formulations . . . 150
2.6.3 Using Implicit-Explicit (IMEX) Methods . . . 151
2.6.4 IMEX Methods for fast-slow systems . . . 154
2.6.5 GLEE methods . . . 155
2.6.6 Using fully implicit methods . . . 156
2.6.7 Using the Explicit Runge-Kutta timestepper with variable timesteps . . . 156
2.6.8 Special Cases . . . 157
2.6.9 Monitoring and visualizing solutions . . . 157
2.6.10 Error control via variable time-stepping . . . 158
2.6.11 Handling of discontinuities . . . 159
2.6.12 Explicit integrators with finite element mass matrices . . . 159
2.6.13 Performing sensitivity analysis with the TS ODE Solvers . . . 159
2.6.14 Using Sundials from PETSc . . . 163
2.6.15 Using TChem from PETSc . . . 164
2.7 Solving Steady-State Problems with Pseudo-Timestepping . . . 164
2.8 TAO: Optimization Solvers . . . 165
2.8.1 Getting Started: A Simple TAO Example . . . 166
2.8.2 TAO Workflow . . . 167
2.8.3 TAO Algorithms . . . 184
2.8.4 Advanced Options . . . 207
2.8.5 Adding a Solver . . . 209
2.9 PetscRegressor: Regression Solvers . . . 215
2.9.1 Basic Regressor Usage . . . 215
2.9.2 Regression Solvers . . . 217
2.9.3 Linear regressor . . . 217
2.10 PetscDA: Data Assimilation . . . 218
2.10.1 Ensemble-based Data Assimilation . . . 218
2.10.2 Lifecycle overview . . . 218

2.10.3 Creating a PetscDA context . . . 225
2.10.4 Degrees of freedom per grid point . . . 226
2.10.5 Managing ensembles . . . 226
2.10.6 Observation error . . . 226
2.10.7 Analysis step . . . 227
2.10.8 Forecast step . . . 227
2.10.9 Inflation . . . 227
2.10.10 Implementations . . . 228
2.10.11 Command-line options . . . 229
2.10.12 Viewing and monitoring . . . 229
2.10.13 Further reading . . . 229

3 DM: Interfacing Between Solvers and Models/Discretizations 231
3.1 DM Basics . . . 231
3.2 PetscSection: Connecting Grids to Data . . . 232
3.2.1 General concept . . . 232
3.2.2 Multiple Fields . . . 233
3.2.3 Working with data . . . 235
3.2.4 Global Sections: Constrained and Distributed Data . . . 235
3.2.5 Permutation: Changing the order of array data . . . 236
3.2.6 DMPlex Specific Functionality: Obtaining data from the array . . . 237
3.3 DMPlex: Unstructured Grids . . . 237
3.3.1 Representing Unstructured Grids . . . 237
3.3.2 Grid Point Orientations . . . 239
3.3.3 Dealing with Periodicity . . . 239
3.3.4 Connecting Grids to Data Using PetscSection: . . . 240
3.3.5 Data Layout using DMPLEX and PetscFE . . . 242
3.3.6 Partitioning and Ordering . . . 242
3.3.7 Evaluating Residuals . . . 243
3.3.8 Saving and Loading DMPlex Data with HDF5 . . . 245
3.3.9 Metric-based mesh adaptation . . . 249
3.4 DMSTAG: Staggered, Structured Grid . . . 251
3.4.1 Terminology . . . 251
3.4.2 Working with vectors and operators (matrices) . . . 252
3.4.3 Coordinates . . . 254
3.4.4 Numberings and internal data layout . . . 255
3.5 Networks . . . 255
3.5.1 Application flow . . . 257
3.5.2 Utility functions . . . 258
3.5.3 Retrieving components and number of variables . . . 259
3.6 DM Commonalities . . . 260
3.6.1 DMDA simple structured grids . . . 260
3.6.2 DMSTAG simple stagger grids . . . 260
3.6.3 DMPLEX unstructured meshes . . . 261
3.6.4 DMNETWORK computations on graphs of nodes and connecting edges . . . 261
3.7 Is it a programming language issue? . . . 262
3.8 PetscDT: Discretization Technology in PETSc . . . 262
3.8.1 Quadrature . . . 262
3.8.2 Probability Distributions . . . 262
3.9 PetscFE: Finite Element Infrastructure in PETSc . . . 263
3.9.1 Using Pointwise Functions to Specify Finite Element Problems . . . 263
3.9.2 Describing a particular finite element problem to PETSc . . . 263
3.9.3 Assembling finite element residuals and Jacobians . . . 266

4 Additional Information **267**
4.1 PETSc for Fortran Users . . . 267
4.1.1 Fortran and MPI . . . 267
4.1.2 Numerical Constants . . . 267
4.1.3 Basic Fortran API Differences . . . 268
4.1.4 Sample Fortran Programs . . . 274
4.2 Checking the PETSc version . . . 283
4.2.1 During configure/make time . . . 284
4.2.2 During compile time . . . 284
4.2.3 At Runtime . . . 284
4.3 Using MATLAB with PETSc . . . 284
4.3.1 Dumping Data for MATLAB . . . 285
4.3.2 Sending Data to an Interactive MATLAB Session . . . 286
4.3.3 Using the MATLAB Compute Engine . . . 286
4.3.4 Licensing the MATLAB Compute Engine on a cluster . . . 287
4.4 Profiling . . . 291
4.4.1 Basic Profiling Information . . . 291
4.4.2 Profiling Application Codes . . . 297
4.4.3 Profiling Multiple Sections of Code . . . 298
4.4.4 Restricting Event Logging . . . 298
4.4.5 Interpreting `-info` Output: Informative Messages . . . 299
4.4.6 Time . . . 300
4.4.7 Saving Output to a File . . . 300
4.4.8 Accurate Profiling and Paging Overheads . . . 300
4.4.9 NVIDIA Nsight Systems profiling . . . 301
4.4.10 ROCProfiler profiling . . . 301
4.4.11 Using TAU . . . 301
4.5 Hints for Performance Tuning . . . 302
4.5.1 Maximizing Memory Bandwidth . . . 302
4.5.2 Performance Pitfalls and Advice . . . 308
4.6 STREAMS: Example Study . . . 313
4.6.1 Detailed STREAMS study for large arrays . . . 316
4.6.2 Detailed study with application . . . 324
4.6.3 Application with the MPI linear solver server . . . 327
4.7 The Use of BLAS and LAPACK in PETSc and external libraries . . . 327
4.7.1 32 or 64-bit BLAS/LAPACK integers . . . 332
4.7.2 Shared memory BLAS/LAPACK parallelism . . . 332
4.7.3 Available BLAS/LAPACK libraries . . . 333
4.8 Other PETSc Features . . . 333
4.8.1 PETSc on a process subset . . . 333
4.8.2 Runtime Options . . . 333
4.8.3 Viewers: Looking at PETSc Objects . . . 338
4.8.4 Using SAWs with PETSc . . . 340
4.8.5 Debugging . . . 340
4.8.6 Error Handling . . . 341
4.8.7 Numbers . . . 343
4.8.8 Parallel Communication . . . 343
4.8.9 Graphics . . . 343
4.9 Developer Environments . . . 348
4.9.1 Emacs Users . . . 348
4.9.2 VS Code Users . . . 349
4.9.3 Vi and Vim Users . . . 349
4.9.4 Eclipse Users . . . 351
4.9.5 Qt Creator Users . . . 352

4.9.6 Visual Studio Users . . . 354
4.9.7 Xcode IDE Users . . . 354
4.10 Advanced Features of Matrices and Solvers . . . 354
4.10.1 Extracting Submatrices . . . 355
4.10.2 Matrix Factorization . . . 355
4.10.3 Matrix-Matrix Products . . . 358
4.10.4 Creating PC's Directly . . . 358
4.11 Running PETSc Tests . . . 359
4.11.1 Quick start with the tests . . . 359
4.11.2 Understanding test output and more information . . . 360
4.11.3 Using the test harness for your own code . . . 361

Bibliography **363**

CHAPTER

ONE

INTRODUCTION TO PETSC

1.1 About This Manual

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

PETSc/TAO includes a large suite of parallel **linear solvers**, **nonlinear solvers**, **time integrators**, and **optimizers** that may be used in application codes written in Fortran, C, C++, and Python (via petsc4py; see *Getting Started*). The library is organized hierarchically, enabling users to employ the abstraction level 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; it initially has a steeper learning curve than packages such as MATLAB or a simple subroutine library. In particular, for individuals without some 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 makes 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.
- **Several packages (listed on https://petsc.org/),**
  built on PETSc, may satisfy your needs without requiring directly using PETSc. We recommend reviewing these packages' functionality before starting to code directly with PETSc.
- PETSc can be used to provide a "MPI parallel linear solver" in an otherwise sequential or OpenMP parallel code. This approach can provide modest improvements in the application time by utilizing modest numbers of MPI processes. See `PCMPI` for details on how to utilize the PETSc MPI linear solver server.

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 mailing list information on our website for information on contacting support.

1.2 Getting Started

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

- index sets (`IS`), for indexing into vectors, renumbering, permuting, etc;
- *Vectors and Parallel Data* (`Vec`);
- (generally sparse) *Matrices* (`Mat`)
- *KSP: Linear System Solvers* (`KSP`);
- preconditioners, including multigrid, block solvers, patch solvers, and sparse direct solvers (`PC`);
- *SNES: Nonlinear Solvers* (`SNES`);
- *TS: Scalable ODE and DAE Solvers* for solving time-dependent (nonlinear) PDEs, including support for differential-algebraic-equations, and the computation of adjoints (sensitivities/gradients of the solutions) (`TS`);
- scalable *TAO: Optimization Solvers* including a rich set of gradient-based optimizers, Newton-based optimizers and optimization with constraints (`Tao`).
- {any}ch_regressor (`PetscRegressor)`
- *DM Basics* code for managing interactions between mesh data structures and vectors, matrices, and solvers (`DM`);

Each class consists of an abstract interface (simply a set of calling sequences corresponding to an abstract base class in C++) and an implementation for each algorithm and data structure. This design enables easy comparison and use of different algorithms (for example, experimenting 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. 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, enabling users to employ the most appropriate solvers for a particular problem.

1.2.1 Suggested Reading

The manual is divided into four parts:

- *Introduction to PETSc*
- *The Solvers in PETSc/TAO*
- *DM: Interfacing Between Solvers and Models/Discretizations*
- *Additional Information*

Introduction to PETSc describes the basic procedure for using the PETSc library and presents 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.

The Solvers in PETSc/TAO explains in detail the use of the various PETSc algebraic objects, such as vectors, matrices, index sets, and PETSc solvers, including linear and nonlinear solvers, time integrators, and optimization support.

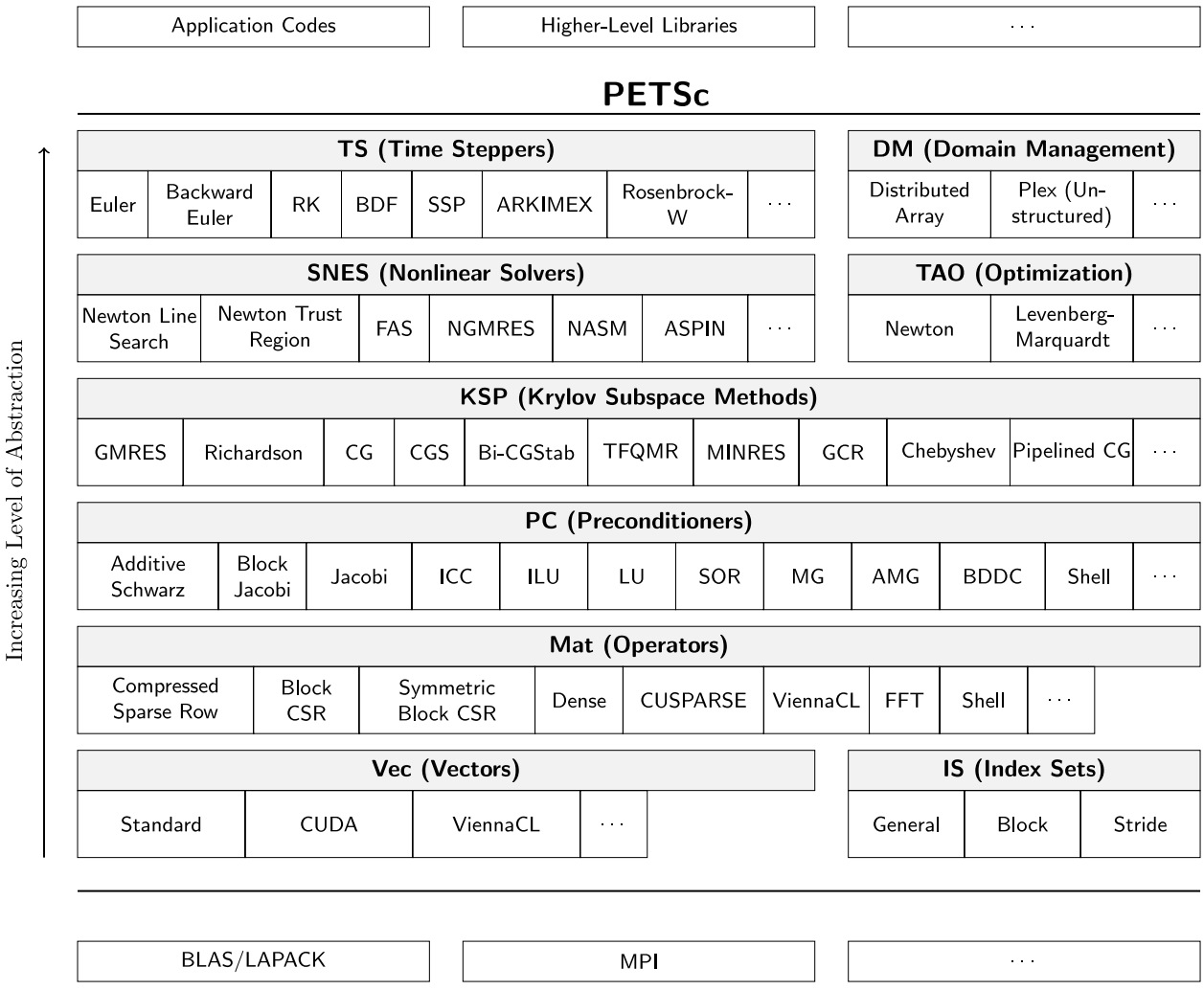


Fig. 1.1: Numerical Libraries in PETSc

DM: Interfacing Between Solvers and Models/Discretizations details how a user's models and discretizations can easily be interfaced with the solvers by using the `DM` construct.

Additional Information describes a variety of useful information, including profiling, the options database, viewers, error handling, and some details of PETSc design.

Visual Studio Code, Eclipse, Emacs, and Vim users may find their development environment's options for searching in the source code are useful for exploring the PETSc source code. Details of this feature are provided in *Developer Environments*.

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` to indicate 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 `$PETSC_ARCH` allows one to switch between several different sets (say debug and optimized versions) of libraries easily. To determine if you need to set `$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`.

See `handson` to immediately jump in and run PETSc code.

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 provides a script that automatically uses the correct `mpiexec` for your configuration.

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

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 and Memory Usage*,

- `-malloc_debug` - enable memory debugging (by default, this is activated for the debugging version of PETSc), see *Detecting Memory Allocation Problems and Memory Usage*,
- `-start_in_debugger [(noxterm)],[(gdb|lldb|...)] [-display name]` - start all (or a subset of the) processes in a 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
- `-version` - display the version of PETSc being used

1.2.3 Writing PETSc Programs

Most PETSc programs begin with a call to

```
PetscInitialize(int *argc,char ***argv,char *file,char *help);
```

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

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

where the file argument is optional.

`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 unfamiliar with MPI, a *communicator* indicates a collection of processes that will be involved 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 programs should call `PetscFinalize()` as their final (or nearly final) statement. 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()`.

Error Checking

Most PETSc functions return a `PetscErrorCode`, an integer indicating whether an error 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 an integer error variable as its final argument.

One should always check these routine values as given below in the C/C++ formats, respectively:

```
PetscCall(PetscFunction(Args));
```

or for Fortran

```
! within the main program
PetscCallA(PetscFunction(Args,ierr))
```

```
! within any subroutine
PetscCall(PetscFunction(Args,ierr))
```

These macros check the returned error code, and if it is nonzero, they call the PETSc error handler and then return from the function with the error code. The macros above should be used on all PETSc calls to enable a complete error traceback. See *Error Checking* for more details on PETSc error handling.

1.2.4 Simple PETSc Examples

To help the user use PETSc immediately, we begin with a simple uniprocessor example that solves the one-dimensional Laplacian problem with finite differences. This sequential code 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: KSP Tutorial src/ksp/ksp/tutorials/ex1.c

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

/*
  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 */
  PetscInt    i, n = 10, col[3], its;
  PetscMPIInt size;
```

(continues on next page)

(continued from previous page)

```
  PetscScalar value[3];

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &args, NULL, help));
  PetscCallMPI(MPI_Comm_size(PETSC_COMM_WORLD, &size));
  PetscCheck(size == 1, PETSC_COMM_WORLD, PETSC_ERR_WRONG_MPI_SIZE, "This is a␣
↪uniprocessor example only!");

  PetscCall(PetscOptionsGetInt(NULL, NULL, "-n", &n, NULL));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         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.
  */
  PetscCall(VecCreate(PETSC_COMM_SELF, &x));
  PetscCall(PetscObjectSetName((PetscObject)x, "Solution"));
  PetscCall(VecSetSizes(x, PETSC_DECIDE, n));
  PetscCall(VecSetFromOptions(x));
  PetscCall(VecDuplicate(x, &b));
  PetscCall(VecDuplicate(x, &u));

  /*
     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.
  */
  PetscCall(MatCreate(PETSC_COMM_SELF, &A));
  PetscCall(MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, n, n));
  PetscCall(MatSetFromOptions(A));
  PetscCall(MatSetUp(A));

  /*
     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;
    PetscCall(MatSetValues(A, 1, &i, 3, col, value, INSERT_VALUES));
  }
  if (n > 1) {
    i      = n - 1;
    col[0] = n - 2;
    col[1] = n - 1;
    PetscCall(MatSetValues(A, 1, &i, 2, col, value, INSERT_VALUES));
```

(continues on next page)

(continued from previous page)

```
  }
  i        = 0;
  col[0]   = 0;
  col[1]   = 1;
  value[0] = 2.0;
  value[1] = -1.0;
  PetscCall(MatSetValues(A, 1, &i, n > 1 ? 2 : 1, col, value, INSERT_VALUES));
  PetscCall(MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY));
  PetscCall(MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY));

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

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
               Create the linear solver and set various options
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  PetscCall(KSPCreate(PETSC_COMM_SELF, &ksp));

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

  /*
     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();
  */
  if (!PCMPIServerActive) { /* cannot directly set KSP/PC options when using the MPI␣
↪linear solver server */
    PetscCall(KSPGetPC(ksp, &pc));
    PetscCall(PCSetType(pc, PCJACOBI));
    PetscCall(KSPSetTolerances(ksp, 1.e-5, PETSC_CURRENT, PETSC_CURRENT, PETSC_
↪CURRENT));
  }

  /*
    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.
  */
  PetscCall(KSPSetFromOptions(ksp));

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

(continues on next page)

(continued from previous page)

```
  PetscCall(KSPSolve(ksp, b, x));

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

  /* check that KSP automatically handles the fact that the new non-zero values in␣
↪the matrix are propagated to the KSP solver */
  PetscCall(MatShift(A, 2.0));
  PetscCall(KSPSolve(ksp, b, x));

  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  PetscCall(KSPDestroy(&ksp));

  /* test if prefixes properly propagate to PCMPI objects */
  if (PCMPIServerActive) {
    PetscCall(KSPCreate(PETSC_COMM_SELF, &ksp));
    PetscCall(KSPSetOptionsPrefix(ksp, "prefix_test_"));
    PetscCall(MatSetOptionsPrefix(A, "prefix_test_"));
    PetscCall(KSPSetOperators(ksp, A, A));
    PetscCall(KSPSetFromOptions(ksp));
    PetscCall(KSPSolve(ksp, b, x));
    PetscCall(KSPDestroy(&ksp));
  }

  PetscCall(VecDestroy(&x));
  PetscCall(VecDestroy(&u));
  PetscCall(VecDestroy(&b));
  PetscCall(MatDestroy(&A));

  /*
     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).
  */
  PetscCall(PetscFinalize());
  return 0;
}
```

Include Files

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

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

where petscksp.h is the include file for the linear solver library. Each PETSc program must specify an include file corresponding 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 *Modules and 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,NULL);` checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged. A complete description of the options database may be found in *Runtime Options*.

Vectors

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

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

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

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

The commands

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

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

Note the use of the PETSc variable type `PetscScalar` in this example. `PetscScalar` is 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

The 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 use the compressed sparse row storage format to store the matrix entries. See MatType for a list of all matrix types. Values can then be set with the command

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

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

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

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

Linear Solvers

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

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

The user first creates the KSP context and sets the operators associated with the system (matrix that defines the linear system, Amat and matrix from which the preconditioner is constructed, Pmat). The user then sets various options for customized solutions, solves the linear system, and finally destroys the KSP context. The command KSPSetFromOptions() enables the user to customize the linear solution method at runtime 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 can also 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

PETSc provides an interface to tackle nonlinear problems called `SNES`. *SNES: Nonlinear Solvers* describes the nonlinear solvers in detail. We highly recommend most PETSc users work directly with `SNES`, rather than using PETSc for the linear problem and writing their own nonlinear solver. Similarly, users should use `TS` rather than rolling their own time integrators.

Error Checking

As noted above, PETSc functions return a `PetscErrorCode`, which is an integer indicating whether an error has occurred during the call. Below, we indicate a traceback generated by error detection within a sample PETSc program. The error occurred on line 3618 of the file `$PETSC_DIR/src/mat/impls/aij/seq/aij.c` and was caused by trying to allocate too large an array in memory. The routine was called in the program `ex3.c` on line 66. See *Error Checking* for details regarding error checking when using the PETSc Fortran interface.

```
$ cd $PETSC_DIR/src/ksp/ksp/tutorials
$ make ex3
$ mpiexec -n 1 ./ex3 -m 100000
[0]PETSC ERROR: --------------------- Error Message --------------------------------
[0]PETSC ERROR: Out of memory. This could be due to allocating
[0]PETSC ERROR: too large an object or bleeding by not properly
[0]PETSC ERROR: destroying unneeded objects.
[0]PETSC ERROR: Memory allocated 11282182704 Memory used by process 7075897344
[0]PETSC ERROR: Try running with -malloc_dump or -malloc_view for info.
[0]PETSC ERROR: Memory requested 18446744072169447424
[0]PETSC ERROR: PETSc Development Git Revision: v3.7.1-224-g9c9a9c5 Git Date: 2016-05-
↪18 22:43:00 -0500
[0]PETSC ERROR: ./ex3 on a arch-darwin-double-debug named Patricks-MacBook-Pro-2.
↪local by patrick Mon Jun 27 18:04:03 2016
[0]PETSC ERROR: Configure options PETSC_DIR=/Users/patrick/petsc PETSC_ARCH=arch-
↪darwin-double-debug --download-mpich --download-f2cblaslapack --with-cc=clang --
↪with-cxx=clang++ --with-fc=gfortran --with-debugging=1 --with-precision=double --
↪with-scalar-type=real --with-viennacl=0 --download-c2html -download-sowing
[0]PETSC ERROR: #1 MatSeqAIJSetPreallocation_SeqAIJ() line 3618 in /Users/patrick/
↪petsc/src/mat/impls/aij/seq/aij.c
[0]PETSC ERROR: #2 PetscTrMallocDefault() line 188 in /Users/patrick/petsc/src/sys/
↪memory/mtr.c
[0]PETSC ERROR: #3 MatSeqAIJSetPreallocation_SeqAIJ() line 3618 in /Users/patrick/
↪petsc/src/mat/impls/aij/seq/aij.c
[0]PETSC ERROR: #4 MatSeqAIJSetPreallocation() line 3562 in /Users/patrick/petsc/src/
↪mat/impls/aij/seq/aij.c
[0]PETSC ERROR: #5 main() line 66 in /Users/patrick/petsc/src/ksp/ksp/tutorials/ex3.c
[0]PETSC ERROR: PETSc Option Table entries:
[0]PETSC ERROR: -m 100000
[0]PETSC ERROR: ----------------End of Error Message ------- send entire error␣
↪message to petsc-maint@mcs.anl.gov----------
```

When running the debug version[1] of the PETSc libraries, it checks 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.

For complete error handling, calls to MPI functions should be made

[1] Configure PETSc with `--with-debugging`.

with `PetscCallMPI(MPI_Function(Args))`. In Fortran subroutines use `PetscCallMPI(MPI_Function(Args, ierr))` and in Fortran main use `PetscCallMPIA(MPI_Function(Args, ierr))`.

PETSc has a small number of C/C++-only macros that do not explicitly return error codes. These are used in the style

```
XXXBegin(Args);
other code
XXXEnd();
```

and include `PetscOptionsBegin()`, `PetscOptionsEnd()`, `PetscObjectOptionsBegin()`, `PetscOptionsHeadBegin()`, `PetscOptionsHeadEnd()`, `PetscDrawCollectiveBegin()`, `PetscDrawCollectiveEnd()`, `MatPreallocateEnd()`, and `MatPreallocateBegin()`. These should not be checked for error codes. Another class of functions with the `Begin()` and `End()` paradigm including `MatAssemblyBegin()`, and `MatAssemblyEnd()` do return error codes that should be checked.

PETSc also has a set of C/C++-only macros that return an object, or `NULL` if an error has been detected. These include `PETSC_VIEWER_STDOUT_WORLD`, `PETSC_VIEWER_DRAW_WORLD`, `PETSC_VIEWER_STDOUT_(MPI_Comm)`, and `PETSC_VIEWER_DRAW_(MPI_Comm)`.

Finally `PetscObjectComm((PetscObject)x)` returns the communicator associated with the object `x` or `MPI_COMM_NULL` if an error was detected.

1.3 Parallel and GPU Programming

Numerical computing today has multiple levels of parallelism (concurrency).

- Low-level, single instruction multiple data (SIMD) parallelism or, somewhat similar, on-GPU parallelism,
- medium-level, multiple instruction multiple data shared memory parallelism (thread parallelism), and
- high-level, distributed memory parallelism.

Traditional CPUs support the lower two levels via, for example, Intel AVX-like instructions (*CPU SIMD parallelism*) and Unix threads, often managed by using OpenMP pragmas (*CPU OpenMP parallelism*), (or multiple processes). GPUs also support the lower two levels via kernel functions (*GPU kernel parallelism*) and streams (*GPU stream parallelism*). Distributed memory parallelism is created by combining multiple CPUs and/or GPUs and using MPI for communication (*MPI Parallelism*).

In addition, there is also concurrency between computations (floating point operations) and data movement (from memory to caches and registers and via MPI between distinct memory nodes).

PETSc supports all these parallelism levels, but its strongest support is for MPI-based distributed memory parallelism.

1.3.1 MPI Parallelism

Since PETSc uses the message-passing model for parallel programming and employs MPI for all interprocessor communication, the user can 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 vector scatter and gather to assist in the management of parallel data.

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

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

The creation routines are collective on all processes 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 process.

The next example, given below, illustrates the solution of a linear system in parallel. This code, corresponding to KSP Tutorial ex2, 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 each processor forms only its local part of the matrix and vectors in the parallel case.

Listing: KSP Tutorial src/ksp/ksp/tutorials/ex2.c

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

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

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

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &args, NULL, help));
  PetscCall(PetscOptionsGetInt(NULL, NULL, "-m", &m, NULL));
  PetscCall(PetscOptionsGetInt(NULL, NULL, "-n", &n, NULL));
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

(continues on next page)

(continued from previous page)

```
        Compute the matrix and right-hand-side vector that define
        the linear system, Ax = b.
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Create parallel matrix, specifying only its global dimensions.
     When using MatCreate(), the matrix format can be specified at
     runtime. Also, the parallel partitioning of the matrix is
     determined by PETSc at runtime.

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

  /*
     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.
  */
  PetscCall(MatGetOwnershipRange(A, &Istart, &Iend));

  /*
     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 = Ii +- n
     instead of J = I +- m as you might expect. The more standard ordering
     would first do all variables for y = h, then y = 2h etc.

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

(continues on next page)

(continued from previous page)

```
    J = Ii - 1;
    PetscCall(MatSetValues(A, 1, &Ii, 1, &J, &v, ADD_VALUES));
  }
  if (j < n - 1) {
    J = Ii + 1;
    PetscCall(MatSetValues(A, 1, &Ii, 1, &J, &v, ADD_VALUES));
  }
  v = 4.0;
  PetscCall(MatSetValues(A, 1, &Ii, 1, &Ii, &v, ADD_VALUES));
}

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

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

/*
   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).
*/
PetscCall(VecCreate(PETSC_COMM_WORLD, &u));
PetscCall(VecSetSizes(u, PETSC_DECIDE, m * n));
PetscCall(VecSetFromOptions(u));
PetscCall(VecDuplicate(u, &b));
PetscCall(VecDuplicate(b, &x));

/*
   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;
*/
PetscCall(VecSet(u, 1.0));
PetscCall(MatMult(A, u, b));

/*
   View the exact solution vector if desired
*/
```

(continues on next page)

(continued from previous page)

```
  flg = PETSC_FALSE;
  PetscCall(PetscOptionsGetBool(NULL, NULL, "-view_exact_sol", &flg, NULL));
  if (flg) PetscCall(VecView(u, PETSC_VIEWER_STDOUT_WORLD));

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

  /*
     Set operators. Here the matrix that defines the linear system
     also serves as the matrix from which the preconditioner is constructed.
  */
  PetscCall(KSPSetOperators(ksp, A, A));

  /*
     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 statement is 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.
  */
  PetscCall(KSPSetTolerances(ksp, 1.e-2 / ((m + 1) * (n + 1)), 1.e-50, PETSC_CURRENT,␣
↪PETSC_CURRENT));

  /*
    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.
  */
  PetscCall(KSPSetFromOptions(ksp));

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

  PetscCall(KSPSolve(ksp, b, x));

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

  /*
     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.
  */
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Norm of error %g iterations %" PetscInt_
```

(continues on next page)

(continued from previous page)

```
↪FMT "\n", (double)norm, its));

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

  /*
     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).
  */
  PetscCall(PetscFinalize());
  return 0;
}
```

1.3.2 CPU SIMD parallelism

SIMD parallelism occurs most commonly in the Intel advanced vector extensions (AVX) families of instructions (see Wikipedia). It may be automatically used by the optimizing compiler or in low-level libraries that PETSc uses, such as BLAS (see BLIS), or rarely, directly in PETSc C/C++ code, as in MatMult_SeqSELL.

1.3.3 CPU OpenMP parallelism

Compute nodes are the building blocks of HPC systems. A compute node (sometimes called a shared-memory node) consists of one or more physical CPUs. A compute node contains multiple cores that have access to a common memory. Within a compute node, the OS might freely migrate OS threads (independent streams of instructions) and processes between cores, unless constrained. Parallel HPC systems consist of multiple compute nodes connected via a network. On conventional clusters, OS threads and processes cannot migrate between compute nodes.

OpenMP parallelism is thread parallelism within a compute node. Multiple threads process data and perform computations on different parts of memory that is shared (accessible) by all threads. The OpenMP model is based on inserting pragmas into code, indicating that a series of instructions (often within a loop) can be run in parallel. This is also called a fork-join model of parallelism since much of the code remains sequential and only the computationally expensive parts in the 'parallel region' are parallel. Thus, OpenMP makes it relatively easy to add some parallelism to a conventional sequential code in a shared memory environment.

POSIX threads (pthreads) is a library that may be called from C/C++. The library contains routines to create, join, and remove threads, plus manage communications and synchronizations between threads. Pthreads is rarely used directly in numerical libraries and applications. Sometimes OpenMP is implemented on top of pthreads.

When using OpenMP parallelism with an MPI code, one must not **over-subscribe** the hardware resources. For example, if MPI already has one MPI process (rank) per hardware core, then using four OpenMP threads

per MPI process will slow the code down since one core must switch back and forth between four OpenMP threads.

For application codes that use certain external packages, including BLAS/LAPACK, SuperLU_DIST, MUMPS, MKL, and SuiteSparse, one can build PETSc and these packages to take advantage of OpenMP by using the configure option `--with-openmp`. The number of OpenMP threads used in the application can be controlled with the PETSc command line option `-omp_num_threads num` or the environmental variable `OMP_NUM_THREADS`. Running a PETSc program with `-omp_view` will display the number of threads used. The default number is often absurdly high for the given hardware, so we recommend always setting it appropriately.

Users can also put OpenMP pragmas into their own code. However, since standard PETSc is not thread-safe, they should not, in general, call PETSc routines from inside the parallel regions.

There is an OpenMP thread-safe subset of PETSc that may be configured for using `--with-threadsafety` (often used along with `--with-openmp` or `--download-concurrencykit`). KSP Tutorial ex61f demonstrates how this may be used with OpenMP. In this mode, one may have individual OpenMP threads that each manage their own (sequential) PETSc objects (each thread can interact only with its own objects). This is useful when one has many small systems (or sets of ODEs) that must be integrated in an "embarrassingly parallel" fashion on multicore systems.

The ./configure option `--with-openmp-kernels` causes some PETSc numerical kernels to be compiled using OpenMP pragmas to take advantage of multiple cores. On each compute node, the product of the number of MPI processes and the number of OpenMP threads per MPI process must not exceed the number of cores; otherwise, the code will slow down dramatically due to **over-subscription**.

PETSc's MPI-based linear solvers may be accessed from a sequential or non-MPI OpenMP program, see *Using PETSc's MPI parallel linear solvers from a non-MPI program*.

See also:

Edward A. Lee, The Problem with Threads, Technical Report No. UCB/EECS-2006-1 January [DOI] 10, 2006

1.3.4 GPU kernel parallelism

GPUs offer at least two levels of clearly defined parallelism. Kernel-level parallelism is much like SIMD parallelism applied to loops; many "iterations" of the loop index run on different hardware in "lock-step". PETSc utilizes this parallelism with three similar but slightly different models:

- CUDA, which is provided by NVIDIA and runs on NVIDIA GPUs
- HIP, provided by AMD, which can, in theory, run on both AMD and NVIDIA GPUs
- and Kokkos, an open-source package that provides a slightly higher-level programming model to utilize GPU kernels.

To utilize this one configures PETSc with either `--with-cuda` or `--with-hip` and, if they plan to use Kokkos, also `--download-kokkos --download-kokkos-kernels`.

In the GPU programming model that PETSc uses, the GPU memory is distinct from the CPU memory. This means that data that resides on the CPU memory must be copied to the GPU (often, this copy is done automatically by the libraries, and the user does not need to manage it) if one wishes to use the GPU computational power on it. This memory copy is slow compared to the GPU speed; hence, it is crucial to minimize these copies. This often translates to trying to do almost all the computation on the GPU and not constantly switching between computations on the CPU and the GPU on the same data.

PETSc utilizes GPUs by providing vector and matrix classes (Vec and Mat) specifically written to run on the GPU. However, since it is difficult to write an entire PETSc code that runs only on the GPU, one can also access and work with (for example, put entries into) the vectors and matrices on the CPU. The vector

classes are VECCUDA, MATAIJCUSPARSE, VECKOKKOS, MATAIJKOKKOS, and VECHIP (matrices are not yet supported by PETSc with HIP).

More details on using GPUs from PETSc will follow in this document.

1.3.5 GPU stream parallelism

Please contribute to this document.

1.4 Compiling and Running Programs

The output below illustrates compiling and running a PETSc program using MPICH on a macOS laptop. Note that different machines will have compilation commands as determined by the configuration process. See *Writing C/C++ or Fortran Applications* for a discussion about how to compile your PETSc programs. Users who are experiencing difficulties linking PETSc programs should refer to the FAQ.

```
$ cd $PETSC_DIR/src/ksp/ksp/tutorials
$ make ex2
/Users/patrick/petsc/arch-debug/bin/mpicc -o ex2.o -c -g3   -I/Users/patrick/petsc/
↪include -I/Users/patrick/petsc/arch-debug/include `pwd`/ex2.c
/Users/patrick/petsc/arch-debug/bin/mpicc -g3  -o ex2 ex2.o  -Wl,-rpath,/Users/
↪patrick/petsc/arch-debug/lib -L/Users/patrick/petsc/arch-debug/lib  -lpetsc -
↪lf2clapack -lf2cblas -lmpifort -lgfortran -lgcc_ext.10.5 -lquadmath -lm -lclang_rt.
↪osx -lmpicxx -lc++ -ldl -lmpi -lpmpi -lSystem
/bin/rm -f ex2.o
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 1 ./ex2
Norm of error 0.000156044 iterations 6
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 2 ./ex2
Norm of error 0.000411674 iterations 7
```

1.5 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 the 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 because the code solved a tiny system. We include this example merely to demonstrate the ease of extracting performance information.

```
$ $PETSC_DIR/lib/petsc/bin/petscmpiexec -n 1 ./ex1 -n 1000 -pc_type ilu -ksp_type␣
↪gmres -ksp_rtol 1.e-7 -log_view
...
------------------------------------------------------------------------------------------
↪----------------------------------
Event                Count      Time (sec)     Flops                             ---␣
↪Global ---  --- Stage ----  Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   AvgLen  Reduct  %T
↪%F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------
↪----------------------------------

VecMDot                1 1.0 3.2830e-06 1.0 2.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪5  0  0  0   0  5  0  0  0   609
VecNorm                3 1.0 4.4550e-06 1.0 6.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0␣
↪14  0  0  0   0 14  0  0  0  1346
VecScale               2 1.0 4.0110e-06 1.0 2.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪5  0  0  0   0  5  0  0  0   499
VecCopy                1 1.0 3.2280e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
VecSet                11 1.0 2.5537e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪0  0  0  0   2  0  0  0  0     0
VecAXPY                2 1.0 2.0930e-06 1.0 4.00e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0␣
↪10  0  0  0   0 10  0  0  0  1911
```

(continues on next page)

(continued from previous page)

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

Memory usage is given in bytes:

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

--- Event Stage 0: Main Stage

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

1.6 Writing C/C++ or Fortran Applications

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`, `TS`, or `TAO`. The manual pages at https://petsc.org/release/documentation/ provide links (organized by routine names and concepts) to the tutorial examples.

To develop an application program that uses PETSc, we suggest the following:

- Download and install PETSc.
- For completely new applications
  1. Make a directory for your source code: for example, `mkdir $HOME/application`
  2. Change to that directory, for example, `cd $HOME/application`
  3. Copy an example in the directory that corresponds to the problems of interest into your directory, for example, `cp $PETSC_DIR/src/snes/tutorials/ex19.c app.c`
  4. Select an application build process. The `PETSC_DIR` (and `PETSC_ARCH` if the `--prefix=directoryname` option was not used when configuring PETSc) environmental variable(s) must be set for any of these approaches.
     - make (recommended). It uses the pkg-config tool and is the recommended approach. Copy $PETSC_DIR/share/petsc/Makefile.user or $PETSC_DIR/share/petsc/Makefile.basic.user to your directory, for example, `cp $PETSC_DIR/share/petsc/Makefile.user makefile`

       Examine the comments in this makefile.

       Makefile.user uses the pkg-config tool and is the recommended approach.

       Use `make app` to compile your program.
     - CMake. Copy $PETSC_DIR/share/petsc/CMakeLists.txt to your directory, for example, `cp $PETSC_DIR/share/petsc/CMakeLists.txt CMakeLists.txt`

       Edit CMakeLists.txt, read the comments on usage, and change the name of the application from ex1 to your application executable name.
  5. Run the program, for example, `./app`
  6. Start to modify the program to develop your application.
- For adding PETSc to an existing application
  1. Start with a working version of your code that you build and run to confirm that it works.
  2. Upgrade your build process. The `PETSC_DIR` (and `PETSC_ARCH` if the `--prefix=directoryname` option was not used when configuring PETSc) environmental variable(s) must be set for any of these approaches.
     - Using make. Update the application makefile to add the appropriate PETSc include directories and libraries.
       * Recommended approach. Examine the comments in $PETSC_DIR/share/petsc/Makefile.user and transfer selected portions of that file to your makefile.
       * Minimalist. Add the line

```
include ${PETSC_DIR}/lib/petsc/conf/variables
```

to the bottom of your makefile. This will provide a set of PETSc-specific make variables you may use in your makefile. See the comments in the file $PETSC_DIR/share/petsc/Makefile.basic.user for details on the usage.

* Simple, but hands the build process over to PETSc's control. Add the lines

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
```

to the bottom of your makefile. See the comments in the file $PETSC_DIR/share/petsc/Makefile.basic.user for details on the usage. Since PETSc's rules now control the build process, you will likely need to simplify and remove much of the material that is in your makefile.

* Not recommended since you must change your makefile for each new configuration/computing system. This approach does not require the environmental variable `PETSC_DIR` to be set when building your application since the information will be hardwired in your makefile. Run the following command in the PETSc root directory to get the information needed by your makefile:

```
$ make getlinklibs getincludedirs getcflags getcxxflags␣
↪getfortranflags getccompiler getfortrancompiler getcxxcompiler
```

All the libraries listed need to be linked into your executable, and the include directories and flags need to be passed to the compiler(s). Usually, this is done by setting `LDFLAGS=<list of library flags and libraries>` and `CFLAGS=<list of -I and other flags>` and `FFLAGS=<list of -I and other flags>` etc in your makefile.

- Using CMake. Update the application CMakeLists.txt by examining the code and comments in $PETSC_DIR/share/petsc/CMakeLists.txt

3. Rebuild your application and ensure it still runs correctly.
4. Add a `PetscInitialize()` near the beginning of your code and `PetscFinalize()` near the end with appropriate include commands (and use statements in Fortran).
5. Rebuild your application and ensure it still runs correctly.
6. Slowly start utilizing PETSc functionality in your code, and ensure that your code continues to build and run correctly.

1.7 PETSc's Object-Oriented Design

Though PETSc has a large API, conceptually, it's rather simple. There are three abstract basic data objects (classes): index sets, `IS`, vectors, `Vec`, and matrices, `Mat`. Plus, a larger number of abstract algorithm objects (classes) starting with: preconditioners, `PC`, Krylov solvers, `KSP`, and so forth.

Let `Object` represent any of these objects. Objects are created with

```
Object obj;
ObjectCreate(MPI_Comm, &obj);
```

The object is initially empty, and little can be done with it. A particular implementation of the class is associated with the object by setting the object's "type", where type is merely a string name of an implementation class using

```
ObjectSetType(obj,"ImplementationName");
```

Some objects support subclasses, which are specializations of the type. These are set with

```
ObjectNameSetType(obj,"ImplementationSubName");
```

For example, within TS one may do

```
TS ts;
TSCreate(PETSC_COMM_WORLD,&ts);
TSSetType(ts,TSARKIMEX);
TSARKIMEXSetType(ts,TSARKIMEX3);
```

The abstract class TS can embody any ODE/DAE integrator scheme. This example creates an additive Runge-Kutta ODE/DAE IMEX integrator, whose type name is TSARKIMEX, using a 3rd-order scheme with an L-stable implicit part, whose subtype name is TSARKIMEX3.

To allow PETSc objects to be runtime configurable, PETSc objects provide a universal way of selecting types (classes) and subtypes at runtime from what is referred to as the PETSc "options database". The code above can be replaced with

```
TS obj;
TSCreate(PETSC_COMM_WORLD,&obj);
TSSetFromOptions(obj);
```

now, both the type and subtype can be conveniently set from the command line

```
$ ./app -ts_type arkimex -ts_arkimex_type 3
```

The object's type (implementation class) or subclass can also be changed at any time simply by calling TSSetType() again (though to override command line options, the call to TSSetType() must be made _after_ TSSetFromOptions()). For example:

```
// (if set) command line options "override" TSSetType()
TSSetType(ts, TSGLLE);
TSSetFromOptions(ts);

// TSSetType() overrides command line options
TSSetFromOptions(ts);
TSSetType(ts, TSGLLE);
```

Since the later call always overrides the earlier call, the second form shown is rarely – if ever – used, as it is less flexible than configuring command line settings.

The standard methods on an object are of the general form.

```
ObjectSetXXX(obj,...);
ObjectGetXXX(obj,...);
ObjectYYY(obj,...);
```

For example

```
TSSetRHSFunction(obj,...)
```

Particular types and subtypes of objects may have their own methods, which are given in the form

```
ObjectNameSetXXX(obj,...);
ObjectNameGetXXX(obj,...);
ObjectNameYYY(obj,...);
```

and

```
ObjectNameSubNameSetXXX(obj,...);
ObjectNameSubNameGetXXX(obj,...);
ObjectNameSubNameYYY(obj,...);
```

where Name and SubName are the type and subtype names (for example, as above `TSARKIMEX` and `3`. Most "set" operations have options database versions with the same names in lower case, separated by underscores, and with the word "set" removed. For example,

```
KSPGMRESSetRestart(obj,30);
```

can be set at the command line with

```
$ ./app -ksp_gmres_restart 30
```

A special subset of type-specific methods is ignored if the type does not match the function name. These are usually setter functions that control some aspect specific to the subtype. Note that we leveraged this functionality in the MPI example above (*MPI Parallelism*) by calling `Mat*SetPreallocation()` for a number of different matrix types. As another example,

```
KSPGMRESSetRestart(obj,30);   // ignored if the type is not KSPGMRES
```

These allow cleaner application code since it does not have many if statements to avoid inactive methods. That is, one does not need to write code like

```
if (type == KSPGMRES) {     // unneeded clutter
  KSPGMRESSetRestart(obj,30);
}
```

Many "get" routines give one temporary access to an object's internal data. They are used in the style

```
XXX xxx;
ObjectGetXXX(obj,&xxx);
// use xxx
ObjectRestoreXXX(obj,&xxx);
```

Objects obtained with a "get" routine should be returned with a "restore" routine, generally within the same function. Objects obtained with a "create" routine should be freed with a "destroy" routine.

There may be variants of the "get" routines that give more limited access to the obtained object. For example,

```
const PetscScalar *x;

// specialized variant of VecGetArray()
VecGetArrayRead(vec, &x);
// one can read but not write with x[]
PetscReal y = 2*x[0];
// don't forget to restore x after you are done with it
VecRestoreArrayRead(vec, &x);
```

Objects can be displayed (in a large number of ways) with

```
ObjectView(obj,PetscViewer viewer);
ObjectViewFromOptions(obj,...);
```

Where `PetscViewer` is an abstract object that can represent standard output, an ASCII or binary file, a graphical window, etc. The second variant allows the user to delay until runtime the decision of what viewer and format to use to view the object or if to view the object at all.

Objects are destroyed with

```
ObjectDestroy(&obj)
```

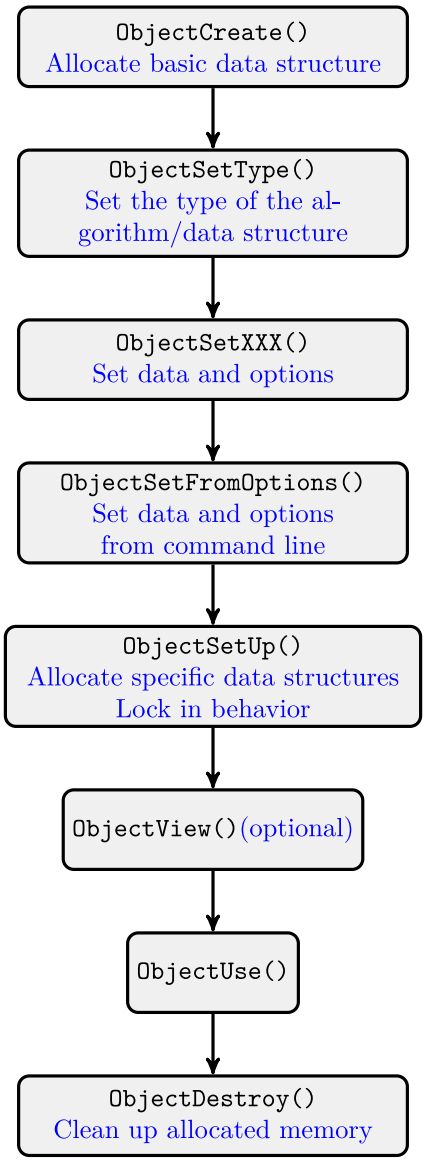


Fig. 1.2: Sample lifetime of a PETSc object

1.7.1 User Callbacks

The user may wish to override or provide custom functionality in many situations. This is handled via callbacks, which the library will call at the appropriate time. The most general way to apply a callback has this form:

```
ObjectCallbackSetter(obj, callbackfunction(), PetscCtx ctx, contextdestroy(PetscCtx␣
↪ctx));
```

where `ObjectCallbackSetter()` is a callback setter such as `SNESSetFunction()`. `callbackfunction()` is what will be called by the library, `ctx` is an optional data structure (array, struct, PETSc object) that is used by `callbackfunction()` and `contextdestroy(PetscCtx ctx)` is an optional function that will be called when `obj` is destroyed to free anything in `ctx`. The use of the `contextdestroy()` allows users to "set and forget" data structures that will not be needed elsewhere but still need to be deleted when no longer needed. Here is an example of the use of a full-fledged callback

```
TS              ts;
TSMonitorLGCtx *ctx;

TSMonitorLGCtxCreate(..., &ctx)
TSMonitorSet(ts, TSMonitorLGTimeStep, ctx, (PetscCtxDestroyFn␣
↪*)TSMonitorLGCtxDestroy);
TSSolve(ts);
```

Occasionally, routines to set callback functions take additional data objects that will be used by the object but are not context data for the function. For example,

```
SNES obj;
Vec  r;
PetscCtx ctx;

SNESSetFunction(snes, r, UserApplyFunction(SNES, Vec, Vec, PetscCtx ctx), ctx);
```

The `r` vector is an optional argument provided by the user, which will be used as work-space by `SNES`. Note that this callback does not provide a way for the user to have the `ctx` destroyed when the `SNES` object is destroyed; the users must ensure that they free it at an appropriate time. There is no logic to the various ways PETSc accepts callback functions in different places in the code.

See *Tao use of PETSc and callbacks* for a cartoon on callbacks in `Tao`.

1.8 Directory Structure

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

- `doc` The source code and Python scripts for building the website and documentation
- `lib/petsc/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 Fortran include files.
- `include/petsc/private` - Private PETSc include files that should *not* need to be used by application programmers.
- `share` - Some small test matrices and other data files

- `src` - The source code for all PETSc libraries, which currently includes
  - `vec` - vectors,
    - `is` - index sets,
  - `mat` - matrices,
  - `ksp` - complete linear equations solvers,
    - `ksp` - Krylov subspace accelerators,
    - `pc` - preconditioners,
  - `snes` - nonlinear solvers
  - `ts` - ODE/DAE solvers and timestepping,
  - `tao` - optimizers,
  - `ml` - Machine Learning
    - `regressor` - Regression solvers
    - `da` - Ensemble data assimilation
  - `dm` - data management between meshes and solvers, vectors, and matrices,
  - `sys` - general system-related routines,
    - `logging` - PETSc logging and profiling routines,
    - `classes` - low-level classes
      - `draw` - simple graphics,
      - `viewer` - a mechanism for printing and visualizing PETSc objects,
      - `bag` - mechanism for saving and loading from disk user data stored in C structs.
      - `random` - random number generators.

Each PETSc source code library directory has the following subdirectories:

- `tutorials` - Programs designed to teach users about PETSc. These codes can serve as templates for applications.
- `tests` - Programs designed for thorough testing of PETSc. As such, these codes are not intended for examination by users.
- `interface` - Provides the abstract base classes for the objects. The code here does not know about particular implementations and does not perform operations on the underlying numerical data.
- `impls` - Source code for one or more implementations of the class for particular data structures or algorithms.
- `utils` - Utility routines. The source here may know about the implementations, but ideally, will not know about implementations for other components.

CHAPTER

TWO

THE SOLVERS IN PETSC/TAO

2.1 Vectors and Parallel Data

Vectors (denoted by `Vec`) are used to store discrete PDE solutions, right-hand sides for linear systems, etc. Users can create and manipulate entries in vectors directly with a basic, low-level interface or they can use the PETSc `DM` objects to connect actions on vectors to the type of discretization and grid that they are working with. These higher-level interfaces handle much of the details of the interactions with vectors and hence, are preferred in most situations. This chapter is organized as follows:

- *Creating Vectors*
  - User managed
  - *DMDA - Creating vectors for structured grids*
  - *DMSTAG - Creating vectors for staggered grids*
  - *DMPLEX - Creating vectors for unstructured grids*
  - *DMNETWORK - Creating vectors for networks*
- Setting vector values
  - For generic vectors
  - *DMDA - Setting vector values*
  - *DMSTAG - Setting vector values*
  - *DMPLEX - Setting vector values*
  - *DMNETWORK - Setting vector values*
- *Basic Vector Operations*
- *Local/global vectors and communicating between vectors*
- *Low-level Vector Communication*
  - *Local to global mappings*
  - *Global Vectors with locations for ghost values*
- *Application Orderings*

2.1.1 Creating Vectors

PETSc provides many ways to create vectors. The most basic, where the user is responsible for managing the parallel distribution of the vector entries, and a variety of higher-level approaches, based on `DM`, for classes of problems such as structured grids, staggered grids, unstructured grids, networks, and particles.

The most basic way to create a vector with a local size of `m` and a global size of `M`, is to use

```
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 type` can be used in conjunction with `VecSetFromOptions()` to specify the use of a particular type of vector. For example, for NVIDIA GPU CUDA, use `cuda`. The GPU-based vectors allow one to set values on either the CPU or GPU but do their computations on the GPU.

We emphasize that all processes in `comm` *must* call the vector creation routines since these routines are collective on all processes in the communicator. If you are unfamiliar with MPI communicators, see the discussion in *Writing PETSc Programs*. In addition, if a sequence of creation routines is used, they must be called in the same order for each process in the communicator.

Instead of, or before calling `VecSetFromOptions()`, one can call

```
VecSetType(Vec v, VecType <VECCUDA, VECHIP, VECKOKKOS, etc.>)
```

One can create vectors whose entries are stored on GPUs using the convenience routine,

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

There are convenience creation routines for almost all vector types; we recommend using the more verbose form because it allows selecting CPU or GPU simulations at runtime.

For applications running in parallel that involve multi-dimensional structured grids, unstructured grids, networks, etc, it is cumbersome for users to explicitly manage the needed local and global sizes of the vectors. Hence, PETSc provides two powerful abstract objects (lower level) `PetscSection` (see *PetscSection: Connecting Grids to Data*) and (higher level) `DM` (see *DM Basics*) to help manage the vectors and matrices needed for such applications. Using `DM`, parallel vectors can be created easily with

```
DMCreateGlobalVector(DM dm, Vec *v)
```

The `DM` object, see *DMDA - Creating vectors for structured grids*, *DMSTAG - Creating vectors for staggered grids*, and *DMPlex: Unstructured Grids* for more details on `DM` for structured grids, staggered structured grids, and for unstructured grids, manages creating the correctly sized parallel vectors efficiently. One controls the type of vector that `DM` creates by calling

```
DMSetVecType(DM dm, VecType vt)
```

or by calling `DMSetFromOptions(DM dm)` and using the option `-dm_vec_type (standard|cuda|kokkos|hip)`.

One can create appropriately sized vectors from `Mat` with `MatCreateVecs()`. One can also create appropriately sized vectors with `KSPCreateVecs()`.

Regardless of how PETSc vectors are created all of their entries are initially zero until a routine such as `VecSet()`, `VecSetRandom()`, `VecSetValues()` or similar routines are called to change the entries. Thus, it is wasteful and unnecessary to call `VecZeroEntries()` on a newly created `Vec`.

DMDA - Creating vectors for structured grids

Each DM type is suitable for a family of problems. The first of these, DMDA are intended for use with *logically structured rectangular grids* when communication of nonlocal data is needed before certain local computations can occur. DMDA is designed only for the case in which data can be thought of as being stored in a standard multidimensional array; thus, DMDA are *not* intended for parallelizing staggered arrays/grids, DMSTAG – *DMSTAG: Staggered, Structured Grid*, or unstructured grid problems, DMPLEX – *DMPlex: Unstructured Grids*, etc.

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, structured 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.

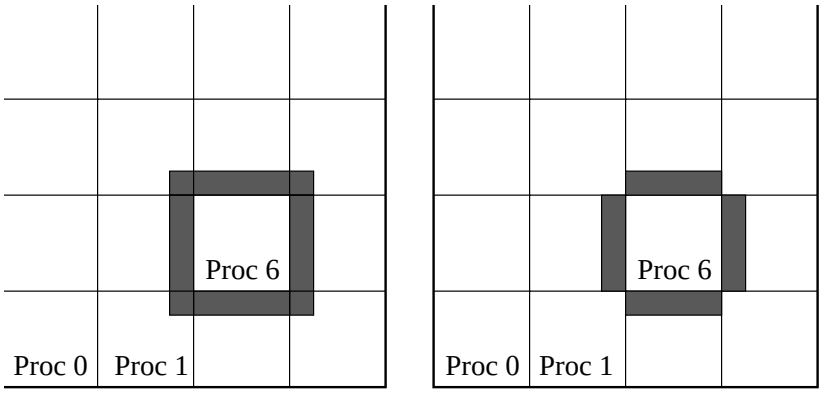


Fig. 2.1: Ghost Points for Two Stencil Types on the Seventh Process

The DMDA object contains parallel data layout information and communication information and is used to create vectors and matrices with the proper layout.

One creates a DMDA two dimensions with the convenience routine

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

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

Two types of DMDA communication data structures can be created, as specified by st. Star-type stencils that radiate outward only in the coordinate directions are indicated by DMDA_STENCIL_STAR, while box-type stencils are specified by DMDA_STENCIL_BOX. For example, for the two-dimensional

case, DMDA_STENCIL_STAR with width 1 corresponds to the standard 5-point stencil, while DMDA_STENCIL_BOX with width 1 denotes the standard 9-point stencil. In both instances, the ghost points are identical, the only difference being that with star-type stencils, certain ghost points are ignored, substantially decreasing the number of messages sent. Note that the DMDA_STENCIL_STAR stencils can save interprocess communication in two and three dimensions.

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

The commands for creating DMDA in one and three dimensions are analogous:

```
DMDACreate1d(MPI_Comm comm, DMBoundaryType xperiod, PetscInt M, PetscInt w, PetscInt␣
↪s, PetscInt *lc, DM *inra);
```

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

The routines to create a DM are collective so that all processes in the communicator comm must call the same creation routines in the same order.

A DM may be created, and its type set with

```
DMCreate(comm,&dm);
DMSetType(dm,"Typename");  // for example, "DMDA"
```

Then DMType specific operations can be performed to provide information from which the specifics of the DM will be provided. For example,

```
DMSetDimension(dm, 1);
DMDASetSizes(dm, M, 1, 1));
DMDASetDof(dm, 1));
DMSetUp(dm);
```

We now very briefly introduce a few more DMType.

DMSTAG - Creating vectors for staggered grids

For structured grids with staggered data (living on elements, faces, edges, and/or vertices), the DMSTAG object is available. It behaves much like DMDA. See *DMSTAG: Staggered, Structured Grid* for discussion of creating vectors with DMSTAG.

DMPLEX - Creating vectors for unstructured grids

See *DMPlex: Unstructured Grids* for a discussion of creating vectors with DMPLEX.

DMNETWORK - Creating vectors for networks

See *Networks* for discussion of creating vectors with DMNETWORK.

2.1.2 Common vector functions and operations

One can examine (print out) a vector with the command

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

To print the vector to the screen, one can use the viewer PETSC_VIEWER_STDOUT_WORLD, which ensures that parallel vectors are printed correctly to stdout. To display the vector in an X-window, one can use the default X-windows viewer PETSC_VIEWER_DRAW_WORLD, or one can create a viewer with the routine PetscViewerDrawOpen(). A variety of viewers are discussed further in *Viewers: Looking at PETSc Objects*.

To create a new vector of the same format and parallel layout as an existing vector, use

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

To create several new vectors of the same format as an existing vector, use

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

This routine creates an array of pointers to vectors. The two routines are 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 create work vectors based on the specified existing vector.

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

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

To destroy an array of vectors, use the command

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

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

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

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

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

The array pointer should be a GPU memory location for GPU vectors.

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

2.1.3 Assembling (putting values in) vectors

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

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

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

```
VecSetValues(Vec x, PetscInt n, PetscInt *indices, PetscScalar *values, INSERT_
↪VALUES);
```

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

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

followed by

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

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

Example usage of `VecSetValues()` may be found in src/vec/vec/tutorials/ex2.c or src/vec/vec/tutorials/exf.F90.

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

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

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

You can call `VecGetValues()` to pull local values from a vector (but not off-process values).

For vectors obtained with `DMCreateGlobalVector()`, one can use `VecSetValuesLocal()` to set values into a global vector but using the local (ghosted) vector indexing of the vector entries. See also *Local to global mappings* that allows one to provide arbitrary local-to-global mapping when not working with a `DM`.

It is also possible to interact directly with the arrays that the vector values are stored in. The routine `VecGetArray()` returns a pointer to the elements local to the process:

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

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

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

For vectors that may also have the array data in GPU memory, for example, VECCUDA, this call ensures the CPU array has the most recent array values by copying the data from the GPU memory if needed.

If the values do not need to be modified, the routines

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

should be used instead.

Listing: SNES Tutorial src/snes/tutorials/ex1.c

```
PetscErrorCode FormFunction1(SNES snes, Vec x, Vec f, PetscCtx ctx)
{
  const PetscScalar *xx;
  PetscScalar       *ff;

  PetscFunctionBeginUser;
  /*
   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.
   */
  PetscCall(VecGetArrayRead(x, &xx));
  PetscCall(VecGetArray(f, &ff));

  /* Compute function */
  ff[0] = xx[0] * xx[0] + xx[0] * xx[1] - 3.0;
  ff[1] = xx[0] * xx[1] + xx[1] * xx[1] - 6.0;

  /* Restore vectors */
  PetscCall(VecRestoreArrayRead(x, &xx));
  PetscCall(VecRestoreArray(f, &ff));
  PetscFunctionReturn(PETSC_SUCCESS);
```

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

For GPU vectors, one can access either the values on the CPU as described above or one can call, for example,

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

Listing: SNES Tutorial src/snes/tutorials/ex47cu.cu

```
    PetscCall(VecCUDAGetArrayRead(xlocal, &xarray));
    PetscCall(VecCUDAGetArrayWrite(f, &farray));
    if (rank) xstartshift = 1;
    else xstartshift = 0;
    if (rank != size - 1) xendshift = 1;
```

(continues on next page)

(continued from previous page)

```
  else xendshift = 0;
  PetscCall(VecGetOwnershipRange(f, &fstart, NULL));
  PetscCall(VecGetLocalSize(x, &lsize));
  // clang-format off
  try {
    thrust::for_each(
      thrust::make_zip_iterator(
        thrust::make_tuple(
          thrust::device_ptr<PetscScalar>(farray),
          thrust::device_ptr<const PetscScalar>(xarray + xstartshift),
          thrust::device_ptr<const PetscScalar>(xarray + xstartshift + 1),
          thrust::device_ptr<const PetscScalar>(xarray + xstartshift - 1),
          thrust::counting_iterator<int>(fstart),
          thrust::constant_iterator<int>(Mx),
          thrust::constant_iterator<PetscScalar>(hx))),
      thrust::make_zip_iterator(
        thrust::make_tuple(
          thrust::device_ptr<PetscScalar>(farray + lsize),
          thrust::device_ptr<const PetscScalar>(xarray + lsize - xendshift),
          thrust::device_ptr<const PetscScalar>(xarray + lsize - xendshift + 1),
          thrust::device_ptr<const PetscScalar>(xarray + lsize - xendshift - 1),
          thrust::counting_iterator<int>(fstart) + lsize,
          thrust::constant_iterator<int>(Mx),
          thrust::constant_iterator<PetscScalar>(hx))),
      ApplyStencil());
  }
```

or

```
VecGetArrayAndMemType(Vec v, PetscScalar **array, PetscMemType *mtype);
```

which, in the first case, returns a GPU memory address and, in the second case, returns either a CPU or GPU memory address depending on the type of the vector. One can then launch a GPU kernel function that accesses the vector's memory for usage with GPUs. When computing on GPUs, `VecSetValues()` is not used! One always accesses the vector's arrays and passes them to the GPU code.

It can also be convenient to treat the vector entries as a Kokkos view. One first creates Kokkos vectors and then calls

```
VecGetKokkosView(Vec v, Kokkos::View<const PetscScalar*, MemorySpace> *kv)
```

to set or access the vector entries.

Of course, to provide the correct values to a vector, one must know what parts of the vector are owned by each MPI process. For parallel vectors, either CPU or GPU-based, it is possible to determine a process's local range with the routine

```
VecGetOwnershipRange(Vec vec, PetscInt *start, PetscInt *end);
```

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

If the `Vec` was obtained from a `DM` with `DMCreateGlobalVector()`, then the range values are determined by the specific `DM`. If the `Vec` was created directly, the range values are determined by the local size passed to `VecSetSizes()` or `VecCreateMPI()`. If `PETSC_DECIDE` was passed as the local size, then the vector

uses default values for the range using `PetscSplitOwnership()`. For certain `DM`, such as `DMDA`, it is better to use `DM` specific routines, such as `DMDAGetGhostCorners()`, to determine the local values in the vector.

Very occasionally, all MPI processes need to know all the range values, these can be obtained with

```
VecGetOwnershipRanges(Vec vec, PetscInt range[]);
```

The number of elements stored locally can be accessed with

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

The global vector length can be determined by

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

DMDA - Setting vector values

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

where `array` is a multidimensional C array with the same dimension as `da`, and

```
DMDAVecGetArrayDOF(DM dm, Vec l, void *array);
... use the array indexing it with 2, 3, or 4 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArrayDOF(DM dm, Vec l, void *array);
DMDAVecGetArrayDOFRead(DM dm, Vec l, void *array);
... use the array indexing it with 2, 3, or 4 dimensions ...
... depending on the dimension of the DMDA ...
DMDAVecRestoreArrayDOFRead(DM dm, Vec l, void *array);
```

where `array` is a multidimensional C array with one more dimension than `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 this array's local and ghost entries as all other entries are undefined. For example, for a scalar problem in two dimensions, one could use

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

Listing: SNES Tutorial src/snes/tutorials/ex3.c

```
PetscErrorCode FormFunction(SNES snes, Vec x, Vec f, PetscCtx ctx)
{
  ApplicationCtx    *user = (ApplicationCtx *)ctx;
  DM                 da   = user->da;
  PetscScalar       *ff, d;
  const PetscScalar *xx, *FF;
  PetscInt           i, M, xs, xm;
  Vec                xlocal;

  PetscFunctionBeginUser;
  PetscCall(DMGetLocalVector(da, &xlocal));
  /*
     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.
  */
  PetscCall(DMGlobalToLocalBegin(da, x, INSERT_VALUES, xlocal));
  PetscCall(DMGlobalToLocalEnd(da, x, INSERT_VALUES, xlocal));

  /*
     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
  */
  PetscCall(DMDAVecGetArrayRead(da, xlocal, (void *)&xx));
  PetscCall(DMDAVecGetArray(da, f, &ff));
  PetscCall(DMDAVecGetArrayRead(da, user->F, (void *)&FF));

  /*
     Get local grid boundaries (for 1-dimensional DMDA):
       xs, xm  - starting grid index, width of local grid (no ghost points)
  */
  PetscCall(DMDAGetCorners(da, &xs, NULL, NULL, &xm, NULL, NULL));
  PetscCall(DMDAGetInfo(da, NULL, &M, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,␣
↪NULL, NULL, NULL));

  /*
     Set function values for boundary points; define local interior grid point range:
        xsi - starting interior grid index
        xei - ending interior grid index
  */
  if (xs == 0) { /* left boundary */
    ff[0] = xx[0];
    xs++;
    xm--;
  }
  if (xs + xm == M) { /* right boundary */
    ff[xs + xm - 1] = xx[xs + xm - 1] - 1.0;
    xm--;
  }

  /*
     Compute function over locally owned part of the grid (interior points only)
```

(continues on next page)

(continued from previous page)

```
  */
  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
  */
  PetscCall(DMDAVecRestoreArrayRead(da, xlocal, (void *)&xx));
  PetscCall(DMDAVecRestoreArray(da, f, &ff));
  PetscCall(DMDAVecRestoreArrayRead(da, user->F, (void *)&FF));
  PetscCall(DMRestoreLocalVector(da, &xlocal));
  PetscFunctionReturn(PETSC_SUCCESS);
```

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

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

and write the residual evaluation using

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

The `DMDAVecGetArray` routines are also provided for GPU access with CUDA, HIP, and Kokkos. For example,

```
DMDAVecGetKokkosOffsetView(DM dm, Vec vec, Kokkos::View<const PetscScalar*XX*,␣
↪MemorySpace> *ov)
```

where `*XX*` can contain any number of `*`. This allows one to write very natural Kokkos multi-dimensional parallel for kernels that act on the local portion of `DMDA` vectors.

Listing: SNES Tutorial src/snes/tutorials/ex3k.kokkos.cxx

```
PetscErrorCode KokkosFunction(SNES snes, Vec x, Vec r, PetscCtx ctx)
{
  ApplicationCtx                  *user = (ApplicationCtx *)ctx;
  DM                               da   = user->da;
  PetscScalar                      d;
  PetscInt                         M;
  Vec                              xl;
  PetscScalarKokkosOffsetView      R;
  ConstPetscScalarKokkosOffsetView X, F;

  PetscFunctionBeginUser;
```

(continues on next page)

(continued from previous page)

```
  PetscCall(DMGetLocalVector(da, &xl));
  PetscCall(DMGlobalToLocal(da, x, INSERT_VALUES, xl));
  d = 1.0 / (user->h * user->h);
  PetscCall(DMDAGetInfo(da, NULL, &M, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,␣
↪NULL, NULL, NULL));
  PetscCall(DMDAVecGetKokkosOffsetView(da, xl, &X));      /* read only */
  PetscCall(DMDAVecGetKokkosOffsetViewWrite(da, r, &R));  /* write only */
  PetscCall(DMDAVecGetKokkosOffsetView(da, user->F, &F)); /* read only */
  Kokkos::parallel_for(
    Kokkos::RangePolicy<>(R.begin(0), R.end(0)), KOKKOS_LAMBDA(int i) {
      if (i == 0) R(0) = X(0);                                                  /*␣
↪left boundary */
      else if (i == M - 1) R(i) = X(i) - 1.0;                                   /*␣
↪right boundary */
      else R(i) = d * (X(i - 1) - 2.0 * X(i) + X(i + 1)) + X(i) * X(i) - F(i); /*␣
↪interior */
    });
  PetscCall(DMDAVecRestoreKokkosOffsetView(da, xl, &X));
  PetscCall(DMDAVecRestoreKokkosOffsetViewWrite(da, r, &R));
  PetscCall(DMDAVecRestoreKokkosOffsetView(da, user->F, &F));
  PetscCall(DMRestoreLocalVector(da, &xl));
  PetscFunctionReturn(PETSC_SUCCESS);
```

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

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

These values can then be used as loop bounds for local function evaluations as demonstrated in the function examples above.

The first version excludes ghost points, while the second 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.

DMSTAG - Setting vector values

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

Listing: SNES Tutorial src/dm/impls/stag/tutorials/ex6.c

```
static PetscErrorCode UpdateVelocity_2d(const Ctx *ctx, Vec velocity, Vec stress, Vec␣
↪buoyancy)
{
  Vec                  velocity_local, stress_local, buoyancy_local;
  PetscInt             ex, ey, startx, starty, nx, ny;
  PetscInt             slot_coord_next, slot_coord_element, slot_coord_prev;
  PetscInt             slot_vx_left, slot_vy_down, slot_buoyancy_down, slot_buoyancy_
↪left;
  PetscInt             slot_txx, slot_tyy, slot_txy_downleft, slot_txy_downright,␣
↪slot_txy_upleft;
  const PetscScalar  **arr_coord_x, **arr_coord_y;
  const PetscScalar ***arr_stress, ***arr_buoyancy;
  PetscScalar       ***arr_velocity;

  PetscFunctionBeginUser;
  /* Prepare direct access to buoyancy data */
  PetscCall(DMStagGetLocationSlot(ctx->dm_buoyancy, DMSTAG_LEFT, 0, &slot_buoyancy_
↪left));
  PetscCall(DMStagGetLocationSlot(ctx->dm_buoyancy, DMSTAG_DOWN, 0, &slot_buoyancy_
↪down));
  PetscCall(DMGetLocalVector(ctx->dm_buoyancy, &buoyancy_local));
  PetscCall(DMGlobalToLocal(ctx->dm_buoyancy, buoyancy, INSERT_VALUES, buoyancy_
↪local));
  PetscCall(DMStagVecGetArrayRead(ctx->dm_buoyancy, buoyancy_local, (void *)&arr_
↪buoyancy));

  /* Prepare read-only access to stress data */
  PetscCall(DMStagGetLocationSlot(ctx->dm_stress, DMSTAG_ELEMENT, 0, &slot_txx));
  PetscCall(DMStagGetLocationSlot(ctx->dm_stress, DMSTAG_ELEMENT, 1, &slot_tyy));
  PetscCall(DMStagGetLocationSlot(ctx->dm_stress, DMSTAG_UP_LEFT, 0, &slot_txy_
↪upleft));
  PetscCall(DMStagGetLocationSlot(ctx->dm_stress, DMSTAG_DOWN_LEFT, 0, &slot_txy_
↪downleft));
  PetscCall(DMStagGetLocationSlot(ctx->dm_stress, DMSTAG_DOWN_RIGHT, 0, &slot_txy_
↪downright));
  PetscCall(DMGetLocalVector(ctx->dm_stress, &stress_local));
  PetscCall(DMGlobalToLocal(ctx->dm_stress, stress, INSERT_VALUES, stress_local));
  PetscCall(DMStagVecGetArrayRead(ctx->dm_stress, stress_local, (void *)&arr_stress));

  /* Prepare read-write access to velocity data */
  PetscCall(DMStagGetLocationSlot(ctx->dm_velocity, DMSTAG_LEFT, 0, &slot_vx_left));
  PetscCall(DMStagGetLocationSlot(ctx->dm_velocity, DMSTAG_DOWN, 0, &slot_vy_down));
  PetscCall(DMGetLocalVector(ctx->dm_velocity, &velocity_local));
  PetscCall(DMGlobalToLocal(ctx->dm_velocity, velocity, INSERT_VALUES, velocity_
↪local));
  PetscCall(DMStagVecGetArray(ctx->dm_velocity, velocity_local, &arr_velocity));

  /* Prepare read-only access to coordinate data */
  PetscCall(DMStagGetProductCoordinateLocationSlot(ctx->dm_velocity, DMSTAG_LEFT, &
↪slot_coord_prev));
```

(continues on next page)

(continued from previous page)

```
  PetscCall(DMStagGetProductCoordinateLocationSlot(ctx->dm_velocity, DMSTAG_RIGHT, &
↪slot_coord_next));
  PetscCall(DMStagGetProductCoordinateLocationSlot(ctx->dm_velocity, DMSTAG_ELEMENT, &
↪slot_coord_element));
  PetscCall(DMStagGetProductCoordinateArrays(ctx->dm_velocity, (void *)&arr_coord_x,␣
↪(void *)&arr_coord_y, NULL));

  /* Iterate over interior of the domain, updating the velocities */
  PetscCall(DMStagGetCorners(ctx->dm_velocity, &startx, &starty, NULL, &nx, &ny, NULL,
↪ NULL, NULL, NULL));
  for (ey = starty; ey < starty + ny; ++ey) {
    for (ex = startx; ex < startx + nx; ++ex) {
      /* Update y-velocity */
      if (ey > 0) {
        const PetscScalar dx = arr_coord_x[ex][slot_coord_next] - arr_coord_
↪x[ex][slot_coord_prev];
        const PetscScalar dy = arr_coord_y[ey][slot_coord_element] - arr_coord_y[ey -␣
↪1][slot_coord_element];
        const PetscScalar B  = arr_buoyancy[ey][ex][slot_buoyancy_down];

        arr_velocity[ey][ex][slot_vy_down] += B * ctx->dt * ((arr_stress[ey][ex][slot_
↪txy_downright] - arr_stress[ey][ex][slot_txy_downleft]) / dx + (arr_
↪stress[ey][ex][slot_tyy] - arr_stress[ey - 1][ex][slot_tyy]) / dy);
      }

      /* Update x-velocity */
```

DMPLEX - Setting vector values

See *DMPlex: Unstructured Grids* for a discussion on setting vector values with DMPLEX.

DMNETWORK - Setting vector values

See *Networks* for a discussion on setting vector values with DMNETWORK.

2.1.4 Basic Vector Operations

Table 2.1: PETSc Vector Operations

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
VecNormEnd(Vec x, NormType NORM_2, PetscReal *norm2);
VecNormEnd(Vec x, NormType NORM_1, PetscReal *norm1);
```

With this code, the communication is delayed until the first call to `VecxxxEnd()` at which a single MPI reduction is used to communicate all the 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()`.

For GPU vectors (like CUDA), the numerical computations will, by default, run on the GPU. Any scalar output, like the result of a `VecDot()` are placed in CPU memory.

2.1.5 Local/global vectors and communicating between vectors

Many PDE problems require ghost (or halo) values in each MPI process or even more general parallel communication of vector values. These values are needed to perform function evaluation on that MPI process. The exact structure of the ghost values needed depends on the type of grid being used. `DM` provides a uniform API for communicating the needed values. We introduce the concept in detail for `DMDA`.

Each `DM` object defines the layout of two vectors: a distributed global vector and a local vector that includes room for the appropriate ghost points. The `DM` object provides information about the size and layout of these vectors. The user can create vector objects that use the `DM` layout information with the routines

```
DMCreateGlobalVector(DM dm, Vec *g);
DMCreateLocalVector(DM dm, 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 `DM` 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 `DM`). The design of the `DM` object makes this easy, as each `DM` operation may operate on vectors of the appropriate size, as obtained via `DMCreateLocalVector()` and `DMCreateGlobalVector()` or as produced by `VecDuplicate()`.

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

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

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

One can scatter the local vectors into the distributed global vector with the command

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

or the commands

```
DMLocalToGlobalBegin(DM dm, Vec l, InsertMode mode, Vec g);
/* (Computation to overlap with communication) */
DMLocalToGlobalEnd(DM dm, 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 access the global vector directly and put in the values.

A third type of `DM` 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 dm, Vec l1, InsertMode iora, Vec l2);
DMLocalToLocalEnd(DM dm, Vec l1, InsertMode iora, Vec l2);
```

Since both local vectors, `l1` and `l2`, must be compatible with `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`.

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

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

2.1.6 Low-level Vector Communication

Most users of PETSc who can utilize a `DM` will not need to utilize the lower-level routines discussed in the rest of this section and should skip ahead to *Matrices*.

To facilitate creating general vector scatters and gathers used, for example, in updating ghost points for problems for which no `DM` currently exists PETSc employs the concept of an *index set*, via the `IS` class. An index set, a generalization of a set of integer indices, is used to define scatters, gathers, and similar operations on vectors and matrices. Much of the underlying code that implements `DMGlobalToLocal` communication is built on the infrastructure discussed below.

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

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

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

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

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

The meaning of `n`, `first`, and `step` correspond to the MATLAB notation `first:step:first+n*step`.

Index sets can be destroyed with the command

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

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

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

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

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

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

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

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

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

Most PETSc applications use a particular `DM` object to manage the communication details needed for their grids. In some rare cases, however, codes need to directly set up their required communication patterns. This is done using PETSc's `VecScatter` and `PetscSF` (for more general data than vectors). 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 a combination of scatters and gathers.

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

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

Here `ix` denotes the index set of the first vector, while `iy` indicates the index set of the destination vector. The vectors can be parallel or sequential. The only requirements are that the number of entries in the index set of the first vector, `ix`, equals the number in the destination index set, `iy`, and that the vectors be long enough to contain all the indices referred to in the index sets. If both `x` and `y` are parallel, their communicator must have the same set of processes, but their process order can differ. 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 the destination with the maximal or minimal of its current value and the scattered values.

To perform a conventional gather operation, the user 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 contributes 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 be 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 backward, switching the roles of the sender and receiver. This is done by using

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

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

Once a VecScatter object has been created, it may be used with any vectors that have the same 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 (the 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().

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
↪VALUES, ScatterMode SCATTER_REVERSE);
VecScatterEnd(VecScatter scatter, Vec localout, Vec globalout, InsertMode ADD_VALUES,␣
↪ScatterMode SCATTER_REVERSE);
```

In this case, the scatter is used in a way similar to the usage of `DMGlobalToLocal()` and `DMLocalToGlobal()` discussed above.

Local to global mappings

When working with a global representation of a vector (usually on a vector obtained with `DMCreateGlobalVector()`) and a local representation of the same vector that includes ghost points required for local computation (obtained with `DMCreateLocalVector()`). PETSc provides routines to help map indices from a local numbering scheme to the PETSc global numbering scheme, recall their use above for the routine `VecSetValuesLocal()` introduced above. This is done via the following routines

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

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

Note that the `ISLocalToGlobalMapping` routines serve a different purpose than the `AO` routines. In the former case, they provide a mapping from a local numbering scheme (including ghost points) to a global numbering scheme, while in the latter, they provide a mapping between two global numbering schemes. 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, to perform function or Jacobian evaluations locally on each process, one works with a local numbering scheme that includes ghost points. The mapping from this local numbering scheme back to the global PETSc numbering can be handled with the `ISLocalToGlobalMapping` routines.

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

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

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

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

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

and then call

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

Now the `indices` use the local numbering rather than the global, meaning the entries lie in $[0, n)$ where n is the local size of the vector. Global vectors obtained from a `DM` already have the global to local mapping provided by the `DM`.

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

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

followed by

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

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

The example SNES Tutorial ex5 illustrates the use of a `DMDA` in the solution of a nonlinear problem. The analogous Fortran program is SNES Tutorial ex5f90; see *SNES: Nonlinear Solvers* for a discussion of the nonlinear solvers.

Global Vectors with locations for ghost values

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

- the extra memory requirement for the local work vector, `localin`, which duplicates the local values in 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.

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

or

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

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

In many ways, a ghosted vector behaves 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 accesses the entries in `l` using the local numbering of elements and ghosts, while they are accessed in `g` using the global numbering.

A common usage of a ghosted vector is given by

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

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

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

`DMPLEX` does not yet support ghosted vectors sharing memory with the global representation. This is a work in progress; if you are interested in this feature, please contact the PETSc community members.

Partitioning discusses the important topic of partitioning an unstructured grid.

2.1.7 Application Orderings

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

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

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

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

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

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

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

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

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

on the first process and

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

on the other process.

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

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

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

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

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

where index sets are used instead of integer arrays.

The mapping routines

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

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

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

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

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

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 dm, AO *ao);
```

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

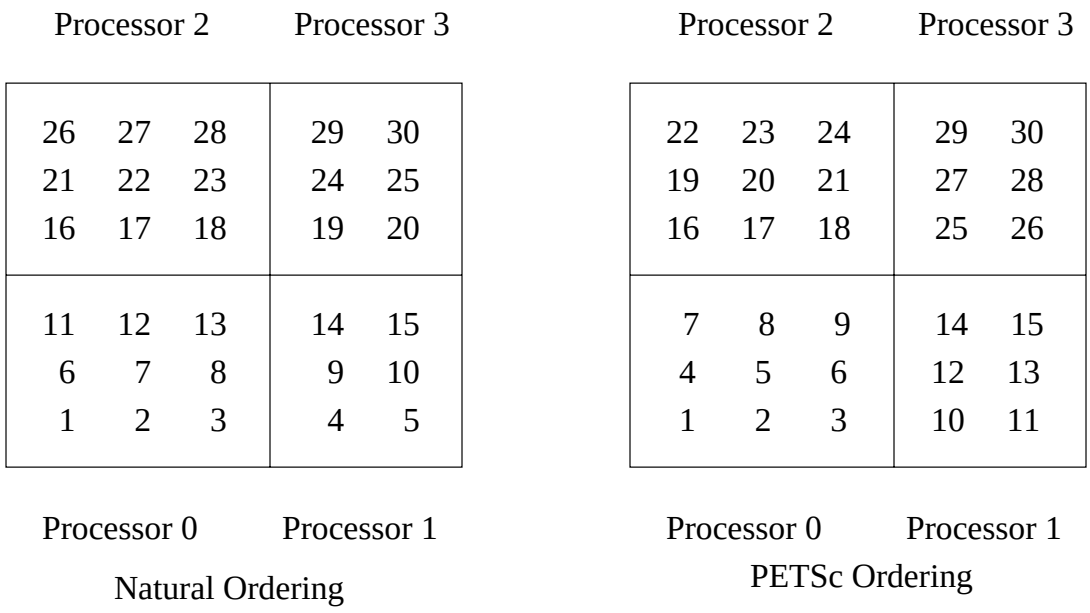


Fig. 2.2: Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processes)

2.2 PetscSF - an alternative to low-level MPI calls for data communication

As discussed above, the `VecScatter` object allows one to define parallel communication between vectors by listing, with `IS` objects, which vector entries from one vector are to be communicated to another vector and where in the second vector they are to be inserted. `PetscSF` provides a similar more general functionality for arrays of any MPI datatype.

`PetscSF` communicates between `rootdata` and `leafdata` arrays. `rootdata` is distributed across the MPI processes and its entries are indicated by a `PetscSFNode` pair consisting of the MPI `rank` the entry is located on and the `index` in the array on that MPI process.

```
typedef struct {
  PetscInt rank;  /* MPI rank of owner */
  PetscInt index; /* Index of node on rank */
} PetscSFNode;
```

Each entry is uniquely owned at that location; in the same way a PETSc global vector has unique MPI process ownership of each entry.

`leafdata` is similar to PETSc local vectors; each MPI process's `leafdata` array can contain "ghost values"

that match values in other locations of the `leafdata` (on the same or different MPI processes). All these matching ghost values share a common root value in `rootdata`.

We begin to explain the use of `PetscSF` with an example. First we construct an array that tells for each leaf entry on that MPI process where its root entry is:

```
PetscInt    nroots, nleaves;
PetscSFNode *roots;

if (rank == 0) {
  // the number of entries in rootdata on this MPI process
  nroots = 2;
  // provide the matching location in rootdata for each entry in leafdata
  nleaves = 3;
  roots[0].rank = 0; roots[0].index = 0;
  roots[1].rank = 1; roots[1].index = 0;
  roots[2].rank = 0; roots[2].index = 1;
} else {
  nroots = 1;
  nleaves = 3;
  roots[0].rank = 0; roots[0].index = 0;
  roots[1].rank = 0; roots[1].index = 1;
  roots[2].rank = 1; roots[1].index = 0;
}
```

Next, we construct the `PetscSF` that encapsulates this information needed for communication:

```
PetscSF sf;

PetscSFCreate(PETSC_COMM_WORLD, &sf);
PetscSFSetFromOptions(sf);
PetscSFSetGraph(sf, nroots, nleaves, NULL, PETSC_OWN_POINTER, roots, PETSC_OWN_
↪POINTER);
PetscSFSetUp(sf);
```

Next we fill `rootdata`:

```
PetscInt    *rootdata, *leafdata;

if (rank == 0) {
  rootdata[0] = 1;
  rootdata[1] = 2;
} else {
  rootdata[0] = 3;
}
```

Finally, we use the `PetscSF` to communicate `rootdata` to `leafdata`:

```
PetscSFBcastBegin(sf, MPIU_INT, rootdata, leafdata, MPI_REPLACE);
PetscSFBcastEnd(sf, MPIU_INT, rootdata, leafdata, MPI_REPLACE);
```

Now `leafdata` on MPI rank 0 contains (1, 3, 2) and on MPI rank 1 contains (1, 2, 3).

It is also possible to move `leafdata` to `rootdata` using

```
PetscSFReduceBegin(sf, MPIU_INT, leafdata, rootdata, MPIU_SUM);
PetscSFReduceEnd(sf, MPIU_INT, leafdata, rootdata, MPIU_SUM);
```

In this case, since the reduction operation performed (the final argument of `PetscSFReduceBegin()`),

is `MPIU_SUM` the final result in each entry of `rootdata` is the sum of the previous value at that location plus all the values it that entries leafs. So `rootdata` on MPI rank 0 contains (3, 6) while on MPI rank 1 it contains (9).

As shown in the example above, `PetscSFBcastBegin()` and `PetscSFBcastEnd()` (as well as other `PetscSF` functions) also take an `MPI_Op` reduction argument, though that is almost always `MPI_REPLACE`.

2.2.1 Non-contiguous storage of leafdata

In the example above we treated the `leafdata` as sitting in a contiguous array with entries from 0 to one less than `nleaves`. This is indicated by the `NULL` argument in the call to `PetscSFSetGraph()`. More generally the `leafdata` array can have entries in it that are not accessed by the `PetscSF` operations. For example,

```
PetscInt *leaves;

if (rank == 0) {
  leaves[0] = 1;
  leaves[1] = 2;
  leaves[2] = 4;
} else {
  leaves[0] = 2;
  leaves[1] = 1;
  leaves[2] = 0;
}
 PetscSFSetGraph(sf, nroots, nleaves, leaves, PETSC_OWN_POINTER, roots, PETSC_OWN_
↪POINTER);
```

means that the three entries of `leafdata` affected by `PetscSF` communication on MPI rank 0 are the array locations (1, 2, 4); meaning also that `leafdata` must be of length at least 5. On MPI rank 1, the arriving values from the three roots listed in `roots` are placed backwards in `leafdata`. Note that providing the `leaves` permutation array on MPI rank 1 is equivalent to listing the three values in `roots` in the opposite order.

If we reran the initial communication with `PetscSFBcastBegin()` and `PetscSFBcastEnd()` using the modified `sf` the resulting values in `leavedata` would be on MPI rank 0 (x, 1, 3, x, 2) and on MPI rank 1 (3, 2, 1) where x indicates the previous value in `leafdata` that was unchanged.

2.2.2 GPU usage

`rootdata` and `leafdata` can live either on CPU memory or GPU memory. The `PetscSF` routines automatically detect the memory type. But the time for the calls to the `CUDA` or `HIP` routines for doing this determination (`cudaPointerGetAttributes()` or `hipPointerGetAttributes()`) is not trivial. To avoid the cost of the check, `PetscSF` provides the routines `PetscSFBcastWithMemTypeBegin()` and `PetscSFReduceWithMemTypeBegin()` where the user provides the memory type information.

2.2.3 Gathering leafdata but not reducing it

One may wish to gather the entries of the `leafdata` for each root but not reduce them to a single value. This is done with

```
PetscSFGatherBegin(sf, MPIU_INT, leafdata, multirootdata);
PetscSFGatherEnd(sf, MPIU_INT, leafdata, multirootdata);
```

Here `multirootdata` is (generally) an array larger than `rootdata` that has enough locations to store the value of each `leaf` of each local root. The values are stored contiguously for each root; that is `multirootdata` will contain

```
(first leaf of first root, second leaf of first root, third leaf of first root, ...,␣
↪last leaf of first root, first leaf of second root, second leaf of second root, ...)
```

The number of leaves for each local root (sometimes called the degree of the root) can be obtained with calls to `PetscSFComputeDegreeBegin()` and `PetscSFComputeDegreeEnd()`.

The data in `multirootdata` can be communicated to `leafdata` using

```
PetscSFScatterBegin(sf, MPIU_INT, multirootdata, leafdata);
PetscSFScatterEnd(sf, MPIU_INT, multirootdata, leafdata);
```

2.2.4 Optimized communication patterns

A performance drawback to using `PetscSFSetGraph()` is that it requires explicitly listing in arrays all the entries of `roots`. `PetscSFSetGraphWithPattern()` provides an alternative way to indicate the communication graph for specific communication patterns.

2.3 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) for CPU and GPU based matrices, as well as several specialized formats. Additional specialized formats can be easily 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.

2.3.1 Creating matrices

As with vectors, PETSc has APIs that allow the user to specify the exact details of the matrix creation process but also DM based creation routines that handle most of the details automatically for specific families of applications. This is done with

```
DMCreateMatrix(DM dm,Mat *A)
```

The type of matrix created can be controlled with either

```
DMSetMatType(DM dm,MatType <MATAIJ or MATBAIJ or MATAIJCUSPARSE etc>)
```

or with

```
DMSetFromOptions(DM dm)
```

and the options database option `-dm_mat_type (aij|baij|aijcusparse|...)` Matrices can be created for CPU usage, for GPU usage and for usage on both the CPUs and GPUs.

The creation of DM objects is discussed in *DMDA - Creating vectors for structured grids*, *DMPLEX - Creating vectors for unstructured grids*, *DMNETWORK - Creating vectors for networks*.

2.3.2 Low-level matrix creation routines

When using a DM is not practical for a particular application one can create matrices directly using

```
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 in *Sparse Matrices*. See the manual pages for further information about available matrix formats.

2.3.3 Assembling (putting values into) matrices

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

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

or

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

This routine inserts or adds a logically dense subblock of dimension `m*n` into the matrix. The integer indices `idxm` and `idxn`, respectively, indicate the global row and column numbers to be inserted. `MatSetValues()` uses the standard C convention, where the row and column matrix indices begin with zero *regardless of the programming language employed*. The array `values` is logically two-dimensional, containing the values that are to be inserted. By default the values are given in row major order, which is the opposite of the

Fortran convention, meaning that the value to be put in row `idxm[i]` and column `idxn[j]` is located in `values[i*n+j]`. To allow the insertion of values in column major order, one can call the command

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

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

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

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

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

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

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

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

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

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

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

If the `Mat` was obtained from a `DM` with `DMCreateMatrix()`, then the range values are determined by the specific `DM`. If the `Mat` was created directly, the range values are determined by the local sizes passed to `MatSetSizes()` or `MatCreateAIJ()` (and such low-level functions for other `MatType`). If `PETSC_DECIDE` was passed as the local size, then the vector uses default values for the range using `PetscSplitOwnership()`. For certain `DM`, such as `DMDA`, it is better to use `DM` specific routines, such as `DMDAGetGhostCorners()`, to determine the local values in the matrix. See *Matrix and Vector Layouts and Storage Locations* for full details on row and column layouts.

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.

For matrices associated with DMDA there is a higher-level interface for providing the numerical values based on the concept of stencils. See the manual page of MatSetValuesStencil() for usage.

For GPUs the routines MatSetPreallocationCOO() and MatSetValuesCOO() should be used for efficient matrix assembly instead of MatSetValues().

We now introduce the various families of PETSc matrices. DMCreateMatrix() manages the preallocation process (introduced below) automatically so many users do not need to worry about the details of the preallocation process.

Matrix and Vector Layouts and Storage Locations

The layout of PETSc matrices across MPI ranks is defined by two things

- the layout of the two compatible vectors in the computation of the matrix-vector product y = A * x and
- the memory where various parts of the matrix are stored across the MPI ranks.

PETSc vectors always have a contiguous range of vector entries stored on each MPI rank. The first rank has entries from 0 to rend1 - 1, the next rank has entries from rend1 to rend2 - 1, etc. Thus the ownership range on each rank is from rstart to rend, these values can be obtained with VecGetOwnershipRange(Vec x, PetscInt * rstart, PetscInt * rend). Each PETSc Vec has a PetscLayout object that contains this information.

All PETSc matrices have two PetscLayouts, they define the vector layouts for y and x in the product, y = A * x. Their ownership range information can be obtained with MatGetOwnershipRange(), MatGetOwnershipRangeColumn(), MatGetOwnershipRanges(), and MatGetOwnershipRangesColumn(). Note that MatCreateVecs() provides two vectors that have compatible layouts for the associated vector.

For most PETSc matrices, excluding MATELEMENTAL and MATSCALAPACK, the row ownership range obtained with MatGetOwnershipRange() also defines where the matrix entries are stored; the matrix entries for rows rstart to rend - 1 are stored on the corresponding MPI rank. For other matrices the rank where each matrix entry is stored is more complicated; information about the storage locations can be obtained with MatGetOwnershipIS(). Note that for most PETSc matrices the values returned by MatGetOwnershipIS() are the same as those returned by MatGetOwnershipRange() and MatGetOwnershipRangeColumn().

The PETSc object PetscLayout contains the ownership information that is provided by VecGetOwnershipRange() and with MatGetOwnershipRange(), MatGetOwnershipRangeColumn(). Each vector has one layout, which can be obtained with VecGetLayout() and MatGetLayouts(). Layouts support the routines PetscLayoutGetLocalSize(), PetscLayoutGetSize(), PetscLayoutGetBlockSize(), PetscLayoutGetRanges(), PetscLayoutCompare() as well as a variety of creation routines. These are used by the Vec and Mat and so are rarely needed directly. Finally PetscSplitOwnership() is a utility routine that does the same splitting of ownership ranges as PetscLayout.

Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the 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 symmetric storage, which are generally much more efficient for problems with multiple degrees of freedom per node) are discussed below. Beginning users need not concern themselves initially with such details and may wish to proceed directly to *Basic Matrix Operations*. However, when an application code progresses to the point of tuning for efficiency and/or generating timing results, it is *crucial* to read this information.

Sequential AIJ Sparse Matrices

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

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

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

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

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

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

Preallocation of Memory for Sequential AIJ Sparse Matrices

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

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

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

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

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

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

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

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

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

1. Allocate integer array `nnz`.
2. Loop over vertices, computing the number of neighbor vertices, which determines the number of nonzeros for the corresponding matrix row(s).
3. Create the sparse matrix via `MatCreateSeqAIJ()` or alternative.
4. Loop over elements, generating matrix entries and inserting in matrix via `MatSetValues()`.

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

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

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

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

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

Parallel AIJ Sparse Matrices

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

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

`A` is the newly created matrix, while the arguments `m`, `M`, and `N`, indicate the number of local rows and the number of global rows and columns, respectively. In the PETSc partitioning scheme, all the matrix columns are local and `n` is the number of columns corresponding to the 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 i^{th} process in that communicator contains the i^{th} block of the matrix.

Preallocation of Memory for Parallel AIJ Sparse Matrices

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

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

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

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

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

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

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

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

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

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

has 2. The corresponding `o` submatrix for the second process is

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

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

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

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

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

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

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

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

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

The routine `MatCreateAIJCUSPARSE()` allows one to create GPU based matrices for NVIDIA systems. `MatCreateAIJKokkos()` can create matrices for use with CPU, OpenMP, NVIDIA, AMD, or Intel based GPU systems.

It is sometimes difficult to compute the required preallocation information efficiently, hence PETSc provides a special `MatType`, `MATPREALLOCATOR` that helps make computing this information more straightforward. One first creates a matrix of this type and then, using the same code that one would use to actually compute the matrices numerical values, calls `MatSetValues()` for this matrix, without needing to provide any preallocation information (one need not provide the matrix numerical values). Once this is complete one uses `MatPreallocatorPreallocate()` to provide the accumulated preallocation information to the actual matrix one will use for the computations. We hope to simplify this process in the future, allowing the removal of `MATPREALLOCATOR`, instead simply allowing the use of its efficient insertion process automatically during the first assembly of any matrix type directly without requiring the detailed preallocation information.

See `doc_matrix` for a table of the matrix types available in PETSc.

Limited-Memory Variable Metric (LMVM) Matrices

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

Table 2.2: PETSc LMVM matrix implementations.

Method	PETSc Type	Name	Property
"Good" Broyden [ref-Gri12]	`MATLMVMBrdn`	`lmvmbrdn`	Square
"Bad" Broyden [ref-Gri12]	`MATLMVMBadBrdn`	`lmvmbadbrdn`	Square
Symmetric Rank-1 [ref-NW06]	`MATLMVMSR1`	`lmvmsr1`	Symmetric
Davidon-Fletcher-Powell (DFP) [ref-NW06]	`MATLMVMDFP`	`lmvmdfp`	SPD
Dense Davidon-Fletcher-Powell (DFP) [ref-EM17b]	`MATLMVMDDFP`	`lmvmddfp`	SPD
Broyden-Fletcher-Goldfarb-Shanno (BFGS) [ref-NW06]	`MATLMVMBFGS`	`lmvmbfgs`	SPD
Dense Broyden-Fletcher-Goldfarb-Shanno (BFGS) [ref-EM17b]	`MATLMVMDBFGS`	`lmvmdbfgs`	SPD
Dense Quasi-Newton	`MATLMVMDQN`	`lmvmdqn`	SPD
Restricted Broyden Family [ref-EM17a]	`MATLMVMSymBrdn`	`lmvmsymbrdn`	SPD
Restricted Broyden Family (full-memory diagonal)	`MATLMVMDiagBrdn`	`lmvmdiagbrdn`	SPD

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

- `MatLMVMAllocate(Mat B, Vec X, Vec F)` – Creates the internal data structures necessary to store nonlinear updates and compute forward/inverse applications. The `X` vector defines the solution space while the `F` defines the function space for the history of updates.
- `MatLMVMUpdate(Mat B, 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 `MatCreateVecs()`, `MatDuplicate()` and `MatCopy()` operations.

Restricted Broyden Family, DFP and BFGS methods, including their dense versions, additionally implement special Jacobian initialization and scaling options available via `-mat_lmvm_scale_type (none|scalar|diagonal)`. We describe these choices below:

- `none` – Sets the initial Jacobian to be equal to the identity matrix. No extra computations are required when obtaining the search direction or updating the approximation. However, the number of function

evaluations required to converge the Newton solution is typically much larger than what is required when using other initializations.

- `scalar` – Defines the initial Jacobian as a scalar multiple of the identity matrix. The scalar value σ is chosen by solving the one dimensional optimization problem

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

where S and Y are the matrices whose columns contain a subset of update vectors s_k and y_k, and $\alpha \in [0, 1]$ is defined by the user via `-mat_lmvm_alpha` and has a different default value for each LMVM implementation (e.g.: default $\alpha = 1$ for BFGS produces the well-known $y_k^T s_k / y_k^T y_k$ scalar initialization). The number of updates to be used in the S and Y matrices is 1 by default (i.e.: the latest update only) and can be changed via `-mat_lmvm_sigma_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.

The dense implementations are numerically equivalent to DFP and BFGS, but they try to minimize memory transfer at the cost of storage [ref-EM17b]. Generally, dense formulations of DFP and BFGS, `MATLMVMDDFP` and `MATLMVMDBFGS`, should be faster than classical recursive versions - on both CPU and GPU. It should be noted that `MatMult` of dense BFGS, and `MatSolve` of dense DFP requires Cholesky factorization, which may be numerically unstable, if a Jacobian option other than `none` is used. Therefore, the default implementation is to enable classical recursive algorithms to avoid the Cholesky factorization. This option can be toggled via `-mat_lbfgs_recursive` and `-mat_ldfp_recursive`.

Dense Quasi-Newton, `MATLMVMDQN` is an implementation that uses `MatSolve` of `MATLMVMDBFGS` for its `MatSolve`, and uses `MatMult` of `MATLMVMDDFP` for its `MatMult`. It can be seen as a hybrid implementation to avoid both recursive implementation and Cholesky factorization, trading numerical accuracy for performances.

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

Dense Matrices

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

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

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

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

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

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

Block Matrices

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

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

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

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

The key to format-independent assembly is the function

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

which provides a "view" `submat` into a matrix `A` that operates in the monolithic global space. The `submat` transforms from the split local space defined by `iscol` to the split local space defined by `isrow`. The index sets specify the parts of the monolithic local space that `submat` should operate in. If a nested matrix format is used, then `MatGetLocalSubMatrix()` finds the nested block and returns it without making any copies. In this case, `submat` is fully functional and has a parallel communicator. If a monolithic matrix format is used, then `MatGetLocalSubMatrix()` returns a proxy matrix on `PETSC_COMM_SELF` that does not provide values or implement `MatMult()`, but does implement `MatSetValuesLocal()` and, if `isrow,iscol` have a constant block size, `MatSetValuesBlockedLocal()`. Note that although `submat` may not be a fully functional matrix and the caller does not even know a priori which communicator it will reside on, it always implements the local assembly functions (which are not collective). The index sets `isrow,iscol` can be obtained using `DMCompositeGetLocalISs()` if `DMCOMPOSITE` is being used. `DMCOMPOSITE` can also be used to create matrices, in which case the `MATNEST` format can be specified using `-prefix_dm_mat_type nest` and `MATAIJ` can be specified using `-prefix_dm_mat_type aij`. See SNES Tutorial ex28 for a simple example using this interface.

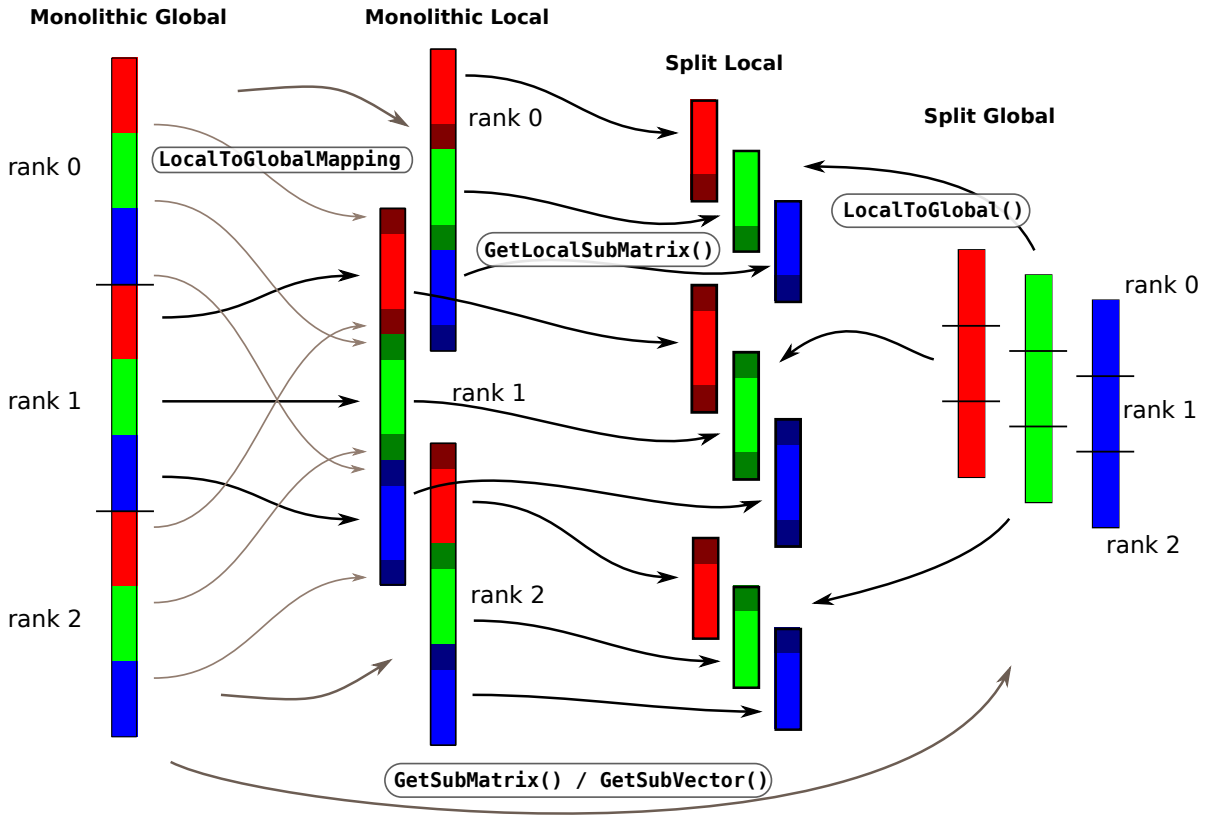


Fig. 2.3: The relationship between spaces used for coupled assembly.

2.3.4 Basic Matrix Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Also one can use

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

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

Table 2.3: PETSc Matrix Operations

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

Table 2.4: Values of MatStructure

Name	Meaning
`SAME_NONZERO_PATTERN`	the matrices have an identical nonzero pattern
`DIFFERENT_NONZERO_PATTERN`	the matrices may have a different nonzero pattern
`SUBSET_NONZERO_PATTERN`	the second matrix has a subset of the nonzeros in the first matrix
`UNKNOWN_NONZERO_PATTERN`	there is nothing known about the relation between the nonzero patterns of the two matrices

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

2.3.5 Application Specific Custom 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. Similarly, users may already have a custom matrix data structure and routines for that data structure and would like to wrap their code up into a `Mat`; that is, provide their own custom matrix type.

To use the PETSc provided matrix-free matrix that uses finite differencing to approximate the matrix-vector product use `MatCreateMFFD()`, see *Matrix-Free Methods*. To provide your own matrix operations (such as `MatMult()`) use the following command to create a `Mat` structure without ever actually generating the matrix:

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

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

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

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

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

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

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

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

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

2.3.6 Transposes of Matrices

PETSc provides several ways to work with transposes of matrix.

```
MatTranspose(Mat A,MatReuse MAT_INITIAL_MATRIX or MAT_INPLACE_MATRIX or MAT_REUSE_
↪MATRIX,Mat *B)
```

will either do an in-place or out-of-place matrix explicit formation of the matrix transpose. After it has been called with `MAT_INPLACE_MATRIX` it may be called again with `MAT_REUSE_MATRIX` and it will recompute the transpose if the A matrix has changed. Internally it keeps track of whether the nonzero pattern of A has not changed so will reuse the symbolic transpose when possible for efficiency.

```
MatTransposeSymbolic(Mat A,Mat *B)
```

only does the symbolic transpose on the matrix. After it is called `MatTranspose()` may be called with `MAT_REUSE_MATRIX` to compute the numerical transpose.

Occasionally one may already have a B matrix with the needed sparsity pattern to store the transpose and wants to reuse that space instead of creating a new matrix by calling `MatTranspose`(A,"MAT_INITIAL_MATRIX",&B) but they cannot just call `MatTranspose`(A,"MAT_REUSE_MATRIX",&B) so instead they can call `MatTransposeSetPrecusor`(A,B) and then call `MatTranspose`(A,"MAT_REUSE_MATRIX",&B). This routine just provides to B the meta-data it needs to compute the numerical factorization efficiently.

The routine `MatCreateTranspose`(A,&B) provides a surrogate matrix B that behaviors like the transpose of A without forming the transpose explicitly. For example, `MatMult`(B,x,y) will compute the matrix-vector

product of A transpose times x.

2.3.7 Other Matrix Operations

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

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

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

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

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

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

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

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

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

or equivalently,

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

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

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

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

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

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

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

or equivalently,

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

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

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

Another matrix routine of interest is

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

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

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

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

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

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

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

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

2.3.8 Symbolic and Numeric Stages in Sparse Matrix Operations

Many sparse matrix operations can be optimized by dividing the computation into two stages: a symbolic stage that creates any required data structures and does all the computations that do not require the matrices' numerical values followed by one or more uses of a numerical stage that use the symbolically computed information. Examples of such operations include `MatTranspose()`, `MatCreateSubMatrices()`, `MatCholeskyFactorSymbolic()`, and `MatCholeskyFactorNumeric()`. PETSc uses two different API's to take advantage of these optimizations.

The first approach explicitly divides the computation in the API. This approach is used, for example, with `MatCholeskyFactorSymbolic()`, `MatCholeskyFactorNumeric()`. The caller can take advantage of their knowledge of changes in the nonzero structure of the sparse matrices to call the appropriate routines as needed. In fact, they can use `MatGetNonzeroState()` to determine if a new symbolic computation is needed. The drawback of this approach is that the caller of these routines has to manage the creation of new matrices when the nonzero structure changes.

The second approach, as exemplified by `MatTranspose()`, does not expose the two stages explicit in the API, instead a flag, `MatReuse` is passed through the API to indicate if a symbolic data structure is already available or needs to be computed. Thus `MatTranspose(A,MAT_INITIAL_MATRIX,&B)` is called first, then `MatTranspose(A,MAT_REUSE_MATRIX,&B)` can be called repeatedly with new numerical values in the A matrix. In theory, if the nonzero structure of A changes, the symbolic computations for B could be redone automatically inside the same B matrix when there is a change in the nonzero state of the A matrix. In practice, in PETSc, the `MAT_REUSE_MATRIX` for most PETSc routines only works if the nonzero structure does not change and the code may crash otherwise. The advantage of this approach (when the nonzero structure changes are handled correctly) is that the calling code does not need to keep track of the nonzero state of the matrices; everything "just works". However, the caller must still know when it is the first call to the routine so the flag `MAT_INITIAL_MATRIX` is being used. If the underlying implementation language supported detecting a yet to be initialized variable at run time, the `MatReuse` flag would not be need.

PETSc uses two approaches because the same programming problem was solved with two different ways during PETSc's early development. A better model would combine both approaches; an explicit separation of the stages and a unified operation that internally utilized the two stages appropriately and also handled changes to the nonzero structure. Code could be simplified in many places with this approach, in most places the use of the unified API would replace the use of the separate stages.

See *Extracting Submatrices* and *Matrix-Matrix Products*.

2.3.9 Graph Operations

PETSc has four families of graph operations that treat sparse `Mat` as representing graphs.

Operation	Type	Available methods	User guide
Ordering to reduce fill	N/A	`MatOrderingType`	*Matrix Factorization*
Partitioning for parallelism	`MatPartitioning`	`MatPartitioningType`	*Partitioning*
Coloring for parallelism or Jacobians	`MatColoring`	`MatColoringType`	*Finite Difference Jacobian Approximations*
Coarsening for multigrid	`MatCoarsen`	`MatCoarsenType`	*Algebraic Multigrid (AMG) Preconditioners*

2.3.10 Partitioning

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

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

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

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

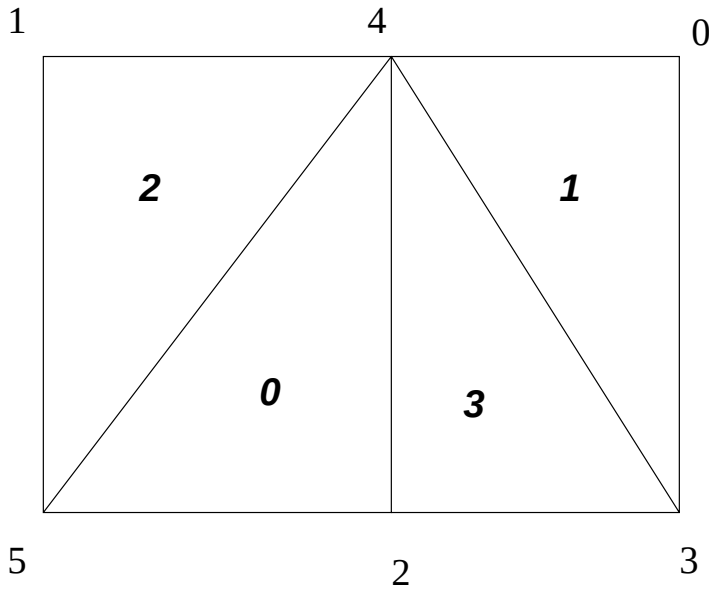


Fig. 2.4: Numbering on Simple Unstructured Grid

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

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

1. partition by element (triangle)

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

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

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

2. partition by vertex.

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

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

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

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

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

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

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

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

See *Finite Difference Jacobian Approximations* and *Matrix Factorization* for discussions on performing graph coloring and computing graph reorderings to reduce fill in sparse matrix factorizations.

2.4 KSP: Linear System Solvers

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

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

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

KSP can also be used to solve least squares problems, using, for example, KSPLSQR. See PETSCREGRESSORLINEAR for tools focusing on linear regression.

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.

2.4.1 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 *Application Specific Custom 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 matrix used to compute the preconditioner 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 right hand size and solution vectors using the command

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

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

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

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

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

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

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

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

before calling KSPSolve() to perform any setup required for the linear solvers. The explicit call of this routine enables the separate profiling 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, KSPGMRES, preconditioned for the uniprocess case with ILU(0), and for the multiprocess case with the block Jacobi method (with one block per process, each of which is solved with ILU(0)). A variety of other solvers and options are also available. To allow application programmers to set any of the preconditioner or Krylov subspace options directly within the code, we provide routines that extract the PC and KSP contexts,

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

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

To solve a linear system with a direct solver (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` (or the equivalent `-ksp_type none`) `-pc_type lu` or `-pc_type cholesky` (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.

2.4.2 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* matrix from which to construct the preconditioner (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* matrix values, because you have changed the matrix values in the Mat objects you passed to KSPSetOperators(), still simply call KPSSolve(). In this case the preconditioner will be recomputed automatically. Use the option `-ksp_reuse_preconditioner true`, or call KSPSetReusePreconditioner(), to reuse the previously computed preconditioner. For many problems, if the matrix changes values only slightly, reusing the old preconditioner can be more efficient.

If you wish to reuse the KSP with a different sized matrix and vectors, you must call KSPReset() before calling KSPSetOperators() with the new matrix.

2.4.3 Krylov Methods

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

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

The type can be one of KSPRICHARDSON, KSPCHEBYSHEV, KSPCG, KSPGMRES, KSPTCQMR, KSPBCGS, KSPCGS, KSPTFQMR, KSPCR, KSPLSQR, KSPBICG, KSPPREONLY (or the equivalent KSPNONE), or others; see *KSP Objects* or the KSPType man page for more. The KSP method can also be set with the options database command `-ksp_type`, followed by one of the options `richardson`, `chebyshev`, `cg`, `gmres`,

tcqmr, bcgs, cgs, tfqmr, cr, lsqr, bicg, preonly (or the equivalent none), or others (see *KSP Objects* or the KSPType man page). There are method-specific options. For instance, for the Richardson, Chebyshev, and GMRES methods:

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

The default parameter values are scale=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 Krylov vectors in GMRES is the unmodified (classical) Gram-Schmidt method, which can be set with

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

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

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

or via the options database with

```
-ksp_gmres_cgs_refinement_type (refine_never|refine_ifneeded|refine_always)
```

The values for KSPGMRESCGSRefinementType() are KSP_GMRES_CGS_REFINE_NEVER, KSP_GMRES_CGS_REFINE_IFNEEDED and KSP_GMRES_CGS_REFINE_ALWAYS.

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

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

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

Note that this option is not valid for all matrices.

Some KSP types do not support preconditioning. For instance, the CGLS algorithm does not involve a preconditioner; any preconditioner set to work with the KSP object is ignored if KSPCGLS 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 (2.1) by

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

where M_L and M_R indicate preconditioning matrices (or, matrices from which the preconditioner is to be constructed). If $M_L = I$ in (2.2), right preconditioning results, and the residual of (2.1),

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

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

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

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

```
KSPSetPCSide(ksp,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
```

Table 2.5: KSP Objects

Method	KSPType	Options Database
Richardson	`KSPRICHARDSON`	`richardson`
Chebyshev	`KSPCHEBYSHEV`	`chebyshev`
Conjugate Gradient [HS52]	`KSPCG`	`cg`
Pipelined Conjugate Gradients [GV14]	`KSPPIPECG`	`pipecg`
Pipelined Conjugate Gradients (Gropp)	`KSPGROPPCG`	`groppcg`
Pipelined Conjugate Gradients with Residual Replacement	`KSPPIPECGRR`	`pipecgrr`
Conjugate Gradients for the Normal Equations	`KSPCGNE`	`cgne`
Flexible Conjugate Gradients [Not00]	`KSPFCG`	`fcg`
Pipelined, Flexible Conjugate Gradients [SSM16]	`KSPPIPEFCG`	`pipefcg`
Conjugate Gradients for Least Squares	`KSPCGLS`	`cgls`
Conjugate Gradients with Constraint (1)	`KSPNASH`	`nash`
Conjugate Gradients with Constraint (2)	`KSPSTCG`	`stcg`
Conjugate Gradients with Constraint (3)	`KSPGLTR`	`gltr`
Conjugate Gradients with Constraint (4)	`KSPQCG`	`qcg`
BiConjugate Gradient	`KSPBICG`	`bicg`
BiCGSTAB [vandVorst92]	`KSPBCGS`	`bcgs`

continues on next page

Table 2.5 – continued from previous page

Method	KSPType	Options Database
Improved BiCGSTAB	KSPIBCGS	ibcgs
QMRCGSTAB [CGS+94]	KSPQMRCGS	qmrcgs
Flexible BiCGSTAB	KSPFBCGS	fbcgs
Flexible BiCGSTAB (variant)	KSPFBCGSR	fbcgsr
Enhanced BiCGSTAB(L)	KSPBCGSL	bcgsl
Minimal Residual Method [PS75]	KSPMINRES	minres
Generalized Minimal Residual [SS86]	KSPGMRES	gmres
Flexible Generalized Minimal Residual [Saa93]	KSPFGMRES	fgmres
Deflated Generalized Minimal Residual	KSPDGMRES	dgmres
Pipelined Generalized Minimal Residual [GAMV13]	KSPPGMRES	pgmres
Pipelined, Flexible Generalized Minimal Residual [SSM16]	KSPPIPEFGMRES	pipefgmres
Generalized Minimal Residual with Accelerated Restart	KSPLGMRES	lgmres
Conjugate Residual [EES83]	KSPCR	cr
Generalized Conjugate Residual	KSPGCR	gcr
Pipelined Conjugate Residual	KSPPIPECR	pipecr
Pipelined, Flexible Conjugate Residual [SSM16]	KSPPIPEGCR	pipegcr
FETI-DP	KSPFETIDP	fetidp
Conjugate Gradient Squared [Son89]	KSPCGS	cgs
Transpose-Free Quasi-Minimal Residual (1) [Fre93]	KSPTFQMR	tfqmr
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr
Least Squares Method	KSPLSQR	lsqr
Symmetric LQ Method [PS75]	KSPSYMMLQ	symmlq
TSIRM	KSPTSIRM	tsirm
Python Shell	KSPPYTHON	python
Shell for no KSP method	KSPNONE	none

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, KSPConvergedDefault(), uses the $ l_2 $ norm of the preconditioned $ B(b - A x) $ or unconditioned residual $ b - Ax$, depending on the KSPType and the value of KSPNormType set with KSPSetNormType. For KSPCG and KSPGMRES the default is the norm of the preconditioned residual. 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_NORM_UNPRECONDITIONED);
```

KSPCG also supports using the natural norm induced by the symmetric positive-definite matrix that defines the linear system with the options database command -ksp_norm_type natural or by calling the routine

```
KSPSetNormType(ksp, KSP_NORM_NATURAL);
```

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

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

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

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

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

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

The user can retain the current value of any of these parameters by specifying `PETSC_CURRENT` as the corresponding tolerance; the defaults are `rtol=1e-5`, `atol=1e-50`, `dtol=1e5`, and `maxits=1e4`. Using `PETSC_DETERMINE` will set the parameters back to their initial values when the object's type was set. These parameters can also be set from the options database with the commands `-ksp_rtol rtol`, `-ksp_atol atol`, `-ksp_divtol dtol`, and `-ksp_max_it maxit`.

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

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

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

Convergence Monitoring

By default, the Krylov solvers, `KSPSolve()`, run silently without displaying information about the iterations. The user can indicate that the norms of the residuals should be displayed at each iteration by using `-ksp_monitor` with the options database. To display the residual norms in a graphical window (running under X Windows), one should use `-ksp_monitor draw::draw_lg`. Application programmers can also provide their own routines to perform the monitoring by using the command

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

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

Many monitoring routines are supplied with PETSc, including

```
KSPMonitorResidual(KSP, PetscInt, PetscReal, PetscCtx);
KSPMonitorSingularValue(KSP, PetscInt, PetscReal, PetscCtx);
KSPMonitorTrueResidual(KSP, PetscInt, PetscReal, PetscCtx);
```

The default monitor simply prints an estimate of a 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 computed via the Lanczos or Arnoldi algorithms.

Since `KSPMonitorTrueResidual()` 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 `KSPSolve()` 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`.

Understanding the Operator's Spectrum

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

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

Then after the `KSP` solve one calls

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

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

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

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

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

Flexible Krylov Methods

Standard Krylov methods require that the preconditioner be a linear operator, thus, for example, a standard KSP method cannot use a KSP in its preconditioner, as is common in the Block-Jacobi method PCBJACOBI, for example. Flexible Krylov methods are a subset of methods that allow (with modest additional requirements on memory) the preconditioner to be nonlinear. For example, they can be used with the PCKSP preconditioner. The flexible KSP methods have the label "Flexible" in *KSP Objects*.

One can use KSPMonitorDynamicTolerance() to control the tolerances used by inner KSP solvers in PCKSP, PCBJACOBI, and PCDEFLATION.

In addition to supporting PCKSP, the flexible methods support KSPFlexibleSetModifyPC() to allow the user to provide a callback function that changes the preconditioner at each Krylov iteration. Its calling sequence is as follows.

```
PetscErrorCode f(KSP ksp, PetscInt total_its, PetscInt its_since_restart, PetscReal␣
↪res_norm, PetscCtx ctx);
```

Pipelined Krylov Methods

Standard Krylov methods have one or more global reductions resulting from the computations of inner products or norms in each iteration. These reductions need to block until all MPI processes have received the results. For a large number of MPI processes (this number is machine dependent but can be above 10,000 processes) this synchronization is very time consuming and can significantly slow the computation. Pipelined Krylov methods overlap the reduction operations with local computations (generally the application of the matrix-vector products and precondtiioners) thus effectively "hiding" the time of the reductions. In addition, they may reduce the number of global synchronizations by rearranging the computations in a way that some of them can be collapsed, e.g., two or more calls to MPI_Allreduce() may be combined into one call. The pipeline KSP methods have the label "Pipeline" in *KSP Objects*.

Special configuration of MPI may be necessary for reductions to make asynchronous progress, which is important for performance of pipelined methods. See doc_faq_pipelined for details.

Other KSP Options

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

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

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

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

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

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

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

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

2.4.4 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 type` or set the method with the command

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

In *PETSc Preconditioners (partial list)* we summarize the basic preconditioning methods supported in PETSc. See the `PCType` manual page for a complete list.

The `PCSHELL` preconditioner allows users to provide their own specific, application-provided custom 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.

PETSc provides several domain decomposition methods/preconditioners including `PCASM`, `PCGASM`, `PCBDDC`, and `PCHPDDM`. In addition PETSc provides multiple multigrid solvers/preconditioners including `PCMG`, `PCGAMG`, `PCHYPRE`, and `PCML`. See further discussion below.

Table 2.6: PETSc Preconditioners (partial list)

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

Each preconditioner may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. Such routine names and commands are all of the form `PCTYPEOPTION` and `-pc_TYPE_OPTION value`, where `TYPE` represents the type name of the object, for example `JACOBI` and `OPTION` represents the name of the option for that type. A complete list can be found by consulting the `PCType` manual page; we discuss just a few in the sections below.

ILU and ICC Preconditioners

Some of the options for ILU preconditioner are

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

Note here that all the factorization based preconditioners share some common options indicated by `PCFactorXXX()`. 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 provides only a sequential SOR preconditioner; it can only be used with sequential matrices or as the subblock preconditioner when using block Jacobi or ASM preconditioning (see below).

The options for SOR preconditioning with `PCSOR` are

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

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

SOR_LOCAL_FORWARD_SWEEP, SOR_LOCAL_BACKWARD_SWEEP, and SOR_LOCAL_SYMMETRIC_SWEEP. These variants can also be set with the options -pc_sor_omega omega, -pc_sor_its its, -pc_sor_lits lits, -pc_sor_backward, -pc_sor_symmetric, -pc_sor_local_forward, -pc_sor_local_backward, and -pc_sor_local_symmetric.

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

LU Factorization

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

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

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

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

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

The sparse LU factorization provided in PETSc does not perform pivoting for numerical stability (since they are designed to preserve nonzero structure), and thus occasionally an 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 (domain decomposition) methods in PETSc are supported in parallel; however, only the uniprocess version of the block Gauss-Seidel method is available. By default, the PETSc implementations of these methods employ ILU(0) factorization on each individual block (that is, the default solver on each subblock is `PCType=PCILU`, `KSPType=KSPPREONLY` (or equivalently `KSPType=KSPNONE`); 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 [CS99]. `PC_ASM_RESTRICT` is the PETSc default, as it saves substantial communication and for many problems has the added benefit of requiring fewer iterations for convergence than the standard additive Schwarz method.

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

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

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

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

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

`PCGASM` is a generalization of `PCASM` that allows the user to specify subdomains that span multiple MPI processes. This can be useful for problems where small subdomains result in poor convergence. To be effective, the multi-processor subproblems must be solved using a sufficiently strong subsolver, such as `PCLU`, 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 processes. 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` matrix used to compute the preconditioner – 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` processes. In general this might not result in a very good subproblem if the single-rank problems corresponding to the coalesced processes 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 process, 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 three external AMG packages: *hypre*, *ML* and *AMGx* (CUDA platforms only) that can be downloaded in the configuration phase (e.g., `--download-hypre`) and used by specifying that command line parameter (e.g., `-pc_type hypre`). *Hypre* is relatively monolithic in that a PETSc matrix is converted into a hypre matrix, and then *hypre* is called to solve the entire problem. *ML* is more modular because PETSc only has *ML* generate the coarse grid spaces (columns of the prolongation operator), which is the core of an AMG method, and then constructs a `PCMG` with Galerkin coarse grid operator construction. `PCGAMG` 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 (see *Multigrid Preconditioners*). PETSc provides a fully supported (smoothed) aggregation AMG, but supports the addition of new methods (`-pc_type gamg -pc_gamg_type agg` or `PCSetType(pc,PCGAMG)` and `PCGAMGSetType(pc, PCGAMGAGG)`. Examples of extension are reference implementations of a classical AMG method (`-pc_gamg_type classical`), a (2D) hybrid geometric AMG method (`-pc_gamg_type geo`) that are not supported. A 2.5D AMG method DofColumns [ISG15] supports 2D coarsenings extruded in the third dimension. `PCGAMG` does require the use of `MATAIJ` matrices. For instance, `MATBAIJ` matrices are not supported. One can use `MATAIJ` instead of `MATBAIJ` without changing any code other than the constructor (or the `-mat_type` from the command line). For instance, `MatSetValuesBlocked` works with `MATAIJ` matrices.

Important parameters for PCGAMGAGG

- Control the generation of the coarse grid
  - `-pc_gamg_aggressive_coarsening n` Use aggressive coarsening on the finest `n` levels to construct the coarser mesh. The default is only on the finest level. See `PCGAMGAGGSetNSmooths()`. The larger value produces a faster preconditioner to create and solve, but the convergence may be slower.
  - `-pc_gamg_low_memory_threshold_filter (true|false)` Filter small matrix entries before coarsening the mesh. See `PCGAMGSetLowMemoryFilter()`.
  - `-pc_gamg_threshold tol` The threshold of small values to drop when `-pc_gamg_low_memory_threshold_filter` is used. A negative value means keeping even the locations with 0.0. See `PCGAMGSetThreshold()`
  - `-pc_gamg_threshold_scale scale` Set a scale factor applied to each coarser level when `-pc_gamg_low_memory_threshold_filter` is used. See `PCGAMGSetThresholdScale()`.
  - `-pc_gamg_mat_coarsen_type (mis|hem|misk)` Algorithm used to coarsen the matrix graph. See `MatCoarsenSetType()`.
  - `-pc_gamg_mat_coarsen_max_it it` Maximum HEM iterations to use. See `MatCoarsenSetMaximumIterations()`.
  - `-pc_gamg_aggressive_mis_k k` the k distance in MIS coarsening (>2 is 'aggressive') to use in coarsening. See `PCGAMGMISkSetAggressive()`. The larger value produces a preconditioner that is faster to create and solve with but the convergence may be slower. This option and the previous option work to determine how aggressively the grids are coarsened.
  - `-pc_gamg_mis_k_minimum_degree_ordering (true|false)` Use a minimum degree ordering in the greedy MIS algorithm used to coarsen. See `PCGAMGMISkSetMinDegreeOrdering()`
- Control the generation of the prolongation for `PCGAMGAGG`
  - `-pc_gamg_agg_nsmooths n` Number of smoothing steps to be used in constructing the prolongation. For symmetric problems, generally, one or more is best. For some strongly nonsymmetric problems, 0 may be best. See `PCGAMGSetNSmooths()`.

- Control the amount of parallelism on the levels
  - `-pc_gamg_process_eq_limit n` Sets the minimum number of equations allowed per process when coarsening (otherwise, fewer MPI processes are used for the coarser mesh). A larger value will cause the coarser problems to be run on fewer MPI processes, resulting in less communication and possibly a faster time to solution. See `PCGAMGSetProcEqLim()`.
  - `-pc_gamg_rank_reduction_factors rn,rn-1,...,r1` Set a schedule for MPI rank reduction on coarse grids. See `PCGAMGSetRankReductionFactors()` This overrides the lessening of processes that would arise from `-pc_gamg_process_eq_limit`.
  - `-pc_gamg_repartition (true|false)` Run a partitioner on each coarser mesh generated rather than using the default partition arising from the finer mesh. See `PCGAMGSetRepartition()`. This increases the preconditioner setup time but will result in less time per iteration of the solver.
  - `-pc_gamg_parallel_coarse_grid_solver (true|false)` Allow the coarse grid solve to run in parallel, depending on the value of `-pc_gamg_coarse_eq_limit`. See `PCGAMGSetParallelCoarseGridSolve()`. If the coarse grid problem is large then this can improve the time to solution.
    - `-pc_gamg_coarse_eq_limit n` Sets the minimum number of equations allowed per process on the coarsest level when coarsening (otherwise fewer MPI processes will be used). A larger value will cause the coarse problems to be run on fewer MPI processes. This only applies if `-pc_gamg_parallel_coarse_grid_solver` is set to true. See `PCGAMGSetCoarseEqLim()`.
- Control the smoothers
  - `-pc_mg_levels n` Set the maximum number of levels to use.
  - `-mg_levels_ksp_type type` If `KSPCHEBYSHEV` or `KSPRICHARDSON` is not used, then the Krylov method for the entire multigrid solve has to be a flexible method such as `KSPFGMRES`. Generally, the stronger the Krylov method the faster the convergence, but with more cost per iteration. See `KSPSetType()`.
  - `-mg_levels_ksp_max_it maxit` Sets the number of iterations to run the smoother on each level. Generally, the more iterations , the faster the convergence, but with more cost per multigrid iteration. See `PCMGSetNumberSmooth()`.
  - `-mg_levels_ksp_xxx` Sets options for the `KSP` in the smoother on the levels.
  - `-mg_levels_pc_type type` Sets the smoother to use on each level. See `PCSetType()`. Generally, the stronger the preconditioner the faster the convergence, but with more cost per iteration.
  - `-mg_levels_pc_xxx` Sets options for the `PC` in the smoother on the levels.
  - `-mg_coarse_ksp_type type` Sets the solver `KSPType` to use on the coarsest level.
  - `-mg_coarse_pc_type type` Sets the solver `PCType` to use on the coarsest level.
  - `-pc_gamg_asm_use_agg (true|false)` Use `PCASM` as the smoother on each level with the aggregates defined by the coarsening process are the subdomains. This option automatically switches the smoother on the levels to be `PCASM`.
  - `-mg_levels_pc_asm_overlap n` Use non-zero overlap with `-pc_gamg_asm_use_agg`. See `PCASMSetOverlap()`.
- Control the multigrid algorithm
  - `-pc_mg_type (additive|multiplicative|full|kaskade)` The type of multigrid to use. Usually, multiplicative is the fastest.

– `-pc_mg_cycle_type (v|w)` Use V- or W-cycle with `-pc_mg_type multiplicative`

`PCGAMG` provides unsmoothed aggregation (`-pc_gamg_agg_nsmooths 0`) and smoothed aggregation (`-pc_gamg_agg_nsmooths 1` or `PCGAMGSetNSmooths(pc,1)`). Smoothed aggregation (SA), [VanvekMB96], [VanveekBM01], is recommended for symmetric positive definite systems. Unsmoothed aggregation can be useful for asymmetric problems and problems where the highest eigenestimates 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, used 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 faster 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 as with 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 require knowledge of the number of degrees of freedom per vertex; the default is one (a scalar problem). Vector problems like elasticity should set the block size of the matrix appropriately with `-mat_block_size bs` or `MatSetBlockSize(mat,bs)`. Equations must be ordered in "vertex-major" ordering (e.g., $x_1, y_1, z_1, x_2, y_2, ...$).

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

Coarse grid data model: The GAMG framework provides for reducing the number of active processes on coarse grids to reduce communication costs when there is not enough parallelism to keep relative communication costs down. Most AMG solvers 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 reduce communication costs potentially). However, 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. GAMG will select the number of active processors by fitting the desired number of equations per process (set with `-pc_gamg_process_eq_limit n`) 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 processes 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 moved 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 (e.g., LU or SVD) in a relatively short time. GAMG will stop coarsening when the number of the equation on a grid falls below the threshold given 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 using `PCGAMG` with `-help` to get a full listing of GAMG parameters with short descriptions. The rate of coarsening is critical in AMG performance – too slow 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 -1` and `-pc_gamg_aggressive_coarsening N` 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 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,

and 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.

The parallel MIS algorithms require symmetric weights/matrices. Thus PCGAMG will automatically make the graph symmetric if it is not symmetric. Since this has additional cost, users should indicate the symmetry of the matrices they provide by calling

```
MatSetOption(mat,MAT_SYMMETRIC,PETSC_TRUE (or PETSC_FALSE))
```

or

```
MatSetOption(mat,MAT_STRUCTURALLY_SYMMETRIC,PETSC_TRUE (or PETSC_FALSE)).
```

If they know that the matrix will always have symmetry despite future changes to the matrix (with, for example, MatSetValues()) then they should also call

```
MatSetOption(mat,MAT_SYMMETRY_ETERNAL,PETSC_TRUE (or PETSC_FALSE))
```

or

```
MatSetOption(mat,MAT_STRUCTURAL_SYMMETRY_ETERNAL,PETSC_TRUE (or PETSC_FALSE)).
```

Using this information allows the algorithm to skip unnecessary computations.

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

I am converging slowly, what do I do? AMG methods are sensitive to coarsening rates and methods; for GAMG use `-pc_gamg_threshold x` or PCGAMGSetThreshold() to regulate coarsening rates; higher values decrease the coarsening rate. 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.

Aggressive_coarsening is the second mechanism for increasing the coarsening rate and thereby decreasing the cost of the coarse grids and generally decreasing the solver convergence rate. Use `-pc_gamg_aggressive_coarsening N`, or PCGAMGSetAggressiveLevels(pc,N), to aggressively coarsen the graph on the finest N levels. The default is $N = 1$. There are two options for aggressive coarsening: 1) the default, square graph: use $A^T A$ in the MIS coarsening algorithm and 2) coarsen with MIS-2 (instead of the default of MIS-1). Use `-pc_gamg_aggressive_square_graph false` to use MIS-k coarsening and `-pc_gamg_aggressive_mis_k k` to select the level of MIS other than the default $k = 2$. The square graph approach seems to coarsen slower, which results in larger coarse grids and is more expensive, but generally improves the convergence rate. If the coarse grids are expensive to compute, and use a lot of memory, using MIS-2 is a good alternative (setting MIS-1 effectively turns aggressive coarsening off). Note that MIS-3 is also supported.

One can run with `-info :pc` and grep for `PCGAMG` to get statistics on each level, which can be used to see if you are coarsening at an appropriate rate. With smoothed aggregation, you generally want to coarse at about a rate of 3:1 in each dimension. Coarsening too slowly 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, one can safely decrease the coarsening rate. `-pc_gamg_threshold` -1.0 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, decreasing the coarsening rate (increasing the threshold) should be tried if convergence is slow.

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), the solvers can fail with an indefinite preconditioner message. One can run with `-info` and grep for `PCGAMG` 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`. Bad Eigen estimates often cause indefinite matrix messages. 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 one 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. However, 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 using aggressive coarsening (`-pc_gamg_aggressive_coarsening N`, squares the graph on the finest N levels). Likewise, if your `MatPtAP()` time is short and your convergence If the rate is not ideal, you could decrease the coarsening rate.

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

Using algebraic multigrid as a "standalone" solver is possible but not recommended, as it does not accelerate it with a Krylov method. Use a `KSPType` of `KSPRICHARDSON` (or equivalently `-ksp_type richardson`) to achieve this. Using `KSPPREONLY` will not work since it only applies a single multigrid cycle.

Adaptive Interpolation

Interpolation transfers a function from the coarse space to the fine space. We would like this process to be accurate for the functions resolved by the coarse grid, in particular the approximate solution computed there. By default, we create these matrices using local interpolation of the fine grid dual basis functions in the coarse basis. However, an adaptive procedure can optimize the coefficients of the interpolator to reproduce pairs of coarse/fine functions which should approximate the lowest modes of the generalized eigenproblem

$$Ax = \lambda Mx$$

where A is the system matrix and M is the smoother. Note that for defect-correction MG, the interpolated solution from the coarse space need not be as accurate as the fine solution, for the same reason that updates in iterative refinement can be less accurate. However, in FAS or in the final interpolation step for each level of Full Multigrid, we must have interpolation as accurate as the fine solution since we are moving the entire solution itself.

Injection should accurately transfer the fine solution to the coarse grid. Accuracy here means that the action of a coarse dual function on either should produce approximately the same result. In the structured grid case, this means that we just use the same values on coarse points. This can result in aliasing.

Restriction is intended to transfer the fine residual to the coarse space. Here we use averaging (often the transpose of the interpolation operation) to damp out the fine space contributions. Thus, it is less accurate than injection, but avoids aliasing of the high modes.

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

$$\min_P \|f^F - P f^C\|_2$$

Now we can express P as a matrix by looking at the matrix elements $P_{ij} = \phi_i^F P \phi_j^C$. Then we have

$$\begin{aligned} &\phi_i^F f^F - \phi_i^F P f^C \\ =&f_i^F - \sum_j P_{ij} f_j^C \end{aligned}$$

so that our discrete optimization problem is

$$\min_{P_{ij}} \|f_i^F - \sum_j P_{ij} f_j^C\|_2$$

and we will treat each row of the interpolator as a separate optimization problem. We could allow an arbitrary sparsity pattern, or try to determine adaptively, as is done in sparse approximate inverse preconditioning. However, we know the supports of the basis functions in finite elements, and thus the naive sparsity pattern from local interpolation can be used.

We note here that the BAMG framework of Brannick et al. [BBKL11] does not use fine and coarse functions spaces, but rather a fine point/coarse point division which we will not employ here. Our general PETSc routine should work for both since the input would be the checking set (fine basis coefficients or fine space points) and the approximation set (coarse basis coefficients in the support or coarse points in the sparsity pattern).

We can easily solve the above problem using QR factorization. However, there are many smooth functions from the coarse space that we want interpolated accurately, and a single f would not constrain the values

P_{ij}‘ well. Therefore, we will use several functions $\{f_k\}$ in our minimization,

$$\begin{aligned}
&\min_{P_{ij}} \sum_k w_k \| f_i^{F,k} - \sum_j P_{ij} f_j^{C,k} \|_2 \\
=&\min_{P_{ij}} \sum_k \| \sqrt{w_k} f_i^{F,k} - \sqrt{w_k} \sum_j P_{ij} f_j^{C,k} \|_2 \\
=&\min_{P_{ij}} \| W^{1/2} \mathbf{f}_i^F - W^{1/2} \mathbf{f}^C p_i \|_2
\end{aligned}$$

where

$$\begin{aligned}
W &= \begin{pmatrix} w_0 & & \\ & \ddots & \\ & & w_K \end{pmatrix} \\
\mathbf{f}_i^F &= \begin{pmatrix} f_i^{F,0} \\ \vdots \\ f_i^{F,K} \end{pmatrix} \\
\mathbf{f}^C &= \begin{pmatrix} f_0^{C,0} & \cdots & f_n^{C,0} \\ \vdots & \ddots & \vdots \\ f_0^{C,K} & \cdots & f_n^{C,K} \end{pmatrix} \\
p_i &= \begin{pmatrix} P_{i0} \\ \vdots \\ P_{in} \end{pmatrix}
\end{aligned}$$

or alternatively

$$\begin{aligned}
[W]_{kk} &= w_k \\
[f_i^F]_k &= f_i^{F,k} \\
[f^C]_{kj} &= f_j^{C,k} \\
[p_i]_j &= P_{ij}
\end{aligned}$$

We thus have a standard least-squares problem

$$\min_{P_{ij}} \| b - Ax \|_2$$

where

$$\begin{aligned}
A &= W^{1/2} f^C \\
b &= W^{1/2} f_i^F \\
x &= p_i
\end{aligned}$$

which can be solved using LAPACK.

We will typically perform this optimization on a multigrid level l when the change in eigenvalue from level $l+1$ is relatively large, meaning

$$\frac{|\lambda_l - \lambda_{l+1}|}{|\lambda_l|}.$$

This indicates that the generalized eigenvector associated with that eigenvalue was not adequately represented by P_{l+1}^l‘, and the interpolator should be recomputed.

Balancing Domain Decomposition by Constraints

PETSc provides the Balancing Domain Decomposition by Constraints (`PCBDDC`) method for preconditioning parallel finite element problems stored in unassembled format (see `MATIS`). `PCBDDC` 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 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 matrix used to compute the preconditioner 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 `PCBDDC` 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()`, `PCISSetSubdomainDiagonalScal-`

ing() or PCISSetUseStiffnessScaling(). However, the convergence properties of the PCBDDC 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 PCBDDC 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 PCBDDC 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. That is, it allows users to write or wrap their own custom preconditioners as a PC and use it with KSP, etc.

To provide a custom preconditioner application, use

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

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

```
PCShellSetContext(PC pc, PetscCtx ctx);
```

to set this data structure and

```
PCShellGetContext(PC pc, PetscCtxRt ctx);
```

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

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

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

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

Combining Preconditioners

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

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

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

Any number of preconditioners may added in this way.

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

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

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

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

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

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

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

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

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

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

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

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

One can also change the operator that is used to construct a particular PC in the composite PC calling PCSetOperators() on the obtained PC. PCFIELDSPLIT, *Solving Block Matrices with PCFIELDSPLIT*, provides an alternative approach to defining composite preconditioners with a variety of pre-defined compositions.

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 `KSPSolve()` linear solver. This is achieved with the `PCKSP` type.

```
PCSetType(PC pc,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[1]. In the `PCMG` framework, the user is required to provide the coarse grid solver, smoothers, restriction and interpolation operators, and code to calculate residuals. The `PCMG` 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

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

If the number of levels is not set with `PCMGSetLevels()` or `-pc_mg_levels` and no `DM` has been attached to the `PCMG` with `KSPSetDM()` (or `SNESSetDM()` or `TSSetDM()`), then `PCMG` uses only one level! This is different from the algebraic multigrid methods such as `PCGAMG`, `PCML`, and `PCHYPRE` which internally determine the number of levels to use.

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

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

indicates which form of multigrid to apply [SBjorstadG96].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

sets SOR as the smoother to use on level 1.

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

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

Use

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

and

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

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

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

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

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

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

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

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

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

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

If `KSPSetDM()` (or `SNESSetDM()` or `TSSetDM()`) has been called, then `PCMG` will use geometric information from the `DM` to construct the multigrid hierarchy automatically. In this case one does not need to call the various `PCMGSet` routines listed above.

2.4.5 Solving Block Matrices with PCFIELDSPLIT

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 a black box. In this section, we use the common linear algebra definition of block matrices, where matrices are divided into 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, such as 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. The matrices on a parallel computer are not explicitly stored this way. Instead, each process will own some rows of A_{0*}, A_{1*} etc. On a process, the blocks may be stored in one block followed by another

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

or interlaced, for example, with four blocks

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

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

The PETSc `PCFIELDSPLIT` preconditioner implements the "block" solvers in PETSc, [EHS+08]. There are three ways to provide the information that defines the blocks. If the matrices are stored as interlaced then `PCFieldSplitSetFields()` can be called repeatedly to indicate which fields belong to each block. More generally `PCFieldSplitSetIS()` can be used to indicate exactly which rows/columns of the matrix belong to a particular block (field). You can provide names for each block with these routines; if you do not, they are numbered from 0. With these two approaches, the blocks may overlap (though they generally will not overlap). If only one block is defined, then the complement of the matrices is used to define the other block. Finally, the option `-pc_fieldsplit_detect_saddle_point` causes two diagonal blocks to be found, one associated with all rows/columns that have zeros on the diagonals and the rest.

Important parameters for PCFIELDSPLIT

- Control the fields used
  - `-pc_fieldsplit_detect_saddle_point (true|false)` Generate two fields, the first consists of all rows with a nonzero on the diagonal, and the second will be all rows with zero on the diagonal. See `PCFieldSplitSetDetectSaddlePoint()`.
  - `-pc_fieldsplit_dm_splits (true|false)` Use the `DM` attached to the preconditioner to determine the fields. See `PCFieldSplitSetDMSplits()` and `DMCreateFieldDecomposition()`.
  - `-pc_fieldsplit_%d_fields f1,f2,...` Use f1, f2, … to define field `d`. The `fn` are in the range of 0, …, bs-1 where bs is the block size of the matrix or set with `PCFieldSplitSetBlockSize()`. See `PCFieldSplitSetFields()`.
    - `-pc_fieldsplit_default (true|false)` Automatically add any fields needed that have not been supplied explicitly by `-pc_fieldsplit_%d_fields`.
  - `DMFieldsplitSetIS()` Provide the `IS` that defines a particular field.
- Control the type of the block preconditioner
  - `-pc_fieldsplit_type (additive|multiplicative|symmetric_multiplicative|schur|gkb)` The order in which the field solves are applied. For symmetric problems where `KSPCG` is used `symmetric_multiplicative` must be used instead of `multiplicative`. `additive` is the least expensive to apply but provides the worst convergence. `schur` requires either a good preconditioner for the Schur complement or a naturally well-conditioned Schur complement, but when it works well can be extremely effective. See `PCFieldSplitSetType()`. `gkb` is for symmetric saddle-point problems (the lower-right the block is zero).
  - `-pc_fieldsplit_diag_use_amat (true|false)` Use the first matrix that is passed to `KSPSetJacobian()` to construct the block-diagonal sub-matrices used in the algorithms, by default, the second matrix is used.

  - Options for Schur preconditioner: `-pc_fieldsplit_type schur`
    * `-pc_fieldsplit_schur_fact_type (diag|lower|upper|full)` See `PCFieldSplitSetSchurFactType()`. `full` reduces the iterations but each iteration requires additional field solves.
    * `-pc_fieldsplit_schur_precondition (self|selfp|user|a11|full:user)` How the Schur complement is preconditioned. See `PCFieldSplitSetSchurPre()`.
      · `-fieldsplit_1_mat_schur_complement_ainv_type (diag|lump)` Use the lumped diagonal of A_{00} when `-pc_fieldsplit_schur_precondition selfp` is used.
    * `-pc_fieldsplit_schur_scale scale` Controls the sign flip of S for `-pc_fieldsplit_schur_fact_type diag`. See `PCFieldSplitSetSchurScale()`
    * `fieldsplit_1_xxx` controls the solver for the Schur complement system. If a `DM` provided the fields, use the second field name set in the `DM` instead of 1.
      · `-fieldsplit_1_pc_type lsc -fieldsplit_1_lsc_pc_xxx` use the least squares commutators [EHS+06] [SEKW01] preconditioner for the Schur complement with any preconditioner for the least-squares matrix, see `PCLSC`. If a `DM` provided the fields, use the second field name set in the `DM` instead of 1.
    * `-fieldsplit_upper_xxx` Set options for the solver in the upper solver when `-pc_fieldsplit_schur_fact_type upper` or `full` is used. Defaults to using the solver as provided with `-fieldsplit_0_xxx`.
    * `-fieldsplit_1_inner_xxx` Set the options for the solver inside the application of the Schur complement; defaults to using the solver as provided with `-fieldsplit_0_xxx`. If a `DM` provides the fields use the name of the second field name set in the `DM` instead of 1.
  - Options for GKB preconditioner: `-pc_fieldsplit_type gkb`
    * `-pc_fieldsplit_gkb_tol tol` See `PCFieldSplitSetGKBTol()`.
    * `-pc_fieldsplit_gkb_delay delay` See `PCFieldSplitSetGKBDelay()`.
    * `-pc_fieldsplit_gkb_nu nu` See `PCFieldSplitSetGKBNu()`.
    * `-pc_fieldsplit_gkb_maxit maxit` See `PCFieldSplitSetGKBMaxit()`.
    * `-pc_fieldsplit_gkb_monitor (true|false)` Monitor the convergence of the inner solver.
- Options for additive and multiplication field solvers:
  - `-fieldsplit_%d_xxx` Set options for the solver for field number `d`. For example, `-fieldsplit_0_pc_type jacobi`. When the fields are obtained from a `DM` use the field name instead of `d`.

For simplicity, we restrict our matrices to two-by-two blocks in the rest of the section. 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

$$\begin{pmatrix} Ap_{00} & Ap_{01} \\ Ap_{10} & Ap_{11} \end{pmatrix}.$$

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

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

$$\begin{pmatrix} \text{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & \text{ksp}(A_{11}, Ap_{11}) \end{pmatrix}$$

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[2] as

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

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 `-pc_fieldsplit_type (additive|multiplicative|symmetric_multiplicative)` or the function `PCFieldSplitSetType()`. The option prefixes for the internal KSPs are given by `-fieldsplit_name_`.

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

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

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

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

$$\begin{pmatrix} \text{ksp}(A_{00}, Ap_{00}) & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & -A_{01} \\ 0 & I \end{pmatrix}$$

[2] 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.

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

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

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

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

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

The effectiveness of the Schur complement preconditioner depends on the availability of a good preconditioner $\hat{S}p$ for the Schur complement matrix. In general, you are responsible for supplying $\hat{S}p$ via `PCFieldSplitSetSchurPre(pc,PC_FIELDSPLIT_SCHUR_PRE_USER,Sp)`. Without a good problem-specific $\hat{S}p$, you can use some built-in options.

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

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

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

The problem with the last approach is that $\hat{S}$ is used in the 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, to ensure the sparsity of $\hat{S}p$ as much as possible. Specifically, using `-pc_fieldsplit_schur_precondition selfp` will assemble $\hat{S}p = A_{11} - A_{10}\text{inv}(A_{00})A_{01}$.

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 $A_{10}A_{01}$, users can provide their own matrix. This is done by attaching the matrix/matrices to the Sp matrix they provide with

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

2.4.6 Solving Singular Systems

Sometimes one is required to solver singular linear systems. In this case, the system matrix has a nontrivial null space. For example, the discretization of the Laplacian operator with Neumann boundary conditions has a null space of the constant functions. PETSc has tools to help solve these systems. This approach is only guaranteed to work for left preconditioning (see `KSPSetPCSide()`); for example it may not work in some situations with `KSPFGMRES`.

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

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

Here, `dim` is the number of vectors in `basis` and `hasconstants` indicates if the null space contains the constant functions. If the null space contains the constant functions you do not need to include it in the `basis` vectors you provide, nor in the count `dim`.

One then tells the `KSP` object you are using what the null space is with the call

```
MatSetNullSpace(Mat Amat,MatNullSpace nsp);
```

The `Amat` should be the *first* matrix argument used with `KSPSetOperators()`, `SNESSetJacobian()`, or `TSSetIJacobian()`. The PETSc solvers will now handle the null space during the solution process.

If the right-hand side of linear system is not in the range of `Amat`, that is it is not orthogonal to the null space of `Amat` transpose, then the residual norm of the Krylov iteration will not converge to zero; it will converge to a non-zero value while the solution is converging to the least squares solution of the linear system. One can, if one desires, apply `MatNullSpaceRemove()` with the null space of `Amat` transpose to the right-hand side before calling `KSPSolve()`. Then the residual norm will converge to zero.

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.

If the matrix is non-symmetric and you wish to solve the transposed linear system you must provide the null space of the transposed matrix with `MatSetTransposeNullSpace()`.

2.4.7 Using External Linear Solvers

PETSc interfaces to several external linear solvers (also see `acknowledgements`). 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` (or equivalently `-ksp_type none`) `-pc_type type -pc_factor_mat_solver_type packagename`. For eg: `-pc_factor_mat_solver_type packagename`. For e.g.: `-ksp_type preonly -pc_type lu -pc_factor_mat_solver_type superlu_dist`.

Table 2.7: Options for External Solvers

MatType	PCType	MatSolverType	Package
seqaij	lu	MATSOLVERESSL	essl
seqaij	lu	MATSOLVERLUSOL	lusol
seqaij	lu	MATSOLVERMATLAB	matlab
aij	lu	MATSOLVERMUMPS	mumps
aij	cholesky	•	•
sbaij	cholesky	•	•
seqaij	lu	MATSOLVERSUPERLU	superlu
aij	lu	MATSOLVERSU-PERLU_DIST	superlu_dist
seqaij	lu	MATSOLVERUMFPACK	umfpack
seqaij	cholesky	MATSOLVERCHOLMOD	cholmod
seqaij	lu	MATSOLVERKLU	klu
dense	lu	MATSOLVERELEMEN-TAL	elemental
dense	cholesky	•	•
seqaij	lu	MAT-SOLVERMKL_PARDISO	mkl_pardiso
aij	lu	MAT-SOLVERMKL_CPARDISO	mkl_cpardiso
aij	lu	MATSOLVERPASTIX	pastix
aij	cholesky	MATSOLVERBAS	bas
aijcusparse	lu	MATSOLVERCUSPARSE	cusparse
aijcusparse	cholesky	•	•
aij	lu, cholesky	MATSOLVERPETSC	petsc
baij	•	•	•
aijcrl	•	•	•
aijperm	•	•	•
seqdense	•	•	•
aij	•	•	•
baij	•	•	•
aijcrl	•	•	•
aijperm	•	•	•
seqdense	•	•	•

The default and available input options for each external software can be found by specifying `-help` at runtime.

As an alternative to using runtime flags to employ these external packages, procedural calls are provided for some packages. For example, the following procedural calls are equivalent to runtime options `-ksp_type preonly` (or equivalently `-ksp_type none`) `-pc_type lu -pc_factor_mat_solver_type mumps -mat_mumps_icntl_7 3`:

```
KSPSetType(ksp, KSPPREONLY); // (or equivalently KSPSetType(ksp, KSPNONE))
KSPGetPC(ksp, &pc);
PCSetType(pc, PCLU);
PCFactorSetMatSolverType(pc, MATSOLVERMUMPS);
PCFactorSetUpMatSolverType(pc);
PCFactorGetMatrix(pc, &F);
icntl=7; ival = 3;
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: `MATSEQAIJ`, `MATMPIAIJ`, 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.

2.4.8 Using PETSc's MPI parallel linear solvers from a non-MPI program

Using PETSc's MPI linear solver server it is possible to use multiple MPI processes to solve a a linear system when the application code, including the matrix generation, is run on a single MPI process (with or without OpenMP). The application code must be built with MPI and must call `PetscInitialize()` at the very beginning of the program and end with `PetscFinalize()`. The application code may utilize OpenMP. The code may create multiple matrices and `KSP` objects and call `KSPSolve()`, similarly the code may utilize the `SNES` nonlinear solvers, the `TS` ODE integrators, and the `Tao` optimization algorithms which use `KSP`.

The program must then be launched using the standard approaches for launching MPI programs with the additional PETSc option `-mpi_linear_solver_server`. The linear solves are controlled via the options database in the usual manner (using any options prefix you may have provided via `KSPSetOptionsPrefix()`, for example `-ksp_type cg -ksp_monitor -pc_type bjacobi -ksp_view`. The solver options cannot be set via the functional interface, for example `KSPSetType()` etc.

The option `-mpi_linear_solver_server_view` will print a summary of all the systems solved by the MPI linear solver server when the program completes. By default the linear solver server will only parallelize the linear solve to the extent that it believes is appropriate to obtain speedup for the parallel solve, for example, if the matrix has 1,000 rows and columns the solution will not be parallelized by default. One can use the option `-mpi_linear_solver_server_minimum_count_per_rank 5000` to cause the linear solver server to allow as few as 5,000 unknowns per MPI process in the parallel solve.

See PCMPI, PCMPIServerBegin(), and PCMPIServerEnd() for more details on the solvers.

For help when anything goes wrong with the MPI linear solver server see PCMPIServerBegin().

Amdahl's law makes clear that parallelizing only a portion of a numerical code can only provide a limited improvement in the computation time; thus it is crucial to understand what phases of a computation must be parallelized (via MPI, OpenMP, or some other model) to ensure a useful increase in performance. One of the crucial phases is likely the generation of the matrix entries; the use of MatSetPreallocationCOO() and MatSetValuesCOO() in an OpenMP code allows parallelizing the generation of the matrix.

See *Application with the MPI linear solver server* for a study of the use of PCMPI on a specific PETSc application.

References

2.5 SNES: Nonlinear Solvers

The solution of large-scale nonlinear problems pervades many facets of computational science and demands robust and flexible solution strategies. The SNES library of PETSc provides a powerful suite of data-structure-neutral numerical routines for such problems. Built on top of the linear solvers and data structures discussed in preceding chapters, SNES enables the user to easily customize the nonlinear solvers according to the application at hand. Also, the SNES interface is *identical* for the uniprocess and parallel cases; the only difference in the parallel version is that each process typically forms only its local contribution to various matrices and vectors.

The SNES class includes methods for solving systems of nonlinear equations of the form

$$\mathbf{F}(\mathbf{x}) = 0, \tag{2.3}$$

where $\mathbf{F} : \Re^n \to \Re^n$. Newton-like methods provide the core of the package, including both line search and trust region techniques. A suite of nonlinear Krylov methods and methods based upon problem decomposition are also included. The solvers are discussed further in *The Nonlinear Solvers*. Following the PETSc design philosophy, the interfaces to the various solvers are all virtually identical. In addition, the SNES software is completely flexible, so that the user can at runtime change any facet of the solution process.

PETSc's default method for solving the nonlinear equation is Newton's method with line search, SNESNEWTONLS. The general form of the n-dimensional Newton's method for solving (2.3) is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k), \quad k = 0, 1, \ldots, \tag{2.4}$$

where $\mathbf{x}_0$ is an initial approximation to the solution and $\mathbf{J}(\mathbf{x}_k) = \mathbf{F}'(\mathbf{x}_k)$, the Jacobian, is nonsingular at each iteration. In practice, the Newton iteration (2.4) is implemented by the following two steps:

$$\begin{aligned} &1.\text{(Approximately) solve} && \mathbf{J}(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{F}(\mathbf{x}_k). \\ &2.\text{Update} && \mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \Delta\mathbf{x}_k. \end{aligned}$$

Other defect-correction algorithms can be implemented by using different choices for $J(\mathbf{x}_k)$.

2.5.1 Basic SNES Usage

In the simplest usage of the nonlinear solvers, the user must merely provide a C, C++, Fortran, or Python routine to evaluate the nonlinear function (2.3). The corresponding Jacobian matrix can be approximated with finite differences. For codes that are typically more efficient and accurate, the user can provide a routine to compute the Jacobian; details regarding these application-provided routines are discussed below. To provide an overview of the use of the nonlinear solvers, browse the concrete example in *ex1.c* or skip ahead to the discussion.

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

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

/*
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petscsys.h       - base PETSc routines   petscvec.h - vectors
     petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace methods
     petscviewer.h - viewers               petscpc.h  - preconditioners
     petscksp.h   - linear solvers
*/
/*F
This examples solves either
\begin{equation}
  F\genfrac{(}{)}{0pt}{}{x_0}{x_1} = \genfrac{(}{)}{0pt}{}{x^2_0 + x_0 x_1 - 3}{x_0 x_
↪1 + x^2_1 - 6}
\end{equation}
or if the {\tt -hard} options is given
\begin{equation}
  F\genfrac{(}{)}{0pt}{}{x_0}{x_1} = \genfrac{(}{)}{0pt}{}{\sin(3 x_0) + x_0}{x_1}
\end{equation}
F*/
#include <petscsnes.h>

/*
   User-defined routines
*/
extern PetscErrorCode FormJacobian1(SNES, Vec, Mat, Mat, void *);
extern PetscErrorCode FormFunction1(SNES, Vec, Vec, void *);
extern PetscErrorCode FormJacobian2(SNES, Vec, Mat, Mat, void *);
extern PetscErrorCode FormFunction2(SNES, Vec, Vec, void *);

int main(int argc, char **argv)
{
  SNES        snes; /* nonlinear solver context */
  KSP         ksp;  /* linear solver context */
  PC          pc;   /* preconditioner context */
  Vec         x, r; /* solution, residual vectors */
  Mat         J;    /* Jacobian matrix */
  PetscMPIInt size;
  PetscScalar pfive = .5, *xx;
  PetscBool   flg;

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));
  PetscCallMPI(MPI_Comm_size(PETSC_COMM_WORLD, &size));
```

(continues on next page)

(continued from previous page)

```
  PetscCheck(size == 1, PETSC_COMM_WORLD, PETSC_ERR_WRONG_MPI_SIZE, "Example is only␣
↪for sequential runs");

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  PetscCall(SNESCreate(PETSC_COMM_WORLD, &snes));
  PetscCall(SNESSetType(snes, SNESNEWTONLS));
  PetscCall(SNESSetOptionsPrefix(snes, "mysolver_"));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create matrix and vector data structures; set corresponding routines
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Create vectors for solution and nonlinear function
  */
  PetscCall(VecCreate(PETSC_COMM_WORLD, &x));
  PetscCall(VecSetSizes(x, PETSC_DECIDE, 2));
  PetscCall(VecSetFromOptions(x));
  PetscCall(VecDuplicate(x, &r));

  /*
     Create Jacobian matrix data structure
  */
  PetscCall(MatCreate(PETSC_COMM_WORLD, &J));
  PetscCall(MatSetSizes(J, PETSC_DECIDE, PETSC_DECIDE, 2, 2));
  PetscCall(MatSetFromOptions(J));
  PetscCall(MatSetUp(J));

  PetscCall(PetscOptionsHasName(NULL, NULL, "-hard", &flg));
  if (!flg) {
    /*
     Set function evaluation routine and vector.
    */
    PetscCall(SNESSetFunction(snes, r, FormFunction1, NULL));

    /*
     Set Jacobian matrix data structure and Jacobian evaluation routine
    */
    PetscCall(SNESSetJacobian(snes, J, J, FormJacobian1, NULL));
  } else {
    PetscCall(SNESSetFunction(snes, r, FormFunction2, NULL));
    PetscCall(SNESSetJacobian(snes, J, J, FormJacobian2, NULL));
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     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.
  */
  PetscCall(SNESGetKSP(snes, &ksp));
  PetscCall(KSPGetPC(ksp, &pc));
  PetscCall(PCSetType(pc, PCNONE));
```

(continues on next page)

(continued from previous page)

```
  PetscCall(KSPSetTolerances(ksp, 1.e-4, PETSC_CURRENT, PETSC_CURRENT, 20));

  /*
     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.
  */
  PetscCall(SNESSetFromOptions(snes));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Evaluate initial guess; then solve nonlinear system
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  if (!flg) PetscCall(VecSet(x, pfive));
  else {
    PetscCall(VecGetArray(x, &xx));
    xx[0] = 2.0;
    xx[1] = 3.0;
    PetscCall(VecRestoreArray(x, &xx));
  }
  /*
     Note: The user should initialize the vector, x, with the initial guess
     for the nonlinear solver prior to calling SNESSolve().  In particular,
     to employ an initial guess of zero, the user should explicitly set
     this vector to zero by calling VecSet().
  */

  PetscCall(SNESSolve(snes, NULL, x));
  if (flg) {
    Vec f;
    PetscCall(VecView(x, PETSC_VIEWER_STDOUT_WORLD));
    PetscCall(SNESGetFunction(snes, &f, 0, 0));
    PetscCall(VecView(r, PETSC_VIEWER_STDOUT_WORLD));
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  PetscCall(VecDestroy(&x));
  PetscCall(VecDestroy(&r));
  PetscCall(MatDestroy(&J));
  PetscCall(SNESDestroy(&snes));
  PetscCall(PetscFinalize());
  return 0;
}
/* ------------------------------------------------------------------- */
/*
   FormFunction1 - Evaluates nonlinear function, F(x).

   Input Parameters:
.  snes - the SNES context
.  x    - input vector
.  ctx  - optional user-defined context
```

(continues on next page)

(continued from previous page)

```
   Output Parameter:
.  f - function vector
 */
PetscErrorCode FormFunction1(SNES snes, Vec x, Vec f, PetscCtx ctx)
{
  const PetscScalar *xx;
  PetscScalar       *ff;

  PetscFunctionBeginUser;
  /*
   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.
   */
  PetscCall(VecGetArrayRead(x, &xx));
  PetscCall(VecGetArray(f, &ff));

  /* Compute function */
  ff[0] = xx[0] * xx[0] + xx[0] * xx[1] - 3.0;
  ff[1] = xx[0] * xx[1] + xx[1] * xx[1] - 6.0;

  /* Restore vectors */
  PetscCall(VecRestoreArrayRead(x, &xx));
  PetscCall(VecRestoreArray(f, &ff));
  PetscFunctionReturn(PETSC_SUCCESS);
}
/* ------------------------------------------------------------------- */
/*
   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 matrix used to construct the preconditioner

*/
PetscErrorCode FormJacobian1(SNES snes, Vec x, Mat jac, Mat B, void *dummy)
{
  const PetscScalar *xx;
  PetscScalar        A[4];
  PetscInt           idx[2] = {0, 1};

  PetscFunctionBeginUser;
  /*
     Get pointer to vector data
  */
  PetscCall(VecGetArrayRead(x, &xx));

  /*
```

(continues on next page)

(continued from previous page)

```
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
        the matrix at once.
  */
  A[0] = 2.0 * xx[0] + xx[1];
  A[1] = xx[0];
  A[2] = xx[1];
  A[3] = xx[0] + 2.0 * xx[1];
  PetscCall(MatSetValues(B, 2, idx, 2, idx, A, INSERT_VALUES));

  /*
     Restore vector
  */
  PetscCall(VecRestoreArrayRead(x, &xx));

  /*
     Assemble matrix
  */
  PetscCall(MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY));
  PetscCall(MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY));
  if (jac != B) {
    PetscCall(MatAssemblyBegin(jac, MAT_FINAL_ASSEMBLY));
    PetscCall(MatAssemblyEnd(jac, MAT_FINAL_ASSEMBLY));
  }
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
PetscErrorCode FormFunction2(SNES snes, Vec x, Vec f, void *dummy)
{
  const PetscScalar *xx;
  PetscScalar       *ff;

  PetscFunctionBeginUser;
  /*
     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.
  */
  PetscCall(VecGetArrayRead(x, &xx));
  PetscCall(VecGetArray(f, &ff));

  /*
     Compute function
  */
  ff[0] = PetscSinScalar(3.0 * xx[0]) + xx[0];
  ff[1] = xx[1];

  /*
     Restore vectors
  */
  PetscCall(VecRestoreArrayRead(x, &xx));
  PetscCall(VecRestoreArray(f, &ff));
  PetscFunctionReturn(PETSC_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
}
/* ------------------------------------------------------------------- */
PetscErrorCode FormJacobian2(SNES snes, Vec x, Mat jac, Mat B, void *dummy)
{
  const PetscScalar *xx;
  PetscScalar        A[4];
  PetscInt           idx[2] = {0, 1};

  PetscFunctionBeginUser;
  /*
     Get pointer to vector data
  */
  PetscCall(VecGetArrayRead(x, &xx));

  /*
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
        the matrix at once.
  */
  A[0] = 3.0 * PetscCosScalar(3.0 * xx[0]) + 1.0;
  A[1] = 0.0;
  A[2] = 0.0;
  A[3] = 1.0;
  PetscCall(MatSetValues(B, 2, idx, 2, idx, A, INSERT_VALUES));

  /*
     Restore vector
  */
  PetscCall(VecRestoreArrayRead(x, &xx));

  /*
     Assemble matrix
  */
  PetscCall(MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY));
  PetscCall(MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY));
  if (jac != B) {
    PetscCall(MatAssemblyBegin(jac, MAT_FINAL_ASSEMBLY));
    PetscCall(MatAssemblyEnd(jac, MAT_FINAL_ASSEMBLY));
  }
  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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, *possibly*, 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 type`, where details regarding the available methods are presented in *The Nonlinear Solvers*. The application code can take complete control of the linear and nonlinear techniques

used in the Newton-like method by calling

```
SNESSetFromOptions(snes);
```

This routine provides an interface to the PETSc options database, so that at runtime the user can select a particular nonlinear solver, set various parameters and customized routines (e.g., specialized line search variants), prescribe the convergence tolerance, and set monitoring routines. With this routine the user can also control all linear solver options in the KSP, and PC modules, as discussed in *KSP: Linear System Solvers*.

After having set these routines and options, the user solves the problem by calling

```
SNESSolve(SNES snes, Vec b, Vec x);
```

where x should be initialized to the initial guess before calling and contains the solution on return. In particular, to employ an initial guess of zero, the user should explicitly set this vector to zero by calling VecZeroEntries(x). Finally, after solving the nonlinear system (or several systems), the user should destroy the SNES context with

```
SNESDestroy(SNES *snes);
```

Nonlinear Function Evaluation

When solving a system of nonlinear equations, the user must provide a a residual function (2.3), which is set using

```
SNESSetFunction(SNES snes, Vec f, PetscErrorCode (*FormFunction)(SNES snes, Vec x,␣
↪Vec f, PetscCtx ctx), PetscCtx 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 an application context can be a PETSc object or a derived type. SNES Tutorial ex5 and SNES Tutorial ex5f90 give examples of user-defined application contexts in C and Fortran, respectively.

Jacobian Evaluation

The user may also specify a routine to form some approximation of the Jacobian matrix, A, at the current iterate, x, as is typically done with

```
SNESSetJacobian(SNES snes, Mat Amat, Mat Pmat, PetscErrorCode (*FormJacobian)(SNES␣
↪snes, Vec x, Mat A, Mat B, PetscCtx ctx), PetscCtx 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. The FormJacobian() callback is only invoked if the solver requires it, always *after* FormFunction() has been called at the current iterate.

Note that the SNES solvers are all data-structure neutral, so the full range of PETSc matrix formats (including "matrix-free" methods) can be used. *Matrices* discusses information regarding available matrix formats and options, while *Matrix-Free Methods* focuses on matrix-free methods in SNES. We briefly touch on a few details of matrix usage that are particularly important for efficient use of the nonlinear solvers.

A common usage paradigm is to assemble the problem Jacobian in the preconditioner storage `B`, rather than `A`. In the case where they are identical, as in many simulations, this makes no difference. However, it allows us to check the analytic Jacobian we construct in `FormJacobian()` by passing the `-snes_mf_operator` flag. This causes PETSc to approximate the Jacobian using finite differencing of the function evaluation (discussed in *Finite Difference Jacobian Approximations*), and the analytic Jacobian becomes merely the preconditioner. Even if the analytic Jacobian is incorrect, it is likely that the finite difference approximation will converge, and thus this is an excellent method to verify the analytic Jacobian. Moreover, if the analytic Jacobian is incomplete (some terms are missing or approximate), `-snes_mf_operator` may be used to obtain the exact solution, where the Jacobian approximation has been transferred to the preconditioner.

One such approximate Jacobian comes from "Picard linearization", use `SNESSetPicard()`, which writes the nonlinear system as

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b} = 0$$

where $\mathbf{A}(\mathbf{x})$ usually contains the lower-derivative parts of the equation. For example, the nonlinear diffusion problem

$$-\nabla \cdot (\kappa(u)\nabla u) = 0$$

would be linearized as

$$A(u)v \simeq -\nabla \cdot (\kappa(u)\nabla v).$$

Usually this linearization is simpler to implement than Newton and the linear problems are somewhat easier to solve. In addition to using `-snes_mf_operator` with this approximation to the Jacobian, the Picard iterative procedure can be performed by defining $\mathbf{J}(\mathbf{x})$ to be $\mathbf{A}(\mathbf{x})$. Sometimes this iteration exhibits better global convergence than Newton linearization.

During successive calls to `FormJacobian()`, the user can either insert new matrix contexts or reuse old ones, depending on the application requirements. For many sparse matrix formats, reusing the old space (and merely changing the matrix elements) is more efficient; however, if the matrix nonzero 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.

Sometimes a nonlinear solver may produce a step that is not within the domain of a given function, for example one with a negative pressure. When this occurs one can call `SNESSetFunctionDomainError()` or `SNESSetJacobianDomainError()` to indicate to `SNES` the step is not valid. One must also use `SNESGetConvergedReason()` and check the reason to confirm if the solver succeeded. See *Variational Inequalities* for how to provide `SNES` with bounds on the variables to solve (differential) variational inequalities and how to control properties of the line step computed.

2.5.2 Function Domain Errors and infinity or NaN

Occasionally nonlinear solvers will propose solutions u, where the function value (or the objective function set with `SNESSetObjective()`) contains infinity or NaN. This can be due to bugs in the application code or because the proposed solution is not in the domain of the function. The application function can call `SNESSetFunctionDomainError()` or `SNESSetObjectiveDomainError()` to indicate u is not in the function's domain.

Some `SNESSolve()` implementations (and related `SNESLineSearchApply()` routines) attempt to recover from the infinity or NaN; generally by shrinking the step size. If they are unable to recover the `SNESConvergedReason` returned will be `SNES_DIVERGED_FUNCTION_DOMAIN`, `SNES_DIVERGED_OJECTIVE_DOMAIN`, `SNES_DIVERGED_FUNCTION_NANORINF`, or `SNES_DIVERGED_OJECTIVE_NANORINF`.

2.5.3 The Nonlinear Solvers

As summarized in Table *PETSc Nonlinear Solvers*, SNES includes several Newton-like nonlinear solvers based on line search techniques and trust region methods. Also provided are several nonlinear Krylov methods, as well as nonlinear methods involving decompositions of the problem.

Each solver may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. A complete list can be found by consulting the manual pages or by running a program with the `-help` option; we discuss just a few in the sections below.

Table 2.8: PETSc Nonlinear Solvers

Method	SNESType	Options Name	Default Line Search
Line Search Newton	SNESNEWTONLS	newtonls	SNESLINESEARCHBT
Trust region Newton	SNESNEWTONTR	newtontr	—
Newton with Arc Length Continuation	SNESNEWTONAL	newtonal	—
Nonlinear Richardson	SNESNRICHARDSON	nrichardson	SNESLINESEARCHSECANT
Nonlinear CG	SNESNCG	ncg	SNESLINESEARCHCP
Nonlinear GMRES	SNESNGMRES	ngmres	SNESLINESEARCHSECANT
Quasi-Newton	SNESQN	qn	see *PETSc quasi-Newton solvers*
Full Approximation Scheme	SNESFAS	fas	—
Nonlinear ASM	SNESNASM	nasm	–
ASPIN	SNESASPIN	aspin	SNESLINESEARCHBT
Nonlinear Gauss-Seidel	SNESNGS	ngs	–
Anderson Mixing	SNESANDERSON	anderson	–
Newton with constraints (1)	SNESVINEWTONRSLS	vinewtonrsls	SNESLINESEARCHBT
Newton with constraints (2)	SNESVINEWTONSSLS	vinewtonssls	SNESLINESEARCHBT
Multi-stage Smoothers	SNESMS	ms	–
Composite	SNESCOMPOSITE	composite	–
Linear solve only	SNESKSPONLY	ksponly	–
Python Shell	SNESPYTHON	python	–
Shell (user-defined)	SNESSHELL	shell	–

Line Search Newton

The method SNESNEWTONLS (`-snes_type newtonls`) provides a line search Newton method for solving systems of nonlinear equations. By default, this technique employs cubic backtracking [DennisJrS83]. Alternative line search techniques are listed in Table *PETSc Line Search Methods*.

Table 2.9: PETSc Line Search Methods

Line Search	SNESLineSearchType	Options Name
Backtracking	SNESLINESEARCHBT	bt
(damped) step	SNESLINESEARCHBASIC	basic
identical to above	SNESLINESEARCHNONE	none
Secant method	SNESLINESEARCHSECANT	secant
Critical point	SNESLINESEARCHCP	cp
Error-oriented	SNESLINESEARCHNLEQERR	nleqerr
Bisection	SNESLINESEARCHBISECTION	bisection
Shell	SNESLINESEARCHSHELL	shell

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

```
SNESGetLineSearch(SNES snes, SNESLineSearch *ls);.
```

There are several default options for the line searches. The order of polynomial approximation may be set with -snes_linesearch_order or

```
SNESLineSearchSetOrder(SNESLineSearch ls, PetscInt order);
```

for instance, 2 for quadratic or 3 for cubic. Sometimes, it may not be necessary to monitor the progress of the nonlinear iteration. In this case, -snes_linesearch_norms or

```
SNESLineSearchSetComputeNorms(SNESLineSearch ls, PetscBool norms);
```

may be used to turn off function, step, and solution norm computation at the end of the linesearch.

The default line search for the line search Newton method, SNESLINESEARCHBT involves several parameters, which are set to defaults that are reasonable for many applications. The user can override the defaults by using the following options:

- -snes_linesearch_alpha alpha
- -snes_linesearch_maxstep max
- -snes_linesearch_minlambda tol

Besides the backtracking linesearch, there are SNESLINESEARCHSECANT, which uses a polynomial secant minimization of $||F(x)||_2$ or an objective function if set, 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.

For highly non-linear problems, the bisection line search SNESLINESEARCHBISECTION may prove useful due to its robustness. Similar to the critical point line search SNESLINESEARCHCP, it seeks to find the root of $F(x) \cdot Y$. While the latter does so through a secant method, the bisection line search does so by iteratively bisecting the step length interval. It works as follows (with $f(\lambda) = F(x - \lambda Y) \cdot Y / ||Y||$ for brevity):

1. initialize: $j = 1$, $\lambda_0 = \lambda_{\text{left}} = 0.0$, $\lambda_j = \lambda_{\text{right}} = \alpha$, compute $f(\lambda_0)$ and $f(\lambda_j)$
2. check whether there is a change of sign in the interval: $f(\lambda_{\text{left}}) f(\lambda_j) \leq 0$; if not accept the full step length λ_1
3. if there is a change of sign, enter iterative bisection procedure
    1. check convergence/ exit criteria:

- absolute tolerance $f(\lambda_j) <$ `atol`
- relative tolerance $f(\lambda_j) <$ `rtol` $\cdot f(\lambda_0)$
- change of step length $\lambda_j - \lambda_{j-1} <$ `ltol`
- number of iterations $j <$ `max_it`

2. if $j > 1$, determine direction of bisection

$$\lambda_{\text{left}} = \begin{cases} \lambda_{\text{left}} & f(\lambda_{\text{left}})f(\lambda_j) \leq 0 \\ \lambda_j & \text{else} \end{cases}$$

$$\lambda_{\text{right}} = \begin{cases} \lambda_j & f(\lambda_{\text{left}})f(\lambda_j) \leq 0 \\ \lambda_{\text{right}} & \text{else} \end{cases}$$

3. bisect the interval: $\lambda_{j+1} = (\lambda_{\text{left}} + \lambda_{\text{right}})/2$, compute $f(\lambda_{j+1})$
4. update variables for the next iteration: $\lambda_j \leftarrow \lambda_{j+1}$, $f(\lambda_j) \leftarrow f(\lambda_{j+1})$, $j \leftarrow j + 1$

Custom line search types may either be defined using `SNESLineSearchShell`, or by creating a custom user line search type in the model of the preexisting ones and register it using

```
SNESLineSearchRegister(const char sname[], PetscErrorCode␣
↪(*function)(SNESLineSearch));.
```

Trust Region Methods

The trust region method in `SNES` for solving systems of nonlinear equations, `SNESNEWTONTR` (`-snes_type newtontr`), is similar to the one developed in the MINPACK project [MoreSGH84]. Several parameters can be set to control the variation of the trust region size during the solution process. In particular, the user can control the initial trust region radius, computed by

$$\Delta = \Delta_0 \|F_0\|_2,$$

by setting Δ_0 via the option `-snes_tr_delta0 delta0`.

Newton with Arc Length Continuation

The Newton method with arc length continuation reformulates the linearized system $K\delta\mathbf{x} = -\mathbf{F}(\mathbf{x})$ by introducing the load parameter λ and splitting the residual into two components, commonly corresponding to internal and external forces:

$$\mathbf{F}(x, \lambda) = \mathbf{F}^{\text{int}}(\mathbf{x}) - \mathbf{F}^{\text{ext}}(\mathbf{x}, \lambda)$$

Often, $\mathbf{F}^{\text{ext}}(\mathbf{x}, \lambda)$ is linear in λ, which can be thought of as applying the external force in proportional load increments. By default, this is how the right-hand side vector is handled in the implemented method. Generally, however, $\mathbf{F}^{\text{ext}}(\mathbf{x}, \lambda)$ may depend non-linearly on λ or $\mathbf{x}$, or both. To accommodate this possibility, we provide the `SNESNewtonALGetLoadParameter()` function, which allows for the current value of λ to be queried in the functions provided to `SNESSetFunction()` and `SNESSetJacobian()`.

Additionally, we split the solution update into two components:

$$\delta\mathbf{x} = \delta s \delta\mathbf{x}^F + \delta\lambda\delta\mathbf{x}^Q,$$

where $\delta s = 1$ unless partial corrections are used (discussed more below). Each of $\delta\mathbf{x}^F$ and $\delta\mathbf{x}^Q$ are found via solving a linear system with the Jacobian K:

- $\delta \mathbf{x}^F$ is the full Newton step for a given value of λ: $K\delta \mathbf{x}^F = -\mathbf{F}(\mathbf{x}, \lambda)$
- $\delta \mathbf{x}^Q$ is the variation in $\mathbf{x}$ with respect to λ, computed by $K\delta \mathbf{x}^Q = \mathbf{Q}(\mathbf{x}, \lambda)$, where $\mathbf{Q}(\mathbf{x}, \lambda) = -\partial \mathbf{F}(\mathbf{x}, \lambda)/\partial \lambda$ is the tangent load vector.

Often, the tangent load vector $\mathbf{Q}$ is constant within a load increment, which corresponds to the case of proportional loading discussed above. By default, $\mathbf{Q}$ is the full right-hand-side vector, if one was provided. The user can also provide a function which computes $\mathbf{Q}$ to `SNESNewtonALSetFunction()`. This function should have the same signature as for `SNESSetFunction`, and the user should use `SNESNewtonALGetLoadParameter()` to get λ if it is needed.

The Constraint Surface. Considering the $n + 1$ dimensional space of $\mathbf{x}$ and λ, we define the linearized equilibrium line to be the set of points for which the linearized equilibrium equations are satisfied. Given the previous iterative solution $\mathbf{t}^{(j-1)} = [\mathbf{x}^{(j-1)}, \lambda^{(j-1)}]$, this line is defined by the point $\mathbf{t}^{(j-1)} + [\delta \mathbf{x}^F, 0]$ and the vector $\mathbf{t}^Q[\delta \mathbf{x}^Q, 1]$. The arc length method seeks the intersection of this linearized equilibrium line with a quadratic constraint surface, defined by

where L is a user-provided step size corresponding to the radius of the constraint surface, $\Delta \mathbf{x}$ and $\Delta\lambda$ are the accumulated updates over the current load step, and ψ^2 is a user-provided consistency parameter determining the shape of the constraint surface. Generally, $\psi^2 > 0$ leads to a hyper-sphere constraint surface, while $\psi^2 = 0$ leads to a hyper-cylinder constraint surface.

Since the solution will always fall on the constraint surface, the method will often require multiple incremental steps to fully solve the non-linear problem. This is necessary to accurately trace the equilibrium path. Importantly, this is fundamentally different from time stepping. While a similar process could be implemented as a `TS`, this method is particularly designed to be used as a SNES, either standalone or within a `TS`.

To this end, by default, the load parameter is used such that the full external forces are applied at $\lambda = 1$, although we allow for the user to specify a different value via `-snes_newtonal_lambda_max`. To ensure that the solution corresponds exactly to the external force prescribed by the user, i.e. that the load parameter is exactly λ_{max} at the end of the SNES solve, we clamp the value before computing the solution update. As such, the final increment will likely be a hybrid of arc length continuation and normal Newton iterations.

Choosing the Continuation Step. For the first iteration from an equilibrium point, there is a single correct way to choose $\delta\lambda$, which follows from the constraint equations. Specifically the constraint equations yield the quadratic equation $a\delta\lambda^2 + b\delta\lambda + c = 0$, where

$$
\begin{aligned}
a &= \|\delta \mathbf{x}^Q\|^2 + \psi^2, \\
b &= 2\delta \mathbf{x}^Q \cdot (\Delta \mathbf{x} + \delta s \delta \mathbf{x}^F) + 2\psi^2 \Delta\lambda, \\
c &= \|\Delta \mathbf{x} + \delta s \delta \mathbf{x}^F\|^2 + \psi^2 \Delta\lambda^2 - L^2.
\end{aligned}
$$

Since in the first iteration, $\Delta \mathbf{x} = \delta \mathbf{x}^F = \mathbf{0}$ and $\Delta\lambda = 0$, $b = 0$ and the equation simplifies to a pair of real roots:

$$
\delta\lambda = \pm \frac{L}{\sqrt{\|\delta \mathbf{x}^Q\|^2 + \psi^2}},
$$

where the sign is positive for the first increment and is determined by the previous increment otherwise as

$$
\operatorname{sign}(\delta\lambda) = \operatorname{sign}\big(\delta \mathbf{x}^Q \cdot (\Delta \mathbf{x})_{i-1} + \psi^2 (\Delta\lambda)_{i-1}\big),
$$

where $(\Delta \mathbf{x})_{i-1}$ and $(\Delta\lambda)_{i-1}$ are the accumulated updates over the previous load step.

In subsequent iterations, there are different approaches to selecting $\delta\lambda$, all of which have trade-offs. The main difference is whether the iterative solution falls on the constraint surface at every iteration, or only when fully converged. PETSc implements two approaches, set via `SNESNewtonALSetCorrectionType()` or `-snes_newtonal_correction_type (normal|exact)` on the command line.

Corrections in the Normal Hyperplane. The SNES_NEWTONAL_CORRECTION_NORMAL option is simpler and computationally less expensive, but may fail to converge, as the constraint equation is not satisfied at every iteration. The update $\delta\lambda$ is chosen such that the update is within the normal hyper-surface to the quadratic constraint surface. Mathematically, that is

$$\delta\lambda = -\frac{\Delta\mathbf{x} \cdot \delta\mathbf{x}^F}{\Delta\mathbf{x} \cdot \delta\mathbf{x}^Q + \psi^2\Delta\lambda}.$$

This implementation is based on [LPP+11].

Exact Corrections. The SNES_NEWTONAL_CORRECTION_EXACT option is far more complex, but ensures that the constraint is exactly satisfied at every Newton iteration. As such, it is generally more robust. By evaluating the intersection of constraint surface and equilibrium line at each iteration, $\delta\lambda$ is chosen as one of the roots of the above quadratic equation $a\delta\lambda^2 + b\delta\lambda + c = 0$. This method encounters issues, however, if the linearized equilibrium line and constraint surface do not intersect due to particularly large linearized error. In this case, the roots are complex. To continue progressing toward a solution, this method uses a partial correction by choosing δs such that the quadratic equation has a single real root. Geometrically, this is selecting the point on the constraint surface closest to the linearized equilibrium line. See the code or [RCorreaC08] for a mathematical description of these partial corrections.

Nonlinear Krylov Methods

A number of nonlinear Krylov methods are provided, including Nonlinear Richardson (SNESNRICHARDSON), nonlinear conjugate gradient (SNESNCG), nonlinear GMRES (SNESNGMRES), and Anderson Mixing (SNESANDERSON). These methods are described individually below. They are all instrumental to PETSc's nonlinear preconditioning.

Nonlinear Richardson. The nonlinear Richardson iteration, SNESNRICHARDSON, merely takes the form of a line search-damped fixed-point iteration of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda\mathbf{F}(\mathbf{x}_k), \quad k = 0, 1, \ldots,$$

where the default linesearch is SNESLINESEARCHSECANT. 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, SNESNCG, is equivalent to linear CG, but with the steplength determined by line search (SNESLINESEARCHCP by default). Five variants (Fletcher-Reed, Hestenes-Steifel, Polak-Ribiere-Polyak, Dai-Yuan, and Conjugate Descent) are implemented in PETSc and may be chosen using

```
SNESNCGSetType(SNES snes, SNESNCGType btype);
```

Anderson Mixing and Nonlinear GMRES Methods. Nonlinear GMRES (SNESNGMRES), and Anderson Mixing (SNESANDERSON) methods combine the last m iterates, plus a new fixed-point iteration iterate, into an approximate residual-minimizing new iterate.

All of the above methods have support for using a nonlinear preconditioner to compute the preliminary update step, rather than the default which is the nonlinear function's residual, $\mathbf{F}(\mathbf{x}_k)$. The different update is obtained by solving a nonlinear preconditioner nonlinear problem, which has its own SNES object that may be obtained with SNESGetNPC().

Quasi-Newton Methods

Quasi-Newton methods store iterative rank-one updates to the Jacobian instead of computing the Jacobian 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.

The quasi-Newton methods support the use of a nonlinear preconditioner that can be obtained with `SNESGetNPC()` and then configured; or that can be configured with `SNES`, `KSP`, and `PC` options using the options database prefix `-npc_`.

Table 2.10: PETSc quasi-Newton solvers

QN Method	SNESQNType	Options Name	Default Line Search
L-BFGS	`SNES_QN_LBFGS`	`lbfgs`	`SNESLINESEARCHCP`
"Good" Broyden	`SNES_QN_BROYDEN`	`broyden`	`SNESLINESEARCHBASIC` (or equivalently `SNESLINESEARCHNONE`)
"Bad" Broyden	`SNES_QN_BADBROY`	`badbroyden`	`SNESLINESEARCHSECANT`

One may also control the form of the initial Jacobian approximation with

```
SNESQNSetScaleType(SNES snes, SNESQNScaleType stype);
```

and the restart type with

```
SNESQNSetRestartType(SNES snes, SNESQNRestartType rtype);
```

The Full Approximation Scheme

The Nonlinear Full Approximation Scheme (FAS) `SNESFAS`, is a nonlinear multigrid method. At each level, there is a recursive cycle control `SNES` instance, and either one or two nonlinear solvers that act as smoothers (up and down). Problems set up using the `SNES DMDA` interface are automatically coarsened. FAS, `SNESFAS`, differs slightly from linear multigrid `PCMG`, in that the hierarchy is constructed recursively. However, much of the interface is a one-to-one map. We describe the "get" operations here, and it can be assumed that each has a corresponding "set" operation. For instance, the number of levels in the hierarchy may be retrieved using

```
SNESFASGetLevels(SNES snes, PetscInt *levels);
```

There are four `SNESFAS` cycle types, `SNES_FAS_MULTIPLICATIVE`, `SNES_FAS_ADDITIVE`, `SNES_FAS_FULL`, and `SNES_FAS_KASKADE`. The type may be set with

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

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

```
SNESFASSetCycles(SNES snes, PetscInt cycles);.
```

Much like the interface to `PCMG` described in *Multigrid Preconditioners*, there are interfaces to recover the various levels' cycles and smoothers. The level smoothers may be accessed with

```
SNESFASGetSmoother(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmootherUp(SNES snes, PetscInt level, SNES *smooth);
SNESFASGetSmootherDown(SNES snes, PetscInt level, SNES *smooth);
```

and the level cycles with

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

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

```
SNESFASGetInterpolation(SNES snes, PetscInt level, Mat *mat);
SNESFASGetRestriction(SNES snes, PetscInt level, Mat *mat);
```

In addition, FAS requires special restriction for solution-like variables, called injection. This may be set with

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

The coarse solve context may be acquired with

```
SNESFASGetCoarseSolve(SNES snes, SNES *smooth);
```

Nonlinear Additive Schwarz

Nonlinear Additive Schwarz methods (NASM) take a number of local nonlinear subproblems, solves them independently in parallel, and combines those solutions into a new approximate solution.

```
SNESNASMSetSubdomains(SNES snes, PetscInt n, SNES subsnes[], VecScatter iscatter[],␣
↪VecScatter oscatter[], VecScatter gscatter[]);
```

allows for the user to create these local subdomains. Problems set up using the SNES DMDA interface are automatically decomposed. To begin, the type of subdomain updates to the whole solution are limited to two types borrowed from PCASM: PC_ASM_BASIC, in which the overlapping updates added. PC_ASM_RESTRICT updates in a nonoverlapping fashion. This may be set with

```
SNESNASMSetType(SNES snes, PCASMType type);.
```

SNESASPIN is a helper SNES type that sets up a nonlinearly preconditioned Newton's method using NASM as the preconditioner.

2.5.4 General Options

This section discusses options and routines that apply to all SNES solvers and problem classes. In particular, we focus on convergence tests, monitoring routines, and tools for checking derivative computations.

Convergence Tests

Convergence of the nonlinear solvers can be detected in a variety of ways; the user can even specify a customized test, as discussed below. Most of the nonlinear solvers use `SNESConvergenceTestDefault()`, however, `SNESNEWTONTR` uses a method-specific additional convergence test as well. The convergence tests involves several parameters, which are set by default to values that should be reasonable for a wide range of problems. The user can customize the parameters to the problem at hand by using some of the following routines and options.

One method of convergence testing is to declare convergence when the norm of the change in the solution between successive iterations is less than some tolerance, `stol`. Convergence can also be determined based on the norm of the function. Such a test can use either the absolute size of the norm, `atol`, or its relative decrease, `rtol`, from an initial guess. The following routine sets these parameters, which are used in many of the default `SNES` convergence tests:

```
SNESSetTolerances(SNES snes, PetscReal atol, PetscReal rtol, PetscReal stol, PetscInt␣
↪its, PetscInt fcts);
```

This routine also sets the maximum numbers of allowable nonlinear iterations, `its`, and function evaluations, `fcts`. The corresponding options database commands for setting these parameters are:

- `-snes_atol atol`
- `-snes_rtol rtol`
- `-snes_stol stol`
- `-snes_max_it its`
- `-snes_max_funcs fcts` (use `unlimited` for no maximum)

A related routine is `SNESGetTolerances()`. `PETSC_CURRENT` may be used for any parameter to indicate the current value should be retained; use `PETSC_DETERMINE` to restore to the default value from when the object was created.

Users can set their own customized convergence tests in `SNES` by using the command

```
SNESSetConvergenceTest(SNES snes, PetscErrorCode (*test)(SNES snes, PetscInt it,␣
↪PetscReal xnorm, PetscReal gnorm, PetscReal f, SNESConvergedReason reason, PetscCtx␣
↪cctx), PetscCtx cctx, PetscCtxDestroyFn *destroy);
```

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

Convergence Monitoring

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

```
SNESMonitorSet(SNES snes, PetscErrorCode (*mon)(SNES snes, PetscInt its, PetscReal␣
↪norm, PetscCtx mctx), PetscCtx mctx, (PetscCtxDestroyFn *)*monitordestroy);
```

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

The routine set by SNESMonitorSet() is called once after every successful step computation within the nonlinear solver. Hence, the user can employ this routine for any application-specific computations that should be done after the solution update. The option -snes_monitor activates the default SNES monitor routine, SNESMonitorDefault(), while -snes_monitor_lg_residualnorm draws a simple line graph of the residual norm's convergence.

One can cancel hardwired monitoring routines for SNES at runtime with -snes_monitor_cancel.

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

The routines

```
SNESGetSolution(SNES snes, Vec *x);
SNESGetFunction(SNES snes, Vec *r, PetscCtxRt ctx, int(**func)(SNES, Vec, Vec,␣
↪PetscCtx));
```

return the solution vector and function vector from a SNES context. These routines are useful, for instance, if the convergence test requires some property of the solution or function other than those passed with routine arguments.

Checking Accuracy of Derivatives

Since hand-coding routines for Jacobian matrix evaluation can be error prone, SNES provides easy-to-use support for checking these matrices against finite difference versions. In the simplest form of comparison, users can employ the option -snes_test_jacobian to compare the matrices at several points. Although not exhaustive, this test will generally catch obvious problems. One can compare the elements of the two matrices by using the option -snes_test_jacobian_view , which causes the two matrices to be printed to the screen.

Another means for verifying the correctness of a code for Jacobian computation is running the problem with either the finite difference or matrix-free variant, -snes_fd or -snes_mf; see *Finite Difference Jacobian Approximations* or *Matrix-Free Methods*. If a problem converges well with these matrix approximations but not with a user-provided routine, the problem probably lies with the hand-coded matrix. See the note in *Jacobian Evaluation* about assembling your Jabobian in the "preconditioner" slot Pmat.

The correctness of user provided MATSHELL Jacobians in general can be checked with MatShellTestMultTranspose() and MatShellTestMult().

The correctness of user provided MATSHELL Jacobians via TSSetRHSJacobian() can be checked with TSRHSJacobianTestTranspose() and TSRHSJacobianTest() that check the correction of the matrix-transpose vector product and the matrix-product. From the command line, these can be checked with

- -ts_rhs_jacobian_test_mult_transpose
- -mat_shell_test_mult_transpose_view
- -ts_rhs_jacobian_test_mult
- -mat_shell_test_mult_view

2.5.5 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_type` and `-pc_type pc_method`, to set the Krylov subspace and preconditioner methods.

Two levels of iterations occur for the inexact techniques, where during each global or outer Newton iteration a sequence of subsidiary inner iterations of a linear solver is performed. Appropriate control of the accuracy to which the subsidiary iterative method solves the Newton system at each global iteration is critical, since these inner iterations determine the asymptotic convergence rate for inexact Newton techniques. While the Newton systems must be solved well enough to retain fast local convergence of the Newton's iterates, use of excessive inner iterations, particularly when $\|\mathbf{x}_k - \mathbf{x}_*\|$ is large, is neither necessary nor economical. Thus, the number of required inner iterations typically increases as the Newton process progresses, so that the truncated iterates approach the true Newton iterates.

A sequence of nonnegative numbers $\{\eta_k\}$ can be used to indicate the variable convergence criterion. In this case, when solving a system of nonlinear equations, the update step of the Newton process remains unchanged, and direct solution of the linear system is replaced by iteration on the system until the residuals

$$\mathbf{r}_k^{(i)} = \mathbf{F}'(\mathbf{x}_k)\Delta\mathbf{x}_k + \mathbf{F}(\mathbf{x}_k)$$

satisfy

$$\frac{\|\mathbf{r}_k^{(i)}\|}{\|\mathbf{F}(\mathbf{x}_k)\|} \leq \eta_k \leq \eta < 1.$$

Here $\mathbf{x}_0$ is an initial approximation of the solution, and $\|\cdot\|$ denotes an arbitrary norm in $\Re^n$.

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

2.5.6 Matrix-Free Methods

The `SNES` class fully supports matrix-free methods. The matrices specified in the Jacobian evaluation routine need not be conventional matrices; instead, they can point to the data required to implement a particular matrix-free method. The matrix-free variant is allowed *only* when the linear systems are solved by an iterative method in combination with no preconditioning (`PCNONE` or `-pc_type none`), a user-provided matrix from which to construct the preconditioner, or a user-provided preconditioner shell (`PCSHELL`, discussed in *Preconditioners*); that is, obviously matrix-free methods cannot be used with a direct solver, approximate factorization, or other preconditioner which requires access to explicit matrix entries.

The user can create a matrix-free context for use within `SNES` with the routine

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

This routine creates the data structures needed for the matrix-vector products that arise within Krylov space iterative methods [BS90]. The default SNES matrix-free approximations can also be invoked with the command `-snes_mf`. Or, one can retain the user-provided Jacobian preconditioner, but replace the user-provided Jacobian matrix with the default matrix-free variant with the option `-snes_mf_operator`.

`MatCreateSNESMF()` uses

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

which can also be used directly for users who need a matrix-free matrix but are not using SNES.

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

```
MatMFFDSetFunctionError(Mat mat, PetscReal rerror);
```

The parameter `rerror` should be set to the square root of the relative error in the function evaluations, e_{rel}; the default is the square root of machine epsilon (about 10^{-8} in double precision), which assumes that the functions are evaluated to full floating-point precision accuracy. This parameter can also be set from the options database with `-mat_mffd_err err`

In addition, PETSc provides ways to register new routines to compute the differencing parameter (h); see the manual page for `MatMFFDSetType()` and `MatMFFDRegister()`. We currently provide two default routines accessible via `-mat_mffd_type (ds|wp)`. For the default approach there is one "tuning" parameter, set with

```
MatMFFDDSSetUmin(Mat mat, PetscReal umin);
```

This parameter, `umin` (or u_{min}), is a bit involved; its default is 10^{-6} . Its command line form is `-mat_mffd_umin umin`.

The Jacobian-vector product is approximated via the formula

$$F'(u)a \approx \frac{F(u + h * a) - F(u)}{h}$$

where h is computed via

$$h = e_{\text{rel}} \cdot \begin{cases} u^T a / \|a\|_2^2 & \text{if } |u^T a| > u_{\min} \|a\|_1 \\ u_{\min} \operatorname{sign}(u^T a) \|a\|_1 / \|a\|_2^2 & \text{otherwise.} \end{cases}$$

This approach is taken from Brown and Saad [BS90]. The second approach, taken from Walker and Pernice, [PW98], computes h via

$$h = \frac{\sqrt{1 + ||u||} e_{rel}}{||a||}$$

This has no tunable parameters, but note that inside the nonlinear solve for the entire *linear* iterative process u does not change hence $\sqrt{1 + ||u||}$ need be computed only once. This information may be set with the options

```
MatMFFDWPSetComputeNormU(Mat, PetscBool);
```

or `-mat_mffd_compute_normu (true|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. This takes place automatically for the PETSc GMRES solver with left preconditioning.

It is also possible to monitor the differencing parameters h that are computed via the routines

```
MatMFFDSetHHistory(Mat, PetscScalar *, int);
MatMFFDResetHHistory(Mat, PetscScalar *, int);
MatMFFDGetH(Mat, PetscScalar *);
```

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

Listing: `src/snes/tutorials/ex3.c`

```
static char help[] = "Newton methods to solve u'' + u^{2} = f in parallel.\n\
This example employs a user-defined monitoring routine and optionally a user-defined\
↪n\
routine to check candidate iterates produced by line search routines.\n\
The command line options include:\n\
  -pre_check_iterates : activate checking of iterates\n\
  -post_check_iterates : activate checking of iterates\n\
  -check_tol <tol>: set tolerance for iterate checking\n\
  -user_precond : activate a (trivial) user-defined preconditioner\n\n";

/*
   Include "petscdm.h" so that we can use data management objects (DMs)
   Include "petscdmda.h" so that we can use distributed arrays (DMDAs).
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petscsys.h    - base PETSc routines
     petscvec.h    - vectors
     petscmat.h    - matrices
     petscis.h     - index sets
     petscksp.h    - Krylov subspace methods
     petscviewer.h - viewers
     petscpc.h     - preconditioners
     petscksp.h    - linear solvers
*/

#include <petscdm.h>
#include <petscdmda.h>
#include <petscsnes.h>

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

/*
```

(continues on next page)

(continued from previous page)

```
   User-defined application context
*/
typedef struct {
  DM          da;    /* distributed array */
  Vec         F;     /* right-hand side of PDE */
  PetscMPIInt rank;  /* rank of processor */
  PetscMPIInt size;  /* size of communicator */
  PetscReal   h;     /* mesh spacing */
  PetscBool   sjerr; /* if or not to test jacobian domain error */
} ApplicationCtx;

/*
   User-defined context for monitoring
*/
typedef struct {
  PetscViewer viewer;
} MonitorCtx;

/*
   User-defined context for checking candidate iterates that are
   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;
  PetscBool      pre_check, post_check, post_setsubksp; /* flag indicating whether we
↪'re checking candidate iterates */
  PetscScalar    xp, *FF, *UU, none = -1.0;
  PetscInt       its, N = 5, i, maxit, maxf, xs, xm;
  PetscReal      abstol, rtol, stol, norm;
  PetscBool      flg, viewinitial = PETSC_FALSE;

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));
  PetscCallMPI(MPI_Comm_rank(PETSC_COMM_WORLD, &ctx.rank));
  PetscCallMPI(MPI_Comm_size(PETSC_COMM_WORLD, &ctx.size));
  PetscCall(PetscOptionsGetInt(NULL, NULL, "-n", &N, NULL));
  ctx.h     = 1.0 / (N - 1);
  ctx.sjerr = PETSC_FALSE;
```

(continues on next page)

(continued from previous page)

```
  PetscCall(PetscOptionsGetBool(NULL, NULL, "-test_jacobian_domain_error", &ctx.sjerr,
↪ NULL));
  PetscCall(PetscOptionsGetBool(NULL, NULL, "-view_initial", &viewinitial, NULL));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  PetscCall(SNESCreate(PETSC_COMM_WORLD, &snes));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create vector data structures; set function evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Create distributed array (DMDA) to manage parallel grid and vectors
  */
  PetscCall(DMDACreate1d(PETSC_COMM_WORLD, DM_BOUNDARY_NONE, N, 1, 1, NULL, &ctx.da));
  PetscCall(DMSetFromOptions(ctx.da));
  PetscCall(DMSetUp(ctx.da));

  /*
     Extract global and local vectors from DMDA; then duplicate for remaining
     vectors that are the same types
  */
  PetscCall(DMCreateGlobalVector(ctx.da, &x));
  PetscCall(VecDuplicate(x, &r));
  PetscCall(VecDuplicate(x, &F));
  ctx.F = F;
  PetscCall(VecDuplicate(x, &U));

  /*
     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.
  */
  PetscCall(SNESSetFunction(snes, r, FormFunction, &ctx));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create matrix data structure; set Jacobian evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  PetscCall(MatCreate(PETSC_COMM_WORLD, &J));
  PetscCall(MatSetSizes(J, PETSC_DECIDE, PETSC_DECIDE, N, N));
  PetscCall(MatSetFromOptions(J));
  PetscCall(MatSeqAIJSetPreallocation(J, 3, NULL));
  PetscCall(MatMPIAIJSetPreallocation(J, 3, NULL, 3, NULL));

  /*
     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
```

(continues on next page)

(continued from previous page)

```
       context that provides application-specific data for the
       Jacobian evaluation routine.
  */
  PetscCall(SNESSetJacobian(snes, J, J, FormJacobian, &ctx));

  /*
     Optionally allow user-provided preconditioner
   */
  PetscCall(PetscOptionsHasName(NULL, NULL, "-user_precond", &flg));
  if (flg) {
    KSP ksp;
    PC  pc;
    PetscCall(SNESGetKSP(snes, &ksp));
    PetscCall(KSPGetPC(ksp, &pc));
    PetscCall(PCSetType(pc, PCSHELL));
    PetscCall(PCShellSetApply(pc, MatrixFreePreconditioner));
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Customize nonlinear solver; set runtime options
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Set an optional user-defined monitoring routine
  */
  PetscCall(PetscViewerDrawOpen(PETSC_COMM_WORLD, 0, 0, 0, 0, 400, 400, &monP.
↪viewer));
  PetscCall(SNESMonitorSet(snes, Monitor, &monP, 0));

  /*
     Set names for some vectors to facilitate monitoring (optional)
  */
  PetscCall(PetscObjectSetName((PetscObject)x, "Approximate Solution"));
  PetscCall(PetscObjectSetName((PetscObject)U, "Exact Solution"));

  /*
     Set SNES/KSP/KSP/PC runtime options, e.g.,
         -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
  */
  PetscCall(SNESSetFromOptions(snes));

  /*
     Set an optional user-defined routine to check the validity of candidate
     iterates that are determined by line search methods
  */
  PetscCall(SNESGetLineSearch(snes, &linesearch));
  PetscCall(PetscOptionsHasName(NULL, NULL, "-post_check_iterates", &post_check));

  if (post_check) {
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Activating post step checking routine\n
↪"));
    PetscCall(SNESLineSearchSetPostCheck(linesearch, PostCheck, &checkP));
    PetscCall(VecDuplicate(x, &checkP.last_step));

    checkP.tolerance = 1.0;
    checkP.user      = &ctx;
```

(continues on next page)

(continued from previous page)

```
    PetscCall(PetscOptionsGetReal(NULL, NULL, "-check_tol", &checkP.tolerance, NULL));
  }

  PetscCall(PetscOptionsHasName(NULL, NULL, "-post_setsubksp", &post_setsubksp));
  if (post_setsubksp) {
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Activating post setsubksp\n"));
    PetscCall(SNESLineSearchSetPostCheck(linesearch, PostSetSubKSP, &checkP1));
  }

  PetscCall(PetscOptionsHasName(NULL, NULL, "-pre_check_iterates", &pre_check));
  if (pre_check) {
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Activating pre step checking routine\n
↪"));
    PetscCall(SNESLineSearchSetPreCheck(linesearch, PreCheck, &checkP));
  }

  /*
     Print parameters used for convergence testing (optional) ... just
     to demonstrate this routine; this information is also printed with
     the option -snes_view
  */
  PetscCall(SNESGetTolerances(snes, &abstol, &rtol, &stol, &maxit, &maxf));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "atol=%g, rtol=%g, stol=%g, maxit=%"␣
↪PetscInt_FMT ", maxf=%" PetscInt_FMT "\n", (double)abstol, (double)rtol,␣
↪(double)stol, maxit, maxf));

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     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)
  */
  PetscCall(DMDAGetCorners(ctx.da, &xs, NULL, NULL, &xm, NULL, NULL));

  /*
     Get pointers to vector data
  */
  PetscCall(DMDAVecGetArray(ctx.da, F, &FF));
  PetscCall(DMDAVecGetArray(ctx.da, U, &UU));

  /*
     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;
  }

  /*
```

(continues on next page)

(continued from previous page)

```
   Restore vectors
 */
 PetscCall(DMDAVecRestoreArray(ctx.da, F, &FF));
 PetscCall(DMDAVecRestoreArray(ctx.da, U, &UU));
 if (viewinitial) {
   PetscCall(VecView(U, 0));
   PetscCall(VecView(F, 0));
 }

 /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Evaluate initial guess; then solve nonlinear system
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

 /*
    Note: The user should initialize the vector, x, with the initial guess
    for the nonlinear solver prior to calling SNESSolve().  In particular,
    to employ an initial guess of zero, the user should explicitly set
    this vector to zero by calling VecSet().
 */
 PetscCall(FormInitialGuess(x));
 PetscCall(SNESSolve(snes, NULL, x));
 PetscCall(SNESGetIterationNumber(snes, &its));
 PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Number of SNES iterations = %" PetscInt_
↪FMT "\n", its));

 /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Check solution and clean up
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

 /*
    Check the error
 */
 PetscCall(VecAXPY(x, none, U));
 PetscCall(VecNorm(x, NORM_2, &norm));
 PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Norm of error %g Iterations %" PetscInt_
↪FMT "\n", (double)norm, its));
 if (ctx.sjerr) {
   SNESType snestype;
   PetscCall(SNESGetType(snes, &snestype));
   PetscCall(PetscPrintf(PETSC_COMM_WORLD, "SNES Type %s\n", snestype));
 }

 /*
    Free work space.  All PETSc objects should be destroyed when they
    are no longer needed.
 */
 PetscCall(PetscViewerDestroy(&monP.viewer));
 if (post_check) PetscCall(VecDestroy(&checkP.last_step));
 PetscCall(VecDestroy(&x));
 PetscCall(VecDestroy(&r));
 PetscCall(VecDestroy(&U));
 PetscCall(VecDestroy(&F));
 PetscCall(MatDestroy(&J));
 PetscCall(SNESDestroy(&snes));
 PetscCall(DMDestroy(&ctx.da));
 PetscCall(PetscFinalize());
```

(continues on next page)

(continued from previous page)

```
  return 0;
}

/* ------------------------------------------------------------------- */
/*
   FormInitialGuess - Computes initial guess.

   Input/Output Parameter:
.  x - the solution vector
*/
PetscErrorCode FormInitialGuess(Vec x)
{
  PetscScalar pfive = .50;

  PetscFunctionBeginUser;
  PetscCall(VecSet(x, pfive));
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   FormFunction - Evaluates nonlinear function, F(x).

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  ctx - optional user-defined context, as set by SNESSetFunction()

   Output Parameter:
.  f - function vector

   Note:
   The user-defined context can contain any application-specific
   data needed for the function evaluation.
*/
PetscErrorCode FormFunction(SNES snes, Vec x, Vec f, PetscCtx ctx)
{
  ApplicationCtx    *user = (ApplicationCtx *)ctx;
  DM                 da   = user->da;
  PetscScalar       *ff, d;
  const PetscScalar *xx, *FF;
  PetscInt           i, M, xs, xm;
  Vec                xlocal;

  PetscFunctionBeginUser;
  PetscCall(DMGetLocalVector(da, &xlocal));
  /*
     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.
  */
  PetscCall(DMGlobalToLocalBegin(da, x, INSERT_VALUES, xlocal));
  PetscCall(DMGlobalToLocalEnd(da, x, INSERT_VALUES, xlocal));

  /*
```

(continues on next page)

(continued from previous page)

```
    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
  */
  PetscCall(DMDAVecGetArrayRead(da, xlocal, (void *)&xx));
  PetscCall(DMDAVecGetArray(da, f, &ff));
  PetscCall(DMDAVecGetArrayRead(da, user->F, (void *)&FF));

  /*
     Get local grid boundaries (for 1-dimensional DMDA):
       xs, xm  - starting grid index, width of local grid (no ghost points)
  */
  PetscCall(DMDAGetCorners(da, &xs, NULL, NULL, &xm, NULL, NULL));
  PetscCall(DMDAGetInfo(da, NULL, &M, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,␣
↪NULL, NULL, NULL));

  /*
     Set function values for boundary points; define local interior grid point range:
        xsi - starting interior grid index
        xei - ending interior grid index
  */
  if (xs == 0) { /* left boundary */
    ff[0] = xx[0];
    xs++;
    xm--;
  }
  if (xs + xm == M) { /* right boundary */
    ff[xs + xm - 1] = xx[xs + xm - 1] - 1.0;
    xm--;
  }

  /*
     Compute function over locally owned part of the grid (interior points only)
  */
  d = 1.0 / (user->h * user->h);
  for (i = xs; i < xs + xm; i++) ff[i] = d * (xx[i - 1] - 2.0 * xx[i] + xx[i + 1]) +␣
↪xx[i] * xx[i] - FF[i];

  /*
     Restore vectors
  */
  PetscCall(DMDAVecRestoreArrayRead(da, xlocal, (void *)&xx));
  PetscCall(DMDAVecRestoreArray(da, f, &ff));
  PetscCall(DMDAVecRestoreArrayRead(da, user->F, (void *)&FF));
  PetscCall(DMRestoreLocalVector(da, &xlocal));
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   FormJacobian - Evaluates Jacobian matrix.

   Input Parameters:
.  snes - the SNES context
.  x - input vector
```

(continues on next page)

(continued from previous page)

```
.  dummy - optional user-defined context (not used here)

   Output Parameters:
.  jac - Jacobian matrix
.  B - optionally different matrix used to construct the preconditioner

*/
PetscErrorCode FormJacobian(SNES snes, Vec x, Mat jac, Mat B, PetscCtx ctx)
{
  ApplicationCtx *user = (ApplicationCtx *)ctx;
  PetscScalar    *xx, d, A[3];
  PetscInt        i, j[3], M, xs, xm;
  DM              da = user->da;

  PetscFunctionBeginUser;
  if (user->sjerr) {
    PetscCall(SNESSetJacobianDomainError(snes));
    PetscFunctionReturn(PETSC_SUCCESS);
  }
  /*
     Get pointer to vector data
  */
  PetscCall(DMDAVecGetArrayRead(da, x, &xx));
  PetscCall(DMDAGetCorners(da, &xs, NULL, NULL, &xm, NULL, NULL));

  /*
    Get range of locally owned matrix
  */
  PetscCall(DMDAGetInfo(da, NULL, &M, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,␣
↪NULL, NULL, NULL));

  /*
     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;

    PetscCall(MatSetValues(jac, 1, &i, 1, &i, A, INSERT_VALUES));
    xs++;
    xm--;
  }
  if (xs + xm == M) { /* right boundary */
    i    = M - 1;
    A[0] = 1.0;
    PetscCall(MatSetValues(jac, 1, &i, 1, &i, A, INSERT_VALUES));
    xm--;
  }

  /*
     Interior grid points
      - Note that in this case we set all elements for a particular
        row at once.
  */
```

(continues on next page)

(continued from previous page)

```
  d = 1.0 / (user->h * user->h);
  for (i = xs; i < xs + xm; i++) {
    j[0] = i - 1;
    j[1] = i;
    j[2] = i + 1;
    A[0] = A[2] = d;
    A[1]        = -2.0 * d + 2.0 * xx[i];
    PetscCall(MatSetValues(jac, 1, &i, 3, j, A, INSERT_VALUES));
  }

  /*
     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.
  */

  PetscCall(MatAssemblyBegin(jac, MAT_FINAL_ASSEMBLY));
  PetscCall(DMDAVecRestoreArrayRead(da, x, &xx));
  PetscCall(MatAssemblyEnd(jac, MAT_FINAL_ASSEMBLY));
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   Monitor - Optional user-defined monitoring routine that views the
   current iterate with an x-window plot. Set by SNESMonitorSet().

   Input Parameters:
   snes - the SNES context
   its - iteration number
   norm - 2-norm function value (may be estimated)
   ctx - optional user-defined context for private data for the
         monitor routine, as set by SNESMonitorSet()

   Note:
   See the manpage for PetscViewerDrawOpen() for useful runtime options,
   such as -nox to deactivate all x-window output.
 */
PetscErrorCode Monitor(SNES snes, PetscInt its, PetscReal fnorm, PetscCtx ctx)
{
  MonitorCtx *monP = (MonitorCtx *)ctx;
  Vec         x;

  PetscFunctionBeginUser;
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "iter = %" PetscInt_FMT ",SNES Function␣
↪norm %g\n", its, (double)fnorm));
  PetscCall(SNESGetSolution(snes, &x));
  PetscCall(VecView(x, monP->viewer));
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
```

(continues on next page)

(continued from previous page)

```
   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, PetscCtx ctx)
{
  PetscFunctionBeginUser;
  *changed_y = PETSC_FALSE;
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   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, PetscCtx ctx)
{
  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;

  PetscCall(SNESLineSearchGetSNES(linesearch, &snes));
  check = (StepCheckCtx *)ctx;
  user  = check->user;
```

(continues on next page)

(continued from previous page)

```
  PetscCall(SNESGetIterationNumber(snes, &iter));

  /* iteration 1 indicates we are working on the second iteration */
  if (iter > 0) {
    da = user->da;
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Checking candidate step at iteration %"␣
↪PetscInt_FMT " with tolerance %g\n", iter, (double)check->tolerance));

    /* Access local array data */
    PetscCall(DMDAVecGetArray(da, check->last_step, &xa_last));
    PetscCall(DMDAVecGetArray(da, x, &xa));
    PetscCall(DMDAGetCorners(da, &xs, NULL, NULL, &xm, NULL, NULL));

    /*
       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.)
    */
    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;
        PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Altering entry %" PetscInt_FMT ":␣
↪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])));
      }
    }
    PetscCall(DMDAVecRestoreArray(da, check->last_step, &xa_last));
    PetscCall(DMDAVecRestoreArray(da, x, &xa));
  }
  PetscCall(VecCopy(x, check->last_step));
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   PostSetSubKSP - Optional user-defined routine that reset SubKSP options when␣
↪hierarchical bjacobi PC is used
   e.g,
     mpiexec -n 8 ./ex3 -nox -n 10000 -ksp_type fgmres -pc_type bjacobi -pc_bjacobi_
↪blocks 4 -sub_ksp_type gmres -sub_ksp_max_it 3 -post_setsubksp -sub_ksp_rtol 1.e-16
   Set by SNESLineSearchSetPostCheck().

   Input Parameters:
   linesearch - the LineSearch context
   xcurrent - current solution
   y - search direction and length
   x    - the new candidate iterate

   Output Parameters:
   y    - proposed step (search direction and length) (possibly changed)
   x    - current iterate (possibly modified)
```

(continues on next page)

(continued from previous page)

```
 */
PetscErrorCode PostSetSubKSP(SNESLineSearch linesearch, Vec xcurrent, Vec y, Vec x,␣
↪PetscBool *changed_y, PetscBool *changed_x, PetscCtx ctx)
{
  SetSubKSPCtx *check;
  PetscInt      iter, its, sub_its, maxit;
  KSP           ksp, sub_ksp, *sub_ksps;
  PC            pc;
  PetscReal     ksp_ratio;
  SNES          snes;

  PetscFunctionBeginUser;
  PetscCall(SNESLineSearchGetSNES(linesearch, &snes));
  check = (SetSubKSPCtx *)ctx;
  PetscCall(SNESGetIterationNumber(snes, &iter));
  PetscCall(SNESGetKSP(snes, &ksp));
  PetscCall(KSPGetPC(ksp, &pc));
  PetscCall(PCBJacobiGetSubKSP(pc, NULL, NULL, &sub_ksps));
  sub_ksp = sub_ksps[0];
  PetscCall(KSPGetIterationNumber(ksp, &its));         /* outer KSP iteration number␣
↪*/
  PetscCall(KSPGetIterationNumber(sub_ksp, &sub_its)); /* inner KSP iteration number␣
↪*/

  if (iter) {
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "    ...PostCheck snes iteration %"␣
↪PetscInt_FMT ", ksp_it %" PetscInt_FMT " %" PetscInt_FMT ", subksp_it %" PetscInt_
↪FMT "\n", iter, check->its0, its, sub_its));
    ksp_ratio = (PetscReal)its / check->its0;
    maxit     = (PetscInt)(ksp_ratio * sub_its + 0.5);
    if (maxit < 2) maxit = 2;
    PetscCall(KSPSetTolerances(sub_ksp, PETSC_CURRENT, PETSC_CURRENT, PETSC_CURRENT,␣
↪maxit));
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "    ...ksp_ratio %g, new maxit %"␣
↪PetscInt_FMT "\n\n", (double)ksp_ratio, maxit));
  }
  check->its0 = its; /* save current outer KSP iteration number */
  PetscFunctionReturn(PETSC_SUCCESS);
}

/* ------------------------------------------------------------------- */
/*
   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)
{
```

(continues on next page)

(continued from previous page)

```
  PetscFunctionBeginUser;
  PetscCall(VecCopy(x, y));
  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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

Matrix Use	Conventional Matrix Formats	Matrix-free versions
Jacobian Matrix	Create matrix with `MatCreate()`*. Assemble matrix with user-defined routine †	Create matrix with `MatCreateShell()`. Use `MatShellSetOperation()` to set various matrix actions, or use `MatCreateMFFD()` or `MatCreateSNESMF()`.
Matrix used to construct the preconditioner	Create matrix with `MatCreate()`*. Assemble matrix with user-defined routine †	Use `SNESGetKSP()` and `KSPGetPC()` to access the `PC`, then use `PCSetType(pc, PCSHELL)` followed by `PCShellSetApply()`.

* Use either the generic `MatCreate()` or a format-specific variant such as `MatCreateAIJ()`.

† Set user-defined matrix formation routine with `SNESSetJacobian()` or with a `DM` variant such as `DMDASNESSetJacobianLocal()`

SNES also provides some less well-integrated code to apply matrix-free finite differencing using an automatically computed measurement of the noise of the functions. This can be selected with `-snes_mf_version 2`; it does not use `MatCreateMFFD()` but has similar options that start with `-snes_mf_` instead of `-mat_mffd_`. Note that this alternative prefix **only** works for version 2 differencing.

2.5.7 Finite Difference Jacobian Approximations

PETSc provides some tools to help approximate the Jacobian matrices efficiently via finite differences. These tools are intended for use in certain situations where one is unable to compute Jacobian matrices analytically, and matrix-free methods do not work well without a preconditioner, due to very poor conditioning. The approximation requires several steps:

- First, one colors the columns of the (not yet built) Jacobian matrix, so that columns of the same color do not share any common rows.
- Next, one creates a `MatFDColoring` data structure that will be used later in actually computing the Jacobian.
- Finally, one tells the nonlinear solvers of `SNES` to use the `SNESComputeJacobianDefaultColor()` routine to compute the Jacobians.

A code fragment that demonstrates this process is given below.

```
ISColoring    iscoloring;
MatFDColoring fdcoloring;
MatColoring   coloring;

/*
  This initializes the nonzero structure of the Jacobian. This is artificial
  because clearly if we had a routine to compute the Jacobian we wouldn't
  need to use finite differences.
*/
FormJacobian(snes, x, &J, &J, &user);

/*
   Color the matrix, i.e. determine groups of columns that share no common
  rows. These columns in the Jacobian can all be computed simultaneously.
*/
MatColoringCreate(J, &coloring);
MatColoringSetType(coloring, MATCOLORINGSL);
MatColoringSetFromOptions(coloring);
MatColoringApply(coloring, &iscoloring);
MatColoringDestroy(&coloring);
/*
   Create the data structure that SNESComputeJacobianDefaultColor() uses
   to compute the actual Jacobians via finite differences.
*/
MatFDColoringCreate(J, iscoloring, &fdcoloring);
ISColoringDestroy(&iscoloring);
MatFDColoringSetFunction(fdcoloring, (MatFDColoringFn *)FormFunction, &user);
MatFDColoringSetFromOptions(fdcoloring);

/*
  Tell SNES to use the routine SNESComputeJacobianDefaultColor()
  to compute Jacobians.
*/
SNESSetJacobian(snes, J, J, SNESComputeJacobianDefaultColor, fdcoloring);
```

Of course, we are cheating a bit. If we do not have an analytic formula for computing the Jacobian, then how do we know what its nonzero structure is so that it may be colored? Determining the structure is problem dependent, but fortunately, for most structured grid problems (the class of problems for which PETSc was originally designed) if one knows the stencil used for the nonlinear function one can usually fairly easily obtain an estimate of the location of nonzeros in the matrix. This is harder in the unstructured case, but one typically knows where the nonzero entries are from the mesh topology and distribution of degrees of freedom. If using `DMPlex` (*DMPlex: Unstructured Grids*) for unstructured meshes, the nonzero locations will be identified in `DMCreateMatrix()` and the procedure above can be used. Most external packages for unstructured meshes have similar functionality.

One need not necessarily use a `MatColoring` object to determine a coloring. For example, if a grid can be colored directly (without using the associated matrix), then that coloring can be provided to `MatFDColoringCreate()`. Note that the user must always preset the nonzero structure in the matrix regardless of which coloring routine is used.

PETSc provides the following coloring algorithms, which can be selected using `MatColoringSetType()` or via the command line argument `-mat_coloring_type`.

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

As for the matrix-free computation of Jacobians (*Matrix-Free Methods*), two parameters affect the accuracy of the finite difference Jacobian approximation. These are set with the command

```
MatFDColoringSetParameters(MatFDColoring fdcoloring, PetscReal rerror, PetscReal␣
↪umin);
```

The parameter `rerror` is the square root of the relative error in the function evaluations, e_{rel}; the default is the square root of machine epsilon (about 10^{-8} in double precision), which assumes that the functions are evaluated approximately to floating-point precision accuracy. The second parameter, `umin`, is a bit more involved; its default is 10^{-6}. Column i of the Jacobian matrix (denoted by $F_{:i}$) is approximated by the formula

$$F'_{:i} \approx \frac{F(u + h * dx_i) - F(u)}{h}$$

where h is computed via:

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

for `MATMFFD_DS` or:

$$h = e_{\text{rel}} \sqrt{\|u\|}$$

for `MATMFFD_WP` (default). These parameters may be set from the options database with

```
-mat_fd_coloring_err err
-mat_fd_coloring_umin umin
-mat_fd_type htype
```

Note that `MatColoring` type `MATCOLORINGSL`, `MATCOLORINGLF`, and `MATCOLORINGID` are sequential algorithms. `MATCOLORINGJP` and `MATCOLORINGGREEDY` are parallel algorithms, although in practice they may create more colors than the sequential algorithms. If one computes the coloring `iscoloring` reasonably with a parallel algorithm or by knowledge of the discretization, the routine `MatFDColoringCreate()` is scalable. An example of this for 2D distributed arrays is given below that uses the utility routine `DMCreateColoring()`.

```
DMCreateColoring(dm, 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.

2.5.8 Variational Inequalities

SNES can also solve (differential) variational inequalities with box (bound) constraints. These are nonlinear algebraic systems with additional inequality constraints on some or all of the variables: $L_i \leq u_i \leq H_i$. For example, the pressure variable cannot be negative. 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 commands

```
SNESVISetVariableBounds(SNES snes, Vec L, Vec H);
SNESVISetComputeVariableBounds(SNES snes, PetscErrorCode (*compute)(SNES, Vec, Vec))
```

are used to indicate that one is solving a variational inequality. Problems with box constraints can be solved with the reduced space, SNESVINEWTONRSLS, and semi-smooth SNESVINEWTONSSLS solvers.

Reduced space methods are also known as active set methods to capture the idea that at each Newton step a linear problem on a reduced space (the active set of variables) is solved to produce an update. See SNESVINEWTONRSLS for a concise definition of the (in)active set used by the algorithms.

The options -snes_vi_monitor, -snes_vi_monitor_residual, and -snes_vi_monitor_active turn on extra monitoring of the active set associated with the bounds.

The option -snes_vi_type allows selecting from several VI solvers, the default is preferred.

SNESLineSearchSetPreCheck() and SNESLineSearchSetPostCheck() can also be used to control properties of the steps selected by SNES.

2.5.9 Nonlinear Preconditioning

The mathematical framework of nonlinear preconditioning is explained in detail in [BKST15]. Nonlinear preconditioning in PETSc involves the use of an inner SNES instance to define the step for an outer SNES instance. The inner instance may be extracted using

```
SNESGetNPC(SNES snes, SNES *npc);
```

and passed run-time options using the -npc_ prefix. Nonlinear preconditioning comes in two flavors: left and right. The side may be changed using -snes_npc_side or SNESSetNPCSide(). Left nonlinear preconditioning redefines the nonlinear function as the action of the nonlinear preconditioner $\mathbf{M}$;

$$\mathbf{F}_M(x) = \mathbf{M}(\mathbf{x}, \mathbf{b}) - \mathbf{x}.$$

Right nonlinear preconditioning redefines the nonlinear function as the function on the action of the nonlinear preconditioner;

$$\mathbf{F}(\mathbf{M}(\mathbf{x}, \mathbf{b})) = \mathbf{b},$$

which can be interpreted as putting the preconditioner into "striking distance" of the solution by outer acceleration.

In addition, basic patterns of solver composition are available with the SNESType SNESCOMPOSITE. This allows for two or more SNES instances to be combined additively or multiplicatively. By command line, a set of SNES types may be given by comma separated list argument to -snes_composite_sneses. There are additive (SNES_COMPOSITE_ADDITIVE), additive with optimal damping (SNES_COMPOSITE_ADDITIVEOPTIMAL), and multiplicative (SNES_COMPOSITE_MULTIPLICATIVE) variants which may be set with

```
SNESCompositeSetType(SNES, SNESCompositeType);
```

New subsolvers may be added to the composite solver with

```
SNESCompositeAddSNES(SNES, SNESType);
```

and accessed with

```
SNESCompositeGetSNES(SNES, PetscInt, SNES *);
```

2.6 TS: Scalable ODE and DAE Solvers

The TS library provides a framework for the scalable solution of ODEs and DAEs arising from the discretization of time-dependent PDEs.

Simple Example: Consider the PDE

$$u_t = u_{xx}$$

discretized with centered finite differences in space yielding the semi-discrete equation

$$\begin{aligned}(u_i)_t = &\quad \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2},\\ u_t = &\quad \tilde{A}u;\end{aligned}$$

or with piecewise linear finite elements approximation in space $u(x,t) \doteq \sum_i \xi_i(t)\phi_i(x)$ yielding the semi-discrete equation

$$B\xi'(t) = A\xi(t)$$

Now applying the backward Euler method results in

$$(B - dt^n A)u^{n+1} = Bu^n,$$

in which

$$u^n{}_i = \xi_i(t_n) \doteq u(x_i, t_n),$$

$$\xi'(t_{n+1}) \doteq \frac{u^{n+1}{}_i - u^n{}_i}{dt^n},$$

A is the stiffness matrix, and B is the identity for finite differences or the mass matrix for the finite element method.

The PETSc interface for solving time dependent problems assumes the problem is written in the form

$$F(t, u, \dot{u}) = G(t, u), \quad u(t_0) = u_0.$$

In general, this is a differential algebraic equation (DAE)[1]. 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 $F(t, u, \dot{u})$

[1] If the matrix $F_{\dot{u}}(t) = \partial F/\partial \dot{u}$ is nonsingular then it is an ODE and can be transformed to the standard explicit form, although this transformation may not lead to efficient algorithms.

```
TSSetIFunction(TS ts, Vec R, PetscErrorCode (*f)(TS, PetscReal, Vec, Vec, Vec,␣
↪PetscCtx), PetscCtxfunP);
```

The vector `R` is an optional location to store the residual. The arguments to the function `f()` are the timestep context, current time, input state u, input time derivative $\dot{u}$, and the (optional) user-provided context `funP`. If $F(t, u, \dot{u}) = \dot{u}$ then one need not call this function.

- Function $G(t, u)$, if it is nonzero, is provided with the function

```
TSSetRHSFunction(TS ts, Vec R, PetscErrorCode (*f)(TS, PetscReal, Vec, Vec,␣
↪PetscCtx), PetscCtxfunP);
```

- Jacobian

  $\sigma F_{\dot{u}}(t^n, u^n, \dot{u}^n) + F_u(t^n, u^n, \dot{u}^n)$

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

  $F()$

  .

```
TSSetIJacobian(TS ts, Mat A, Mat B, PetscErrorCode (*fjac)(TS, PetscReal, Vec,␣
↪Vec, PetscReal, Mat, Mat, PetscCtx), PetscCtx jacP);
```

  The arguments for the function `fjac()` are the timestep context, current time, input state u, input derivative $\dot{u}$, input shift σ, matrix A, matrix used to construct the preconditioner B, and the (optional) user-provided context `jacP`.

  The Jacobian needed for the nonlinear system is, by the chain rule,

$$\frac{dF}{du^n} = \frac{\partial F}{\partial \dot{u}}|_{u^n} \frac{\partial \dot{u}}{\partial u}|_{u^n} + \frac{\partial F}{\partial u}|_{u^n}.$$

  For any ODE integration method the approximation of $\dot{u}$ is linear in u^n hence $\frac{\partial \dot{u}}{\partial u}|_{u^n} = \sigma$, where the shift σ depends on the ODE integrator and time step but not on the function being integrated. Thus

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

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

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

  where w is some linear combination of previous time solutions of u so that

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

  again by the chain rule.

  For example, consider backward Euler's method applied to the ODE $F(t, u, \dot{u}) = \dot{u} - f(t, u)$ with $\dot{u} = (u^n - u^{n-1})/\delta t$ and $\frac{\partial \dot{u}}{\partial u}|_{u^n} = 1/\delta t$ resulting in

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

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

- Jacobian

  G_u

  If using a fully implicit method and the function

  $G()$

  is provided, one also can provide an appropriate (approximate) Jacobian matrix of

  $G()$.

```
TSSetRHSJacobian(TS ts, Mat A, Mat B,
PetscErrorCode (*fjac)(TS, PetscReal, Vec, Mat, Mat, PetscCtx), PetscCtx jacP);
```

  The arguments for the function `fjac()` are the timestep context, current time, input state u, matrix A, matrix used to construct the preconditioner B, and the (optional) user-provided context `jacP`.

Providing appropriate $F()$ and $G()$ for your problem allows for the easy runtime switching between explicit, semi-implicit (IMEX), and fully implicit methods.

2.6.1 Basic TS Options

The user first creates a `TS` object with the command

```
int TSCreate(MPI_Comm comm,TS *ts);
```

```
int TSSetProblemType(TS ts,TSProblemType problemtype);
```

The `TSProblemType` is one of `TS_LINEAR` or `TS_NONLINEAR`.

To set up `TS` for solving an ODE, one must set the "initial conditions" for the ODE with

```
TSSetSolution(TS ts, Vec initialsolution);
```

One can set the solution method with the routine

```
TSSetType(TS ts,TSType type);
```

Some of the currently supported types are `TSEULER`, `TSRK` (Runge-Kutta), `TSBEULER`, `TSCN` (Crank-Nicolson), `TSTHETA`, `TSGLLE` (generalized linear), and `TSPSEUDO`. They can also be set with the options database option `-ts_type euler, rk, beuler, cn, theta, gl, pseudo, sundials, eimex, arkimex, rosw`. A list of available methods is given in `integrator_table`.

Set the initial time with the command

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

One can change the timestep with the command

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

can determine the current timestep with the routine

```
TSGetTimeStep(TS ts,PetscReal* dt);
```

Here, "current" refers to the timestep being used to attempt to promote the solution form u^n to u^{n+1}.

One sets the total number of timesteps to run or the total time to run (whatever is first) with the commands

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

and determines the behavior near the final time with

```
TSSetExactFinalTime(TS ts,TSExactFinalTimeOption eftopt);
```

where `eftopt` is one of `TS_EXACTFINALTIME_STEPOVER`,`TS_EXACTFINALTIME_INTERPOLATE`, or `TS_EXACTFINALTIME_MATCHSTEP`. One performs the requested number of time steps with

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

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

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

One destroys the context with

```
TSDestroy(TS *ts);
```

and views it with

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

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

```
TSStep(TS ts);
```

2.6.2 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

$$\begin{aligned}\dot{u} &= f(t,u,z)\\ 0 &= h(t,u)\end{aligned}$$

Notice that the constraint equation h is not a function of the constraint variable 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,

$$\begin{aligned}0 &= \dot{h}(t,u)\\ &= \frac{dh}{du}\dot{u} + \frac{\partial h}{\partial t}\\ &= \frac{dh}{du}f(t,u,z) + \frac{\partial h}{\partial t}\end{aligned}$$

If the Jacobian $\frac{dh}{du}\frac{df}{dz}$ is non-singular, then we have precisely a semi-explicit index-1 DAE, and we can once again use the PETSc IMEX tools to solve it. A common example of an index-2 DAE is the incompressible Navier-Stokes equations, since the continuity equation $\nabla \cdot u = 0$ does not involve the pressure. Using PETSc IMEX with the above conversion then corresponds to the Segregated Runge-Kutta method applied to this equation [ColomesB16].

2.6.3 Using Implicit-Explicit (IMEX) Methods

For "stiff" problems or those with multiple time scales $F()$ will be treated implicitly using a method suitable for stiff problems and $G()$ will be treated explicitly when using an IMEX method like TSARKIMEX. $F()$ is typically linear or weakly nonlinear while $G()$ may have very strong nonlinearities such as arise in non-oscillatory methods for hyperbolic PDE. The user provides three pieces of information, the APIs for which have been described above.

- "Slow" part $G(t,u)$ using `TSSetRHSFunction()`.
- "Stiff" part $F(t,u,\dot{u})$ using `TSSetIFunction()`.
- Jacobian $F_u + \sigma F_{\dot{u}}$ using `TSSetIJacobian()`.

The user needs to set `TSSetEquationType()` to `TS_EQ_IMPLICIT` or higher if the problem is implicit; e.g., $F(t,u,\dot{u}) = M\dot{u} - f(t,u)$, where M is not the identity matrix:

- the problem is an implicit ODE (defined implicitly through `TSSetIFunction()`) or
- a DAE is being solved.

An IMEX problem representation can be made implicit by setting `TSARKIMEXSetFullyImplicit()`. Note that multilevel preconditioners (e.g. `PCMG`), won't work in the fully implicit case; the same holds true for any other `TS` type requiring a fully implicit formulation in case both Jacobians are specified.

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 2.12 and a list of methods with a summary of their properties is given in *IMEX Runge-Kutta schemes*.

Table 2.12: 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	$F(t,u,\dot{u}) = M\dot{u} - f(t,u)$ $G(t,u) = M^{-1}g(t,u)$
$\dot{u} = f(t,u,z) + g(t,u,z)$ $0 = h(t,y,z)$	semi-explicit index-1 DAE	$F(t,u,\dot{u}) = \begin{pmatrix} \dot{u} - f(t,u,z) \\ h(t,u,z) \end{pmatrix}$ $G(t,u) = g(t,u)$
$f(t,u,\dot{u}) = 0$	fully implicit ODE/DAE	$F(t,u,\dot{u}) = f(t,u,\dot{u})$ $G(t,u) = 0$; the user needs to set TSSetEquationType() to TS_EQ_IMPLICIT or higher

Table 2.13 lists of the currently available IMEX Runge-Kutta schemes. For each method, it gives the -ts_arkimex_type name, the reference, the total number of stages/implicit stages, the order/stage-order, the implicit stability properties (IM), stiff accuracy (SA), the existence of an embedded scheme, and dense output (DO).

Table 2.13: IMEX Runge-Kutta schemes

Name	Reference	Stages (IM)	Order (Stage)	IM	SA	Embed	DO	Remarks
a2	based on CN	2 (1)	2 (2)	A-Stable	yes	yes (1)	yes (2)	
l2	SSP2(2,2,2) [PR05]	2 (2)	2 (1)	L-Stable	yes	yes (1)	yes (2)	SSP SDIRK
ars122	ARS122 [ARS97]	2 (1)	3 (1)	A-Stable	yes	yes (1)	yes (2)	
2c	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
2d	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
2e	[GKC13]	3 (2)	2 (2)	L-Stable	yes	yes (1)	yes (2)	SDIRK
prssp2	PRS(3,3,2) [PR05]	3 (3)	3 (1)	L-Stable	yes	no	no	SSP
3	[KC03]	4 (3)	3 (2)	L-Stable	yes	yes (2)	yes (2)	SDIRK
bpr3	[BPR11]	5 (4)	3 (2)	L-Stable	yes	no	no	SDIRK
ars443	[ARS97]	5 (4)	3 (1)	L-Stable	yes	no	no	SDIRK
4	[KC03]	6 (5)	4 (2)	L-Stable	yes	yes (3)	yes	SDIRK
5	[KC03]	8 (7)	5 (2)	L-Stable	yes	yes (4)	yes (3)	SDIRK

ROSW are linearized implicit Runge-Kutta methods known as Rosenbrock W-methods. They can accommodate inexact Jacobian matrices in their formulation. A series of methods are available in PETSc are listed in Table 2.14 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 2.14: Rosenbrock W-schemes

TS	Reference	Stages (IM)	Order (Stage)	IM	SA	Embed	DO	-W	PDAE	Remarks
theta1	classical	1(1)	1(1)	L-Stable	•	•	•	•	•	•
theta2	classical	1(1)	2(2)	A-Stable	•	•	•	•	•	•
2m	Zoltan	2(2)	2(1)	L-Stable	No	Yes(1)	Yes(2)	Yes	No	SSP
2p	Zoltan	2(2)	2(1)	L-Stable	No	Yes(1)	Yes(2)	Yes	No	SSP
ra3pw	[RA05]	3(3)	3(1)	A-Stable	No	Yes	Yes(2)	No	Yes(3)	•
ra34pw2	[RA05]	4(4)	3(1)	L-Stable	Yes	Yes	Yes(3)	Yes	Yes(3)	•
rodas3	[SVB+9'	4(4)	3(1)	L-Stable	Yes	Yes	No	No	Yes	•
sandu3	[SVB+9'	3(3)	3(1)	L-Stable	Yes	Yes	Yes(2)	No	No	•
assp3p3s	unpub.	3(2)	3(1)	A-Stable	No	Yes	Yes(2)	Yes	No	SSP
lassp3p4	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	SSP
lassp3p4	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	SSP
ark3	unpub.	4(3)	3(1)	L-Stable	No	Yes	Yes(3)	Yes	No	IMEX-RK

2.6.4 IMEX Methods for fast-slow systems

Consider a fast-slow ODE system

$$\dot{u}^{slow} = f^{slow}(t, u^{slow}, u^{fast})$$
$$M\dot{u}^{fast} = g^{fast}(t, u^{slow}, u^{fast}) + f^{fast}(t, u^{slow}, u^{fast})$$

where u^{slow} is the slow component and u^{fast} is the fast component. The fast component can be partitioned additively as described above. Thus we want to treat $f^{slow}()$ and $f^{fast}()$ explicitly and the other terms implicitly when using TSARKIMEX. This is achieved by using the following APIs:

- `TSARKIMEXSetFastSlowSplit()` informs PETSc to use ARKIMEX to solve a fast-slow system.
- `TSRHSSplitSetIS()` specifies the index set for the slow/fast components.
- `TSRHSSplitSetRHSFunction()` specifies the parts to be handled explicitly $f^{slow}()$ and $f^{fast}()$.
- `TSRHSSplitSetIFunction()` and `TSRHSSplitSetIJacobian()` specify the implicit part and its Jacobian.

Note that this ODE system can also be solved by padding zeros in the implicit part and using the standard IMEX methods. However, one needs to provide the full-dimensional Jacobian whereas only a partial Jacobian is needed for the fast-slow split which is more efficient in storage and speed.

2.6.5 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 ε 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),
- γ: factor representing the global error ratio,
- A, U, B, V: method coefficients,
- S: starting method to compute the working vector from the solution (say at the beginning of time integration) so that $Y = Sy$,
- F: finalizing method to compute the solution from the working vector,$y = FY$.
- F_{embed}: coefficients for computing the auxiliary solution $\tilde{y}$ from the working vector ($\tilde{y} = F_{\text{embed}}Y$),
- F_{error}: coefficients to compute the estimated error vector from the working vector ($\varepsilon = F_{\text{error}}Y$).
- S_{error}: coefficients to initialize the auxiliary solution ($\tilde{y}$ or ε) from a specified error vector (ε). It is currently implemented only for $r = 2$. We have $y_{\text{aux}} = S_{error}[0] * \varepsilon + S_{\text{error}}[1] * y$, where y_{aux} is the 2nd component of the working vector Y.

The methods can be described in two mathematically equivalent forms: propagate two components ("$y\tilde{y}$ form") and propagating the solution and its estimated error ("$y\varepsilon$ form"). The two forms are not explicitly specified in `TSGLEE`; rather, the specific values of B, U, S, F, F_{embed}, and F_{error} characterize whether the method is in $y\tilde{y}$ or $y\varepsilon$ form.

The API used by this `TS` method includes:

- `TSGetSolutionComponents`: Get all the solution components of the working vector

```
PetscCall(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 $F_{\text{embed}}Y$)

```
PetscCall(TSGetAuxSolution(TS, Vec*))
```

- `TSGetTimeError`: Returns the estimated error vector ε (computed as $F_{\text{error}}Y$ if $n = 0$ or restores the error estimate at the end of the previous step if $n = -1$)

```
PetscCall(TSGetTimeError(TS, PetscInt n, Vec*))
```

- `TSSetTimeError`: Initializes the auxiliary solution ($\tilde{y}$ or ε) for a specified initial error.

```
PetscCall(TSSetTimeError(TS, Vec))
```

The local error is estimated as $\varepsilon(n+1) - \varepsilon(n)$. This is to be used in the error control. The error in $y\tilde{y}$ GLEE is $\varepsilon(n) = \frac{1}{1-\gamma} * (\tilde{y}(n) - y(n))$.

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

Table 2.15 lists currently available GL schemes with global error estimation [Con16].

Table 2.15: GL schemes with global error estimation

TS	Reference	IM/EX	(p, r, s)	γ	Form	Notes
TSGLEEi1	BE1	IM	$(1, 3, 2)$	0.5	$y\varepsilon$	Based on backward Euler
TSGLEE23	23	EX	$(2, 3, 2)$	0	$y\varepsilon$	
TSGLEE24	24	EX	$(2, 4, 2)$	0	$y\tilde{y}$	
TSGLEE25I	25i	EX	$(2, 5, 2)$	0	$y\tilde{y}$	
TSGLEE35	35	EX	$(3, 5, 2)$	0	$y\tilde{y}$	
TSGLEEEXRK2A	exrk2a	EX	$(2, 6, 2)$	0.25	$y\varepsilon$	
TSGLEERK32G1	rk32g1	EX	$(3, 8, 2)$	0	$y\varepsilon$	
TSGLEERK285EX	rk285ex	EX	$(2, 9, 2)$	0.25	$y\varepsilon$	

2.6.6 Using fully implicit methods

To use a fully implicit method like TSTHETA, TSBDF or TSDIRK, 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.

2.6.7 Using the Explicit Runge-Kutta timestepper with variable timesteps

The explicit Euler and Runge-Kutta methods require the ODE be in the form

$$\dot{u} = G(u, t).$$

The user can either call TSSetRHSFunction() and/or they can call TSSetIFunction() (so long as the function provided to TSSetIFunction() is equivalent to $\dot{u} + \tilde{F}(t, u)$) but the Jacobians need not be provided.[2]

The Explicit Runge-Kutta timestepper with variable timesteps is an implementation of the standard Runge-Kutta with an embedded method. The error in each timestep is calculated using the solutions from the Runge-Kutta method and its embedded method (the 2-norm of the difference is used). The default method is the 3rd-order Bogacki-Shampine method with a 2nd-order embedded method (TSRK3BS). Other available methods are the 5th-order Fehlberg RK scheme with a 4th-order embedded method (TSRK5F), the 5th-order Dormand-Prince RK scheme with a 4th-order embedded method (TSRK5DP), the 5th-order Bogacki-Shampine RK scheme with a 4th-order embedded method (TSRK5BS, and the 6th-, 7th, and 8th-order robust Verner RK schemes with a 5th-, 6th, and 7th-order embedded method, respectively (TSRK6VR, TSRK7VR, TSRK8VR). Variable timesteps cannot be used with RK schemes that do not have an embedded method (TSRK1FE - 1st-order, 1-stage forward Euler, TSRK2A - 2nd-order, 2-stage RK scheme, TSRK3 - 3rd-order, 3-stage RK scheme, TSRK4 - 4-th order, 4-stage RK scheme).

[2] PETSc will automatically translate the function provided to the appropriate form.

2.6.8 Special Cases

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

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

  or

```
TSSetProblemType(ts,TS_LINEAR);
TSSetIFunction(ts,NULL,TSComputeIFunctionLinear,NULL);
TSSetIJacobian(ts,A,A,TSComputeIJacobianConstant,NULL);
```

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

```
TSSetProblemType(ts,TS_LINEAR);
TSSetRHSFunction(ts,NULL,TSComputeRHSFunctionLinear,NULL);
TSSetRHSJacobian(ts,A,A,YourComputeRHSJacobian, &appctx);
```

  where `YourComputeRHSJacobian()` is a function you provide that computes A as a function of time. Or use

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

2.6.9 Monitoring and visualizing solutions

- `-ts_monitor` - prints the time and timestep at each iteration.
- `-ts_adapt_monitor` - prints information about the timestep adaption calculation at each iteration.
- `-ts_monitor_lg_timestep` - plots the size of each timestep, `TSMonitorLGTimeStep()`.
- `-ts_monitor_lg_solution` - for ODEs with only a few components (not arising from the discretization of a PDE) plots the solution as a function of time, `TSMonitorLGSolution()`.
- `-ts_monitor_lg_error` - for ODEs with only a few components plots the error as a function of time, only if `TSSetSolutionFunction()` is provided, `TSMonitorLGError()`.
- `-ts_monitor_draw_solution` - plots the solution at each iteration, `TSMonitorDrawSolution()`.
- `-ts_monitor_draw_error` - plots the error at each iteration only if `TSSetSolutionFunction()` is provided, `TSMonitorDrawSolution()`.
- `-ts_monitor_solution binary[:filename]` - saves the solution at each iteration to a binary file, `TSMonitorSolution()`. Solution viewers work with other time-aware formats, e.g., `-ts_monitor_solution cgns:sol.cgns`, and can output one solution every 10 time steps by adding `-ts_monitor_solution_interval 10`. Use `-ts_monitor_solution_interval -1` to output data only at then end of a time loop.
- `-ts_monitor_solution_vtk <filename-%03D.vts>` - saves the solution at each iteration to a file in vtk format, `TSMonitorSolutionVTK()`.

2.6.10 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 2.16: `TSAdapt`: available adaptors

ID	Nam	Notes
`TSADAPTNONE`	`none`	no adaptivity
`TSADAPTBASIC`	`basic`	the default adaptor
`TSADAPTGLEE`	`glee`	extension of the basic adaptor to treat $\mathrm{Tol_A}$ and $\mathrm{Tol_R}$ as separate criteria. It can also control global errors if the integrator (e.g., `TSGLEE`) provides this information
`TSADAPTDSP`	`dsp`	adaptive controller for time-stepping based on digital signal processing

When using `TSADAPTBASIC` (the default), the user typically provides a desired absolute $\mathrm{Tol_A}$ or a relative $\mathrm{Tol_R}$ error tolerance by invoking `TSSetTolerances()` or at the command line with options `-ts_atol` and `-ts_rtol`. The error estimate is based on the local truncation error, so for every step the algorithm verifies that the estimated local truncation error satisfies the tolerances provided by the user and computes a new step size to be taken. For multistage methods, the local truncation is obtained by comparing the solution y to a lower order $\widehat{p} = p - 1$ approximation, $\widehat{y}$, where p is the order of the method and $\widehat{p}$ the order of $\widehat{y}$.

The adaptive controller at step n computes a tolerance level

$$Tol_n(i) = \mathrm{Tol_A}(i) + \max(y_n(i), \widehat{y}_n(i))\mathrm{Tol_R}(i)\,,$$

and forms the acceptable error level

$$\mathrm{wlte_n} = \frac{1}{m}\sum_{i=1}^{m}\sqrt{\frac{\|y_n(i) - \widehat{y}_n(i)\|}{Tol(i)}}\,,$$

where the errors are computed componentwise, m is the dimension of y and `-ts_adapt_wnormtype` is `2` (default). If `-ts_adapt_wnormtype` is `infinity` (max norm), then

$$\mathrm{wlte_n} = \max_{1\ldots m}\frac{\|y_n(i) - \widehat{y}_n(i)\|}{Tol(i)}\,.$$

The error tolerances are satisfied when $\mathrm{wlte} \le 1.0$.

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

$$\Delta t_{\mathrm{new}}(t) = \Delta t_{\mathrm{old}}\min(\alpha_{\max}, \max(\alpha_{\min}, \beta(1/\mathrm{wlte})^{\frac{1}{\widehat{p}+1}}))\,, \tag{2.5}$$

where $\alpha_{\min}$ =`-ts_adapt_clip`[0] and $\alpha_{\max}$=`-ts_adapt_clip`[1] keep the change in Δt to within a certain factor, and $\beta < 1$ is chosen through `-ts_adapt_safety` so that there is some margin to which the tolerances are satisfied and so that the probability of rejection is decreased.

This adaptive controller works in the following way. After completing step k, if $\mathrm{wlte_{k+1}} \le 1.0$, then the step is accepted and the next step is modified according to (2.5); otherwise, the step is rejected and retaken with the step length computed in (2.5).

2.6.11 Handling of discontinuities

For problems that involve discontinuous right-hand sides, one can set an "event" function $g(t, u)$ for PETSc to detect and locate the times of discontinuities (zeros of $g(t, u)$). Events can be defined through the event monitoring routine

```
TSSetEventHandler(TS ts,PetscInt nevents,PetscInt *direction,PetscBool *terminate,
↪PetscErrorCode (*indicator)(TS,PetscReal,Vec,PetscScalar*,PetscCtx eventP),
↪PetscErrorCode (*postevent)(TS,PetscInt,PetscInt[],PetscReal,Vec,PetscBool,PetscCtx␣
↪eventP),PetscCtxeventP);
```

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 `indicator()` 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 occurred, indices of events occurred, current time, input state u, a boolean flag indicating forward solve (1) or adjoint solve (0), and the (optional) user-provided context `eventP`.

2.6.12 Explicit integrators with finite element mass matrices

Discretized finite element problems often have the form $M\dot{u} = G(t, u)$ where M is the mass matrix. Such problems can be solved using `DMTSSetIFunction()` with implicit integrators. When M is nonsingular (i.e., the problem is an ODE, not a DAE), explicit integrators can be applied to $\dot{u} = M^{-1}G(t, u)$ or $\dot{u} = \hat{M}^{-1}G(t, u)$, where $\hat{M}$ is the lumped mass matrix. While the true mass matrix generally has a dense inverse and thus must be solved iteratively, the lumped mass matrix is diagonal (e.g., computed via collocated quadrature or row sums of M). To have PETSc create and apply a (lumped) mass matrix automatically, first use `DMTSSetRHSFunction()` to specify G and set a `PetscFE` using `DMAddField()` and `DMCreateDS()`, then call either `DMTSCreateRHSMassMatrix()` or `DMTSCreateRHSMassMatrixLumped()` to automatically create the mass matrix and a `KSP` that will be used to apply M^{-1}. This `KSP` can be customized using the `"mass_"` prefix.

2.6.13 Performing sensitivity analysis with the TS ODE Solvers

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`.

Using the discrete adjoint methods

Consider the ODE/DAE

$$F(t, y, \dot{y}, p) = 0, \quad y(t_0) = y_0(p) \quad t_0 \le t \le t_F$$

and the cost function(s)

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

The `TSAdjoint` routines of PETSc provide

$$\frac{\partial \Psi_i}{\partial y_0} = \lambda_i$$

and

$$\frac{\partial \Psi_i}{\partial p} = \mu_i + \lambda_i(\frac{\partial y_0}{\partial p}).$$

To perform the discrete adjoint sensitivity analysis one first sets up the `TS` object for a regular forward run but with one extra function call

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

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

One must create two arrays of n_{cost} vectors λ and μ (if there are no parameters p then one can use `NULL` for the μ array.) The λ vectors are the same dimension and parallel layout as the solution vector for the ODE, the μ vectors are of dimension p; when p is small usually all its elements are on the first MPI process, while the vectors have no entries on the other processes. λ_i and μ_i should be initialized with the values $d\Phi_i/dy|_{t=t_F}$ and $d\Phi_i/dp|_{t=t_F}$ respectively. Then one calls

```
TSSetCostGradients(TS ts,PetscInt numcost, Vec *lambda,Vec *mu);
```

where `numcost` denotes n_{cost}. If $F()$ is a function of p one needs to also provide the Jacobian $-F_p$ with

```
TSSetRHSJacobianP(TS ts,Mat Amat,PetscErrorCode (*fp)(TS,PetscReal,Vec,Mat,PetscCtx),
↪PetscCtx ctx)
```

or

```
TSSetIJacobianP(TS ts,Mat Amat,PetscErrorCode (*fp)(TS,PetscReal,Vec,Vec,PetscReal,
↪Mat,PetscCtx),PetscCtx ctx)
```

or both, depending on which form is used to define the ODE.

The arguments for the function `fp()` are the timestep context, current time, y, and the (optional) user-provided context.

If there is an integral term in the cost function, i.e. r is nonzero, it can be transformed into another ODE that is augmented to the original ODE. To evaluate the integral, one needs to create a child `TS` objective by calling

```
TSCreateQuadratureTS(TS ts,PetscBool fwd,TS *quadts);
```

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

```
TSSetRHSFunction(TS quadts,Vec R,PetscErrorCode (*rf)(TS,PetscReal,Vec,Vec,PetscCtx),
↪PetscCtxctx)
```

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

```
TSSetRHSJacobian(TS quadts,Vec DRDU,Vec DRDU,PetscErrorCode (*drdyf)(TS,PetscReal,Vec,
↪Vec*,PetscCtx),PetscCtxctx)
TSSetRHSJacobianP(TS quadts,Vec DRDU,Vec DRDU,PetscErrorCode (*drdyp)(TS,PetscReal,
↪Vec,Vec*,PetscCtx),PetscCtxctx)
```

where drdyf $= dr/dy$, drdpf $= dr/dp$. Since the integral term is additive to the cost function, its gradient information will be included in λ and μ.

Lastly, one starts the backward run by calling

```
TSAdjointSolve(TS ts).
```

One can obtain the value of the integral term by calling

```
TSGetCostIntegral(TS ts,Vec *q).
```

or accessing directly the solution vector used by `quadts`.

The second argument of `TSCreateQuadratureTS()` allows one to choose if the integral term is evaluated in the forward run (inside `TSSolve()`) or in the backward run (inside `TSAdjointSolve()`) when `TSSetCostGradients()` and `TSSetCostIntegrand()` are called before `TSSolve()`. Note that this also allows for evaluating the integral without having to use the adjoint solvers.

To provide a better understanding of the use of the adjoint solvers, we introduce a simple example, corresponding to TS Power Grid Tutorial ex3sa. The problem is to study dynamic security of power system when there are credible contingencies such as short-circuits or loss of generators, transmission lines, or loads. The dynamic security constraints are incorporated as equality constraints in the form of discretized differential equations and inequality constraints for bounds on the trajectory. The governing ODE system is

$$\begin{aligned} \phi' = & \quad \omega_B(\omega - \omega_S) \\ 2H/\omega_S\,\omega' = & \quad p_m - p_{max} sin(\phi) - D(\omega - \omega_S), \quad t_0 \le t \le t_F, \end{aligned}$$

where ϕ is the phase angle and ω is the frequency.

The initial conditions at time t_0 are

$$\begin{aligned} \phi(t_0) = & \quad \arcsin\left(p_m/p_{max}\right), \\ w(t_0) = & \quad 1. \end{aligned}$$

p_{max} is a positive number when the system operates normally. At an event such as fault incidence/removal, p_{max} will change to 0 temporarily and back to the original value after the fault is fixed. The objective is to maximize p_m subject to the above ODE constraints and $\phi < \phi_S$ during all times. To accommodate the inequality constraint, we want to compute the sensitivity of the cost function

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

with respect to the parameter p_m. *numcost* is 1 since it is a scalar function.

For ODE solution, PETSc requires user-provided functions to evaluate the system $F(t, y, \dot{y}, p)$ (set by `TSSetIFunction()`) and its corresponding Jacobian $F_y + \sigma F_{\dot{y}}$ (set by `TSSetIJacobian()`). Note that the solution state y is $[\phi\ \omega]^T$ here. For sensitivity analysis, we need to provide a routine to compute $f_p = [0\ 1]^T$ using `TSASetRHSJacobianP()`, and three routines corresponding to the integrand $r = c\left(\max(0, \phi - \phi_S)\right)^2$, $r_p = [0\ 0]^T$ and $r_y = [2c\left(\max(0, \phi - \phi_S)\right)\ 0]^T$ using `TSSetCostIntegrand()`.

In the adjoint run, λ and μ are initialized as $[0\ 0]^T$ and $[-1]$ at the final time t_F. After `TSAdjointSolve()`, the sensitivity of the cost function w.r.t. initial conditions is given by the sensitivity variable λ (at time t_0) directly. And the sensitivity of the cost function w.r.t. the parameter p_m can be computed (by users) as

$$\frac{\mathrm{d}\Psi}{\mathrm{d}p_m} = \mu(t_0) + \lambda(t_0)\frac{\mathrm{d}\left[\phi(t_0)\ \omega(t_0)\right]^T}{\mathrm{d}p_m}.$$

For explicit methods where one does not need to provide the Jacobian F_u for the forward solve one still does need it for the backward solve and thus must call

```
TSSetRHSJacobian(TS ts,Mat Amat, Mat Pmat,PetscErrorCode (*f)(TS,PetscReal,Vec,Mat,
↪Mat,PetscCtx),PetscCtxfP);
```

Examples include:

- discrete adjoint sensitivity using explicit and implicit time stepping methods for an ODE problem TS Tutorial ex20adj,
- an optimization problem using the discrete adjoint models of the ERK (for nonstiff ODEs) and the Theta methods (for stiff DAEs) TS Tutorial ex20opt_ic and TS Tutorial ex20opt_p,
- an ODE-constrained optimization using the discrete adjoint models of the Theta methods for cost function with an integral term TS Power Grid Tutorial ex3opt,
- discrete adjoint sensitivity using the Crank-Nicolson methods for DAEs with discontinuities TS Power Grid Stability Tutorial ex9busadj,
- a DAE-constrained optimization problem using the discrete adjoint models of the Crank-Nicolson methods for cost function with an integral term TS Power Grid Tutorial ex9busopt,
- discrete adjoint sensitivity using the Crank-Nicolson methods for a PDE problem TS Advection-Diffusion-Reaction Tutorial ex5adj.

Checkpointing

The discrete adjoint model requires the states (and stage values in the context of multistage timestepping methods) to evaluate the Jacobian matrices during the adjoint (backward) run. By default, PETSc stores the whole trajectory to disk as binary files, each of which contains the information for a single time step including state, time, and stage values (optional). One can also make PETSc store the trajectory to memory with the option `-ts_trajectory_type memory`. However, there might not be sufficient memory capacity especially for large-scale problems and long-time integration.

A so-called checkpointing scheme is needed to solve this problem. The scheme stores checkpoints at selective time steps and recomputes the missing information. The `revolve` library is used by PETSc `TSTrajectory` to generate an optimal checkpointing schedule that minimizes the recomputations given a limited number of available checkpoints. One can specify the number of available checkpoints with the option `-ts_trajectory_max_cps_ram [maximum number of checkpoints in RAM]`. Note that one checkpoint corresponds to one time step.

The `revolve` library also provides an optimal multistage checkpointing scheme that uses both RAM and disk for storage. This scheme is automatically chosen if one uses both the option `-ts_trajectory_max_cps_ram [maximum number of checkpoints in RAM]` and the option `-ts_trajectory_max_cps_disk [maximum number of checkpoints on disk]`.

Some other useful options are listed below.

- `-ts_trajectory_view` prints the total number of recomputations,
- `-ts_monitor` and `-ts_adjoint_monitor` allow users to monitor the progress of the adjoint work flow,

- `-ts_trajectory_type visualization` may be used to save the whole trajectory for visualization. It stores the solution and the time, but no stage values. The binary files generated can be read into MATLAB via the script `$PETSC_DIR/share/petsc/matlab/PetscReadBinaryTrajectory.m`.

2.6.14 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, `installation/index.htm`.)

To use the Sundials integrators, call

```
TSSetType(TS ts,TSType TSSUNDIALS);
```

or use the command line option `-ts_type sundials`.

Sundials' CVODE solver comes with two main integrator families, Adams and BDF (backward differentiation formula). One can select these with

```
TSSundialsSetType(TS ts,TSSundialsLmmType [SUNDIALS_ADAMS,SUNDIALS_BDF]);
```

or the command line option `-ts_sundials_type <adams,bdf>`. BDF is the default.

Sundials does not use the `SNES` library within PETSc for its nonlinear solvers, so one cannot change the nonlinear solver options via `SNES`. Rather, Sundials uses the preconditioners within the `PC` package of PETSc, which can be accessed via

```
TSSundialsGetPC(TS ts,PC *pc);
```

The user can then directly set preconditioner options; alternatively, the usual runtime options can be employed via `-pc_xxx`.

Finally, one can set the Sundials tolerances via

```
TSSundialsSetTolerance(TS ts,double abs,double rel);
```

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

Other PETSc-Sundials options include

```
TSSundialsSetGramSchmidtType(TS ts,TSSundialsGramSchmidtType type);
```

where `type` is either `SUNDIALS_MODIFIED_GS` or `SUNDIALS_UNMODIFIED_GS`. This may be set via the options data base with `-ts_sundials_gramschmidt_type <modifed,unmodified>`.

The routine

```
TSSundialsSetMaxl(TS ts,PetscInt restart);
```

sets the number of vectors in the Krylov subpspace used by GMRES. This may be set in the options database with `-ts_sundials_maxl maxl`.

2.6.15 Using TChem from PETSc

TChem[3] is a package originally developed at Sandia National Laboratory that can read in CHEMKIN[4] 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.

2.7 Solving Steady-State Problems with Pseudo-Timestepping

Simple Example: `TS` provides a general code for performing pseudo timestepping with a variable timestep at each physical node point. For example, instead of directly attacking the steady-state problem

$$G(u) = 0,$$

we can use pseudo-transient continuation by solving

$$u_t = G(u).$$

Using time differencing

$$u_t \doteq \frac{u^{n+1} - u^n}{dt^n}$$

with the backward Euler method, we obtain nonlinear equations at a series of pseudo-timesteps

$$\frac{1}{dt^n} B(u^{n+1} - u^n) = G(u^{n+1}).$$

For this problem the user must provide $G(u)$, the time steps dt^n and the left-hand-side matrix B (or optionally, if the timestep is position independent and B is the identity matrix, a scalar timestep), as well as optionally the Jacobian of $G(u)$.

More generally, this can be applied to implicit ODE and DAE for which the transient form is

$$F(u, \dot{u}) = 0.$$

For solving steady-state problems with pseudo-timestepping one proceeds as follows.

- Provide the function `G(u)` with the routine

```
TSSetRHSFunction(TS ts,Vec r,PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,PetscCtx),
↪PetscCtxfP);
```

  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

[3] bitbucket.org/jedbrown/tchem
[4] en.wikipedia.org/wiki/CHEMKIN

```
TSSetRHSJacobian(TS ts,Mat Amat, Mat Pmat,PetscErrorCode (*f)(TS,PetscReal,Vec,
↪Mat,Mat,PetscCtx),PetscCtxfP);
```

The arguments for the function `f()` are the timestep context, the current time, the location where the Jacobian is to be computed, the (approximate) Jacobian matrix, an alternative approximate Jacobian matrix used to construct the preconditioner, and the optional user-provided context, passed in as `fP`. The user must provide the Jacobian as a matrix; thus, if using a matrix-free approach, one must create a `MATSHELL` matrix.

In addition, the user must provide a routine that computes the pseudo-timestep. This is slightly different depending on if one is using a constant timestep over the entire grid, or it varies with location.

- For location-independent pseudo-timestepping, one uses the routine

```
TSPseudoSetTimeStep(TS ts,PetscInt(*dt)(TS,PetscReal*,PetscCtx),PetscCtx dtctx);
```

The function `dt` is a user-provided function that computes the next pseudo-timestep. As a default one can use `TSPseudoTimeStepDefault(TS,PetscReal*,PetscCtx)` for `dt`. This routine updates the pseudo-timestep with one of two strategies: the default

$$dt^n = dt_{\text{increment}} * dt^{n-1} * \frac{||F(u^{n-1})||}{||F(u^n)||}$$

or, the alternative,

$$dt^n = dt_{\text{increment}} * dt^0 * \frac{||F(u^0)||}{||F(u^n)||}$$

which can be set with the call

```
TSPseudoIncrementDtFromInitialDt(TS ts);
```

or the option `-ts_pseudo_increment_dt_from_initial_dt`. The value $dt_{\text{increment}}$ is by default 1.1, but can be reset with the call

```
TSPseudoSetTimeStepIncrement(TS ts,PetscReal inc);
```

or the option `-ts_pseudo_increment <inc>`.

- For location-dependent pseudo-timestepping, the interface function has not yet been created.

2.8 TAO: Optimization Solvers

The Toolkit for Advanced Optimization (TAO) focuses on algorithms for the solution of large-scale optimization problems on high-performance architectures. Methods are available for

- *Nonlinear Least-Squares*
- *Quadratic Solvers*
- *Unconstrained Minimization*
- *Bound-Constrained Optimization*
- *Generally Constrained Solvers*
- *Complementarity*
- *PDE-constrained Optimization*

2.8.1 Getting Started: A Simple TAO Example

To help start using TAO immediately, we introduce a simple uniprocessor example. Please read *TAO Algorithms* 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 usable from C++, Fortran, and Python. *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

```
#include <petsctao.h>
typedef struct {
  MPI_Comm  comm;
  PetscInt  n;         /* dimension */
  PetscReal alpha;     /* condition parameter */
  PetscBool chained;   /* chained vs. unchained Rosenbrock function */
  PetscBool test;      /* run tests in AppCtxFinalize() */
  PetscBool jacobi_pc; /* Create Jacobi Hpre */
  PetscBool use_fd;    /* Use finite difference for grad and hess */
} AppCtx;

static PetscErrorCode AppCtxInitialize(MPI_Comm, AppCtx *); /* process options */
static PetscErrorCode AppCtxFinalize(AppCtx *, Tao);        /* clean up and␣
↪optionally run tests */
static PetscErrorCode AppCtxCreateSolution(AppCtx *, Vec *);
static PetscErrorCode AppCtxCreateHessianMatrices(AppCtx *, Mat *, Mat *);
```

```
static PetscErrorCode FormFunctionGradient(Tao, Vec, PetscReal *, Vec, void *);
static PetscErrorCode FormHessian(Tao, Vec, Mat, Mat, void *);

int main(int argc, char **argv)
{
  Vec         x; /* solution vector */
  Mat         H, Hpre;
  Tao         tao;  /* Tao solver context */
  PetscMPIInt size; /* number of processes running */
  AppCtx      user; /* user-defined application context */
  MPI_Comm    comm;

  /* Initialize TAO and PETSc */
  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));
  comm = PETSC_COMM_WORLD;
```

(continues on next page)

(continued from previous page)

```
  PetscCallMPI(MPI_Comm_size(comm, &size));
  PetscCheck(size == 1, comm, PETSC_ERR_WRONG_MPI_SIZE, "Incorrect number of␣
↪processors");

  /* Initialize problem parameters */
  PetscCall(AppCtxInitialize(comm, &user));

  /* Allocate vector for the solution */
  PetscCall(AppCtxCreateSolution(&user, &x));

  /* Allocate the Hessian matrix */
  PetscCall(AppCtxCreateHessianMatrices(&user, &H, &Hpre));

  /* The TAO code begins here */

  /* Create TAO solver with desired solution method */
  PetscCall(TaoCreate(comm, &tao));
  PetscCall(TaoSetType(tao, TAOLMVM));

  /* Set solution vec and an initial guess */
  PetscCall(VecZeroEntries(x));
  PetscCall(TaoSetSolution(tao, x));

  /* Set routines for function, gradient, hessian evaluation */
  PetscCall(TaoSetObjectiveAndGradient(tao, NULL, FormFunctionGradient, &user));
  PetscCall(TaoSetHessian(tao, H, Hpre, FormHessian, &user));

  /* Check for TAO command line options */
  PetscCall(TaoSetFromOptions(tao));

  /* SOLVE THE APPLICATION */
  PetscCall(TaoSolve(tao));

  /* Clean up */
  PetscCall(AppCtxFinalize(&user, tao));
  PetscCall(TaoDestroy(&tao));
  PetscCall(VecDestroy(&x));
  PetscCall(MatDestroy(&H));
  PetscCall(MatDestroy(&Hpre));

  PetscCall(PetscFinalize());
return 0;}
```

2.8.2 TAO Workflow

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. Callback routines as well as vector (`Vec`) and matrix (`Mat`) data structures are then set. These callback 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 shown below.

```
TaoCreate(MPI_Comm comm, Tao *tao);
TaoSetType(Tao tao, TaoType type);
TaoSetSolution(Tao tao, Vec x);
TaoSetObjectiveAndGradient(Tao tao, Vec g, PetscErrorCode (*FormFGradient)(Tao, Vec,␣
↪PetscReal*, Vec, PetscCtx), PetscCtx ctx);
TaoSetHessian(Tao tao, Mat H, Mat Hpre, PetscErrorCode (*FormHessian)(Tao, Vec, Mat,␣
↪Mat, PetscCtx), PetscCtx ctx);
TaoSolve(Tao tao);
TaoDestroy(Tao tao);
```

TAO supports constructing an objective function by summing several distinct functions (called terms) via the `TaoTerm` object. With `TaoTerm`, the user can define one or more objective function 'terms'. For an example, consider a data-misfit term and a regularization term, each providing objective, gradient, and optional Hessian routines. TAO automatically composes (sums) the terms to form the overall objective and its derivatives at runtime. This approach promotes code reuse, makes it easy to modify scaling parameters, and simplifies complex problems that are naturally expressed as sums of contributions. In addition, it allows the easy implementation of efficient optimization algorithms that utilize the sum structure of the objective function. See *TaoTerm: composable objective function terms* for more information on the `TaoTerm` objects.

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 *TAO Algorithms* for more information on the solver methods available in TAO.

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, Tao*);
```

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, TaoType);
```

[1] For more on MPI and PETSc, see *Running PETSc Programs*.

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_type` followed by a TAO method will override any method specified by the second argument. The command line option `-tao_type bqnls`, for instance, will specify the limited-memory quasi-Newton line search method for bound-constrained problems. 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.

Command-line Options

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

```
TaoSetFromOptions(Tao)
```

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

In addition to common command line options shared by all TAO solvers, each TAO method also implements its own specialized options. Please refer to the documentation for individual methods for more details.

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

```
TaoSetSolution(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

```
TaoGetSolution(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.

User Defined Callback Routines

A `Tao` must be able to evaluate a function in order to optimize it; depending on the solver chosen, it may also need to evaluate the gradient vector and Hessian matrix. TAO gives users two ways to specify this information: with callback functions for the evaluation operations (described in this section) provided directly to the `Tao` object, or with `TaoTerm` objects that encapsulate the functions and derivatives (see *TaoTerm: composable objective function terms*).

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.

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

```
PetscErrorCode EvaluateObjective(Tao, Vec, PetscReal*, PetscCtx);
```

in order to evaluate an objective function $f : \mathbb{R}^n \to \mathbb{R}$. 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

```
TaoSetObjective(Tao, PetscErrorCode(*)(Tao, Vec, PetscReal*, PetscCtx), PetscCtx);
```

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 `(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

```
PetscErrorCode EvaluateGradient(Tao, Vec, Vec, PetscCtx);
```

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

```
TaoSetGradient(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec, PetscCtx), PetscCtx);
```

routine. In this routine, the first argument is the Tao object, the second argument is the optional vector to hold the computed gradient, the third argument is the function pointer, and the fourth object is the application context, cast to `(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

```
PetscErrorCode EvaluateFunctionAndGradient(Tao, Vec, PetscReal*, Vec, PetscCtx);
```

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

```
TaoSetObjectiveAndGradient(Tao, Vec PetscErrorCode (*)(Tao, Vec, PetscReal*, Vec,␣
↪PetscCtx), PetscCtx);
```

where the arguments are the TAO application, the optional vector to be used to hold the computed gradient, a function pointer, 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 `PETSC_SUCCESS` after successful completion and a nonzero integer if the function is undefined at that point or an error occurred.

Hessian Evaluation

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

```
PetscErrorCode EvaluateHessian(Tao, Vec, Mat, Mat, PetscCtx);
```

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

```
TaoSetHessian(Tao, Mat, Mat, PetscErrorCode (*)(Tao, Vec, Mat, Mat, PetscCtx),␣
↪PetscCtx);
```

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

```
TaoDefaultComputeGradient(Tao, Vec, Vec, PetscCtx);
```

and

```
TaoDefaultComputeHessian(Tao, Vec, Mat, Mat, PetscCtx);
```

respectively. They can be set by using `TaoSetGradient()` and `TaoSetHessian()` 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, PetscCtx);
```

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

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 also 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 matrix from which to construct the preconditioner, or a user-provided preconditioner shell (`PCSHELL`). 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 *User-Guide*.

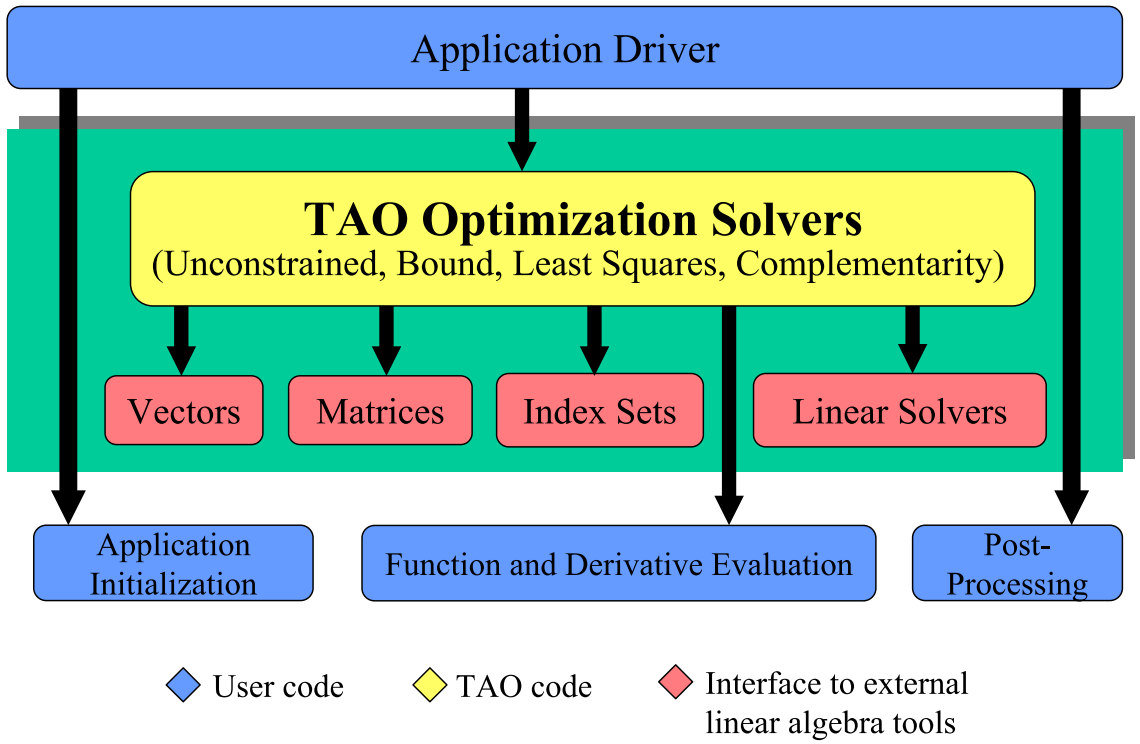


Fig. 2.5: Tao use of PETSc and callbacks

Constraints

Some optimization problems also impose constraints on the variables or intermediate application states. The user defines these constraints through the appropriate TAO interface functions and callback routines where necessary.

Variable Bounds

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 `PETSC_INFINITY` or `PETSC_NINFINITY`. After the two bound vectors have been set, they may be accessed with the command `TaoGetVariableBounds()`.

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.

General Constraints

Some TAO algorithms also support general constraints as a linear or nonlinear function of the optimization variables. These constraints can be imposed either as equalities or inequalities. TAO currently does not make any distinctions between linear and nonlinear constraints, and implements them through the same software interfaces.

In the equality constrained case, TAO assumes that the constraints are formulated as $c_e(x) = 0$ and requires the user to implement a callback routine for evaluating $c_e(x)$ at a given vector of optimization variables,

```
PetscErrorCode EvaluateEqualityConstraints(Tao, Vec, Vec, PetscCtx);
```

As in the previous callback routines, the first argument is the TAO solver object. The second and third arguments are the vector of optimization variables (input) and vector of equality constraints (output), respectively. The final argument is a pointer to the user-defined application context, cast into `(void*)`.

Generally constrained TAO algorithms also require a second user callback function to compute the constraint Jacobian matrix $\nabla_x c_e(x)$,

```
PetscErrorCode EvaluateEqualityJacobian(Tao, Vec, Mat, Mat, PetscCtx);
```

where the first and last arguments are the TAO solver object and the application context pointer as before. The second argument is the vector of optimization variables at which the computation takes place. The third and fourth arguments are the constraint Jacobian and its pseudo-inverse (optional), respectively. The pseudoinverse is optional, and if not available, the user can simply set it to the constraint Jacobian itself.

These callback functions are then given to the TAO solver using the interface functions

```
TaoSetEqualityConstraintsRoutine(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec,␣
↪PetscCtx), PetscCtx);
```

and

```
TaoSetJacobianEqualityRoutine(Tao, Mat, Mat, PetscErrorCode (*)(Tao, Vec, Mat, Mat,␣
↪PetscCtx, PetscCtx);
```

Inequality constraints are assumed to be formulated as $c_i(x) \geq 0$ and follow the same workflow as equality constraints using the `TaoSetInequalityConstraintsRoutine()` and `TaoSetJacobianInequalityRoutine()` interfaces.

Some TAO algorithms may adopt an alternative double-sided $c_l \leq c_i(x) \leq c_u$ formulation and require the lower and upper bounds c_l and c_u to be set using the `TaoSetInequalityBounds(Tao, Vec, Vec)` interface. Please refer to the documentation for each TAO algorithm for further details.

TaoTerm: composable objective function terms

The objective function optimized by `Tao` may be a sum of one or more terms, where each term provides various evaluation routines, such as the objective value, gradient, or Hessian for the term. Here, we define `term` as the basic additive unit used to form an objective function, equipped with appropriate evaluation routines (objective, gradient, and/or Hessian).

For an example, Tikhonov regularization (also known as Ridge Regression), can be formulated as $f(x) + \beta||x||_2^2$. This can be viewed as the summation of two terms, $f(x)$ and $\beta||x||_2^2$.

Each `TaoTerm` encapsulates the routines needed to evaluate its own contribution; `Tao` automatically manages aggregating (summing) the value, gradient, and/or Hessian across all `TaoTerm` objects in the `Tao`

object. This lets users modify terms without changing their base $f(x)$ function code; for example, to add regularization.

Each TaoTerm represents a parametric real-valued function $f(x;p)$ for solution variable x and parameters p. The interface includes methods for evaluating $f(x;p)$ (TaoTermComputeObjective()), $\nabla_x f(x;p)$ (TaoTermComputeGradient() and TaoTermComputeObjectiveAndGradient()), and $\nabla_x^2 f(x;p)$ (TaoTermComputeHessian()).

Adding terms, solution space, parameter space, and the mapping matrix

A TaoTerm can be added to a Tao object by calling the

```
TaoAddTerm(Tao, const char[], PetscReal, TaoTerm, Vec, Mat);
```

routine. The first argument is the Tao object. The second argument is an optional prefix for the TaoTerm. The third argument is the scaling coefficient α, and the fourth argument is the TaoTerm to add to the Tao object. The fifth and sixth arguments are the optional parameters vector and optional mapping matrix, respectively.

If the current objective function of Tao is $f(x)$, then after calling TaoAddTerm() with scale α, term g, parameter p, and map A, the objective becomes

$$f(x) + \alpha g(Ax;p).$$

The mapping matrix A transforms the Tao solution vector x into the term's solution space before evaluation. For example, if the Tao solution vector is $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m\times n}$, then $Ax \in \mathbb{R}^m$ and therefore the term's solution space is $\mathbb{R}^m$. If no mapping matrix is provided, the identity matrix is assumed and the term's solution space must match the Tao solution space. When a mapping matrix is used, the parameter space may depend on either the row or column space of A; see the documentation for each TaoTermType.

Tao automatically applies the scaling and the chain-rule transformation of gradients and Hessians:

- Mapped gradients: $\alpha A^T \nabla g(Ax;p)$
- Mapped Hessians: $\alpha A^T \nabla^2 g(Ax;p) A$

Spaces of TaoTerm

Every TaoTerm has two vector spaces:

- **Solution space** — the vector space of the optimization variable x in $f(x;p)$. Its size is set with TaoTermSetSolutionSizes(), TaoTermSetSolutionLayout(), or TaoTermSetSolutionTemplate(), and reported as N in TaoTermView().

  If a mapping matrix $A \in \mathbb{R}^{m\times n}$ is used, then n must match the dimension of the Tao solution space, and m must match the solution space of the TaoTerm.

- **Parameter space** — the vector space of the parameter vector p in $f(x;p)$. Parameters are fixed data that are *not* optimized over; they are passed to the evaluation routines (e.g., TaoTermComputeObjective()). Its size is set with TaoTermSetParametersSizes(), TaoTermSetParametersLayout(), or TaoTermSetParametersTemplate(), and reported as K in TaoTermView(). Whether a term accepts, requires, or ignores parameters is determined by TaoTermSetParametersMode(). Some TaoTermTypes require the solution and parameter spaces to be related (e.g., have the same size); see the documentation for each type. For users, setting parameter space for built-in TaoTermTypes is generally not needed, except for TAOTERMSHELL.

For an example of using mapping matrices with TaoTerm, see *the elastic net regularization example*, which demonstrates the use of TAOTERMHALFL2SQUARED with a mapping matrix to represent a data misfit term.

Built-in TaoTerm implementations

TAO comes with several built-in implementations for `TaoTerm`:

- `TAOTERMCALLBACKS`: wraps the callback functions set via `TaoSetObjective()`, `TaoSetGradient()`, `TaoSetObjectiveAndGradient()`, and `TaoSetHessian()`. This type is automatically created internally when needed. It does not accept parameters and always has `TAOTERM_PARAMETERS_NONE`.
- `TAOTERMHALFL2SQUARED`: $f(x;p) = \frac{1}{2}\|x-p\|_2^2$ (See `TaoTermCreateHalfL2Squared()`.)
- `TAOTERML1`: $f(x;p) = \|x-p\|_1$ (See `TaoTermCreateL1()`.)
- `TAOTERMQUADRATIC`: $f(x;p) = \frac{1}{2}(x-p)^T A(x-p)$ for matrix A (See `TaoTermCreateQuadratic()`.)
- `TAOTERMSUM`: a sum of other terms implemented by `TaoTerm`, $f(x;p) = \sum_i \alpha_i f(A_i x; p_i)$.
- `TAOTERMSHELL`: an interface for user-defined terms, see *User-defined TaoTerm implementations*.

Specifying TaoTerm Parameters

The parameters p of the parametric function $f(x;p)$ implemented by a `TaoTerm` are passed as arguments in the evaluation routines. For some terms, however, omitting the parameters results in a default value of p being used. For `TAOTERMHALFL2SQUARED`, `TAOTERML1`, and `TAOTERMQUADRATIC` the default is $p = 0$. In general, the parametric behavior of a `TaoTerm` is determined by `TaoTermSetParametersMode()`:

- `TAOTERM_PARAMETERS_OPTIONAL`: default parameters are used if `NULL` is passed for the parameters argument
- `TAOTERM_PARAMETERS_NONE`: the term is not parametric, `NULL` is the only valid parameters argument
- `TAOTERM_PARAMETERS_REQUIRED`: parameters are required, it is an error to pass `NULL` for the parameters argument

Using a TaoTerm in a Tao solver

A `TaoTerm` can be set to an empty `Tao` object or added to an existing `Tao` using `TaoAddTerm()`. The entire objective function of a `Tao` object can be retrieved as a single `TaoTerm` using `TaoGetTerm()`, which returns the term along with its scale, parameters, and mapping matrix (if any).

Currently, `TaoAddTerm()` does not support bounded Newton solvers (`TAOBNK`,`TAOBNLS`,`TAOBNTL`,`TAOBNTR`,and `TAOBQNK`). For these solvers, one must use function callbacks only - `TaoSetObjective()`, `TaoSetGradient()`, `TaoSetObjectiveAndGradient()`, or `TaoSetHessian()`.

For example: if you have specified an objective function $f(x)$ using `TaoSetObjectiveAndGradient()`, and a regularizer $g(x;p)$ is specified by a `TaoTerm`, you can create the objective function $f(x) + \alpha g(Ax;p)$ using:

```
PetscErrorCode (*f_obj_grad)(Tao, Vec, PetscReal *, Vec, void *);
void            *f_ctx;
PetscReal        alpha;
Mat              A;
TaoTerm          g;
Vec              gradient, p;
```

(continues on next page)

(continued from previous page)

```
Tao             tao;

TaoSetObjectiveAndGradient(tao, gradient, f_obj_grad, f_ctx); // f(x)
TaoAddTerm(tao, "regularizer_", alpha, g, p, A);     // + alpha * g(A x ; p)
```

The example $TAO_DIR/src/unconstrained/tutorials/elastic_net_regularization.c uses this interface to define the optimization problem $\min_x \frac{1}{2}\|Ax-b\|_W^2 + \lambda_2 \frac{1}{2}\|x\|_2^2 + \lambda_1 \|Dx-y\|_1$:

Listing: src/tao/unconstrained/tutorials/elastic_net_regularization.c

```
  // the model term,  (1/2) || Ax - b ||_W^2
  PetscCall(TaoTermCreateQuadratic(W, &data_term));
  if (set_prefix) {
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)data_term, "data_"));
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)b, "bvec_"));
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)A, "Amat_"));
  }
  if (set_name) PetscCall(PetscObjectSetName((PetscObject)data_term, "Data TaoTerm"));
  PetscCall(TaoAddTerm(tao, "data_", 1.0, data_term, b, A));
  PetscCall(TaoTermDestroy(&data_term));

  // the L2 term,  (1/2) lambda_2 || x ||_2^2
  PetscCall(TaoTermCreateHalfL2Squared(comm, PETSC_DECIDE, n, &l2_reg_term));
  if (set_prefix) PetscCall(PetscObjectSetOptionsPrefix((PetscObject)l2_reg_term,
↪"ridge_"));
  if (set_name) PetscCall(PetscObjectSetName((PetscObject)l2_reg_term, "Ridge TaoTerm
↪"));
  PetscCall(TaoAddTerm(tao, "ridge_", lambda_2, l2_reg_term, NULL, NULL)); // Note:␣
↪no parameter vector, no map matrix needed
  PetscCall(TaoTermDestroy(&l2_reg_term));

  // the L1 term,  lambda_1 || Dx - y ||_1
  PetscCall(TaoTermCreateL1(comm, PETSC_DECIDE, k, 0.0, &l1_reg_term));
  if (set_prefix) {
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)l1_reg_term, "lasso_"));
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)y, "yvec_"));
    PetscCall(PetscObjectSetOptionsPrefix((PetscObject)D, "Dmat_"));
  }
  if (set_name) PetscCall(PetscObjectSetName((PetscObject)l1_reg_term, "Lasso TaoTerm
↪"));
  PetscCall(TaoAddTerm(tao, "lasso_", lambda_1, l1_reg_term, y, D));
  PetscCall(TaoTermDestroy(&l1_reg_term));

  PetscCall(TaoGetTerm(tao, NULL, &full_objective, NULL, NULL));
  PetscCall(TaoTermCreateSolutionVec(full_objective, &x));
  PetscCall(VecSetRandom(x, rand));
  PetscCall(TaoSetSolution(tao, x));
  PetscCall(TaoSetFromOptions(tao));
  PetscCall(TaoSolve(tao));
```

Regularization terms can also be added to the objective function of a Tao solver from the command line. For instance, the elastic net regularizer $\frac{0.4}{2}\|x\|_2^2 + 0.7\|x\|_1$ can be added with the following options:

```
-tao_add_terms ridge_,lasso_
```

(continues on next page)

(continued from previous page)

```
-ridge_tao_term_type halfl2squared
-lasso_tao_term_type l1
-tao_term_sum_ridge_scale 0.4
-tao_term_sum_lasso_scale 0.7
```

In the above, `ridge_`, and `lasso_` are PETSc option prefixes and could be any unique strings for each term to be added.

When more than one `TaoTerm` object is set to `Tao` (or both `TaoSetObjective()` and `TaoAddTerm()` are used), a `TaoTerm` with type `TAOTERMSUM` gets created internally, and all the subsequently added `TaoTerm` objects get stored in it. With this structure in mind, users can gradually control each term, with the following command line options:

```
// If you want to control how callbacks behave
-callbacks_tao_term_{hessian_mat_type, ...}

// If you want to control regularizers
-ridge_tao_term_{hessian_mat_type, ...}
-lasso_tao_term_{hessian_mat_type, ...}

// If you want to control scaling of each part
-tao_term_sum_{callbacks, ridge, lasso}_scale {number}
```

User-defined TaoTerm implementations

A user-defined `TaoTerm` can be defined from function callbacks using the `TAOTERMSHELL` type. This interface is very similar to `TAOSHELL`: there is a single application context that is set with `TaoTermShellSetContext()` and obtained with `TaoTermShellGetContext()`, and the evaluation routines are set by passing function callbacks with the same signature as routines they implement (see for example `TaoTermShellSetObjectiveAndGradient()`). As an example, $TAO_DIR/src/unconstrained/tutorials/rosenbrock1_taoterm.c in *the example below* demonstrates the same Rosenbrock example as *the first example*.

Listing: `src/tao/unconstrained/tutorials/rosenbrock1_taoterm.c`

```
static PetscErrorCode FormFunctionGradient(TaoTerm, Vec, Vec, PetscReal *, Vec);
static PetscErrorCode FormHessian(TaoTerm, Vec, Vec, Mat, Mat);
static PetscErrorCode CreateSolutionVec(TaoTerm, Vec *);

static PetscErrorCode CtxDestroy(PetscCtxRt ctx)
{
  PetscFunctionBeginUser;
  PetscFunctionReturn(PETSC_SUCCESS);
}

int main(int argc, char **argv)
{
  TaoTerm     objective;
  Tao         tao;  /* Tao solver context */
  PetscMPIInt size; /* number of processes running */
  AppCtx      user; /* user-defined application context */
  MPI_Comm    comm;
  PetscBool   test_gradient_fd_check = PETSC_FALSE; /* test that FD delta is␣
```

(continues on next page)

(continued from previous page)

```
↪preserved */
  PetscReal   fd_delta_set           = 1.e-6;

  /* Initialize TAO and PETSc */
  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));
  comm = PETSC_COMM_WORLD;
  PetscCallMPI(MPI_Comm_size(comm, &size));
  PetscCheck(size == 1, comm, PETSC_ERR_WRONG_MPI_SIZE, "Incorrect number of␣
↪processors");

  PetscOptionsBegin(comm, "", help, "none");
  PetscCall(PetscOptionsGetBool(NULL, NULL, "-test_gradient_fd_check", &test_gradient_
↪fd_check, NULL));
  PetscOptionsEnd();
  /* Initialize problem parameters */
  PetscCall(AppCtxInitialize(comm, &user));

  /* Define the objective function */
  PetscCall(TaoTermCreateShell(comm, &user, CtxDestroy, &objective));
  PetscCall(TaoTermSetParametersMode(objective, TAOTERM_PARAMETERS_NONE));
  PetscCall(TaoTermShellSetCreateSolutionVec(objective, CreateSolutionVec));
  PetscCall(TaoTermShellSetObjectiveAndGradient(objective, FormFunctionGradient));
  PetscCall(TaoTermShellSetCreateHessianMatrices(objective,␣
↪TaoTermCreateHessianMatricesDefault));
  if (user.jacobi_pc) PetscCall(TaoTermSetCreateHessianMode(objective, PETSC_FALSE,␣
↪MATBAIJ, MATBAIJ));
  else PetscCall(TaoTermSetCreateHessianMode(objective, PETSC_TRUE /* H == Hpre */,␣
↪MATBAIJ, NULL));
  PetscCall(TaoTermShellSetHessian(objective, FormHessian));

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

  if (user.use_fd) {
    PetscCall(TaoTermSetFDDelta(objective, 7.e-9));
    PetscCall(TaoTermComputeGradientSetUseFD(objective, PETSC_TRUE));
    PetscCall(TaoTermComputeHessianSetUseFD(objective, PETSC_TRUE));
  }
  /* Set routines for function, gradient, hessian evaluation */
  PetscCall(TaoAddTerm(tao, NULL, 1.0, objective, NULL, NULL));

  /* Check for TAO command line options */
  PetscCall(TaoSetFromOptions(tao));

  /* Set FD delta for testing if requested (after options processing) */
  if (test_gradient_fd_check) PetscCall(TaoTermSetFDDelta(objective, fd_delta_set));

  /* SOLVE THE APPLICATION */
  PetscCall(TaoSolve(tao));

  /* Check that FD delta is preserved if testing */
  if (test_gradient_fd_check) {
    PetscReal fd_delta_get;
```

(continues on next page)

(continued from previous page)

```
    PetscCall(TaoTermGetFDDelta(objective, &fd_delta_get));
    PetscCheck(PetscAbsReal(fd_delta_get - 1.e-6) < 1.e-15, comm, PETSC_ERR_PLIB, "FD␣
↪delta changed: set %g, got %g", (double)fd_delta_set, (double)fd_delta_get);
  }

  /* Clean up */
  PetscCall(AppCtxFinalize(&user, tao));
  PetscCall(TaoDestroy(&tao));
  PetscCall(TaoTermDestroy(&objective));

  PetscCall(PetscFinalize());
return 0;}
```

Masking TaoTerm evaluations

In some cases, for a given `TAOTERMSUM`, the user may only want some evaluation of a specific `TaoTerm` (instead of computing all of them and summing the results). For an example, in a case where `TAOTERMSUM` is composed of `TAOTERMHALFL2SQUARED` and `TAOTERML1`, but the user only wants the objective function evaluation of `TAOTERML1`, and not its gradient and Hessian evaluations. In this case, user can `mask` desired evaluation operations via `TaoTermSumSetTermMask()`. Masking can also be done from the command line. For instance, for the elastic net regularization example above, the user can mask gradient and Hessian evaluation of `TAOTERML1` with the following options:

```
-tao_term_sum_lasso_mask gradient,hessian
```

Solving

Once the application and solver have been set up, the solve takes place with a call to the

```
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 ϵ_{crtol} and either

$$\begin{array}{lcll} ||g(X)|| & \leq & \epsilon_{gatol}, & \\ ||g(X)||/|f(X)| & \leq & \epsilon_{grtol}, & \text{or} \\ ||g(X)||/|g(X_0)| & \leq & \epsilon_{gttol}, & \end{array}$$

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 `TaoSetMaximumFunctionEvaluations()`. `-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 and the gradient can be recovered with the following routines.

```
TaoGetSolution(Tao, Vec*);
TaoGetGradient(Tao, Vec*, NULL, NULL);
```

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

Special Problem structures

Certain special classes of problems solved with TAO utilize specialized code interfaces that are described below per problem type.

PDE-constrained Optimization

TAO solves PDE-constrained optimization problems of the form

$$\begin{aligned} \min_{u,v} \quad & f(u,v) \\ \text{subject to} \quad & g(u,v) = 0, \end{aligned}$$

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.

Unlike other TAO solvers where the solution vector contains only the optimization variables, PDE-constrained problems solved with `tao_lcl` combine the design and state variables together in a monolithic solution vector $x^T = [u^T, v^T]$. Consequently, the user must provide index sets to separate the two,

```
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.

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

```
PetscErrorCode EvaluateConstraints(Tao, Vec, Vec, PetscCtx);
```

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 $g(x)$, 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

```
TaoSetConstraintsRoutine(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec, PetscCtx),␣
↪PetscCtx);
```

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

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

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

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

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

```
PetscErrorCode EvaluateResidual(Tao, Vec, Vec, PetscCtx);
```

and set with the

```
TaoSetResidualRoutine(Tao, PetscErrorCode (*)(Tao, Vec, Vec, PetscCtx), PetscCtx);
```

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

```
PetscErrorCode EvaluateJacobian(Tao, Vec, Mat, PetscCtx;
```

and set with the

```
TaoSetJacobianResidualRoutine(Tao, PetscErrorCode (*)(Tao, Vec, Mat, PetscCtx),␣
↪PetscCtx);
```

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

```
PetscErrorCode EqualityConstraints(Tao, Vec, Vec, PetscCtx);
```

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

```
TaoSetConstraintRoutine(Tao, Vec, PetscErrorCode (*)(Tao, Vec, Vec, PetscCtx),␣
↪PetscCtx);
```

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

```
PetscErrorCode EvaluateJacobian(Tao, Vec, Mat, Mat, PetscCtx);
```

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

```
TaoSetJacobianRoutine(Tao, Mat, Mat, PetscErrorCode (*)(Tao, Vec, Mat, Mat, PetscCtx),
↪ PetscCtx);
```

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.

2.8.3 TAO Algorithms

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_lmvm`) and nonlinear conjugate gradient method (`tao_cg`)
3. Function, gradient, and Hessian evaluations – Newton Krylov methods: Newton line search (`tao_nls`), Newton trust-region (`tao_ntr`), and Newton trust-region line-search (`tao_ntl`)

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.

Newton-Krylov Methods

TAO features three Newton-Krylov algorithms, separated by their globalization methods for unconstrained optimization: line search (NLS), trust region (NTR), and trust region with a line search (NTL). They are available via the TAO solvers `TAONLS`, `TAONTR` and `TAONTL`, respectively, or the `-tao_type nls/ntr/ntl` flag.

Newton Line Search Method (NLS)

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 ρ_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, τ_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 2.17. For the best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

Table 2.17: Summary of `nls` options

Name `-tao_nls_`	Value	Default	Description
`ksp_type`	cg, nash,	stcg	KSPType for linear system
`pc_type`	none, jacobi	lmvm	PCType for linear system
`sval`	real	0	Initial perturbation value
`imin`	real	10^{-4}	Minimum initial perturbation value
`imax`	real	100	Maximum initial perturbation value
`imfac`	real	0.1	Gradient norm factor when initializing perturbation
`pmax`	real	100	Maximum perturbation when increasing value
`pgfac`	real	10	Perturbation growth when increasing value
`pmgfac`	real	0.1	Gradient norm factor when increasing perturbation
`pmin`	real	10^{-12}	Minimum non-zero perturbation when decreasing value
`psfac`	real	0.4	Perturbation shrink factor when decreasing value
`pmsfac`	real	0.1	Gradient norm factor when decreasing perturbation
`nu1`	real	0.25	ν_1 in `step` update
`nu2`	real	0.50	ν_2 in `step` update
`nu3`	real	1.00	ν_3 in `step` update
`nu4`	real	1.25	ν_4 in `step` update
`omega1`	real	0.25	ω_1 in `step` update
`omega2`	real	0.50	ω_2 in `step` update
`omega3`	real	1.00	ω_3 in `step` update
`omega4`	real	2.00	ω_4 in `step` update
`omega5`	real	4.00	ω_5 in `step` update
`eta1`	real	10^{-4}	η_1 in `reduction` update
`eta2`	real	0.25	η_2 in `reduction` update
`eta3`	real	0.50	η_3 in `reduction` update
`eta4`	real	0.90	η_4 in `reduction` update
`alpha1`	real	0.25	α_1 in `reduction` update
`alpha2`	real	0.50	α_2 in `reduction` update
`alpha3`	real	1.00	α_3 in `reduction` update
`alpha4`	real	2.00	α_4 in `reduction` update
`alpha5`	real	4.00	α_5 in `reduction` update
`mu1`	real	0.10	μ_1 in `interpolation` update
`mu2`	real	0.50	μ_2 in `interpolation` update
`gamma1`	real	0.25	γ_1 in `interpolation` update
`gamma2`	real	0.50	γ_2 in `interpolation` update
`gamma3`	real	2.00	γ_3 in `interpolation` update
`gamma4`	real	4.00	γ_4 in `interpolation` update
`theta`	real	0.05	θ in `interpolation` update

The system of equations is approximately solved by applying the conjugate gradient method, Nash conjugate gradient method, Steihaug-Toint conjugate gradient method, generalized Lanczos method, or an alternative Krylov subspace method supplied by PETSc. The method used to solve the systems of equations is specified with the command line argument `-tao_nls_ksp_type (cg|nash|stcg|gltr|gmres)` where `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 ρ_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 ρ_k is zero and a problem was detected with either the direction or the Krylov subspace method, the perturbation is initialized to

$$\rho_{k+1} = \text{median}\left\{\text{imin}, \text{imfac} * \|g(x_k)\|, \text{imax}\right\},$$

   where $g(x_k)$ is the gradient of the objective function and `imin` is set with the command line argument `-tao_nls_imin imin` with a default value of 10^{-4}, `imfac` by `-tao_nls_imfac` 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 λ_1 of the Hessian matrix is available. This value is used to initialize the perturbation to $\rho_{k+1} = \max\left\{\rho_{k+1}, -\lambda_1\right\}$ in this case.

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

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

   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 ρ_k is nonzero and no problems were detected with either the direction or Krylov subspace method, the perturbation is decreased to

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

   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 radius` 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 τ_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 (NTR)

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

$$\begin{array}{ll} \min_d & \frac{1}{2} d^T H_k d + g_k^T d \\ \text{subject to} & \|d\| \leq \Delta_k \end{array}$$

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 Δ_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 2.18. For the best efficiency, function and gradient evaluations should be performed separately when using this algorithm.

Table 2.18: Summary of `ntr` options

Name `-tao_ntr_`	Value	Default	Description
`ksp_type`	nash, stcg	stcg	KSPType for linear system
`pc_type`	none, jacobi	lmvm	PCType for linear system
`init_type`	constant, direction, interpolation	interpolation	Radius initialization method
`mu1_i`	real	0.35	μ_1 in `interpolation` init

continues on next page

Table 2.18 – continued from previous page

Name -tao_ntr_	Value	Default	Description
mu2_i	real	0.50	μ_2 in interpolation init
gamma1_i	real	0.0625	γ_1 in interpolation init
gamma2_i	real	0.50	γ_2 in interpolation init
gamma3_i	real	2.00	γ_3 in interpolation init
gamma4_i	real	5.00	γ_4 in interpolation init
theta_i	real	0.25	θ in interpolation init
update_type	step, reduction, interpolation	step	Radius update method
mu1_i	real	0.35	μ_1 in interpolation init
mu2_i	real	0.50	μ_2 in interpolation init
gamma1_i	real	0.0625	γ_1 in interpolation init
gamma2_i	real	0.50	γ_2 in interpolation init
gamma3_i	real	2.00	γ_3 in interpolation init
gamma4_i	real	5.00	γ_4 in interpolation init
theta_i	real	0.25	θ in interpolation init
eta1	real	:	η_1 in reduction update
eta2	real	0.25	η_2 in reduction update
eta3	real	0.50	η_3 in reduction update
eta4	real	0.90	η_4 in reduction update
alpha1	real	0.25	α_1 in reduction update
alpha2	real	0.50	α_2 in reduction update
alpha3	real	1.00	α_3 in reduction update
alpha4	real	2.00	α_4 in reduction update
alpha5	real	4.00	α_5 in reduction update
mu1	real	0.10	μ_1 in interpolation update
mu2	real	0.50	μ_2 in interpolation update
gamma1	real	0.25	γ_1 in interpolation update
gamma2	real	0.50	γ_2 in interpolation update
gamma3	real	2.00	γ_3 in interpolation update
gamma4	real	4.00	γ_4 in interpolation update
theta	real	0.05	θ 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)` where `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|jacobi|icc|ilu|lmvm)`, respectively. The default is the `lmvm` 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 radius` 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.

Newton Trust Region with Line Search (NTL)

NTL 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.

This algorithm will be deprecated in the next version and replaced by the Bounded Newton Trust Region with Line Search (BNTL) algorithm that can solve both bound constrained and unconstrained problems.

Limited-Memory Variable-Metric Method (LMVM)

The limited-memory, variable-metric method (LMVM) 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, τ_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 (CG)

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-Ribiére method, the Polak-Ribiére-Plus method [NW06], 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 η, the algorithm restarts using the gradient direction. The parameter η can be set by using the command line argument `-tao_cg_eta eta`; where 0.1 is the default value.

This algorithm will be deprecated in the next version and replaced by the Bounded Nonlinear Conjugate Gradient (BNCG) algorithm that can solve both bound constrained and unconstrained problems.

Nelder-Mead Simplex Method (NM)

The Nelder-Mead algorithm [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 μ 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_lambda min_lambda
sets the initial set of vectors (x_0 plus `value` in each coordinate direction); the default value is 1.

-tao_nm_mu mu
sets the value of μ_0; the default is $\mu_0 = 1$.

Bound-Constrained Optimization

Bound-constrained optimization algorithms solve optimization problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) \\ \text{subject to} & l \leq x \leq u. \end{array}$$

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:

$$\mathfrak{P}(g) = \begin{cases} 0 & \text{if } (x \leq l_i \wedge g_i > 0) \vee (x \geq u_i \wedge g_i < 0) \\ g_i & \text{otherwise} \end{cases}$$

- Bound projection:

$$\mathfrak{B}(x) = \begin{cases} l_i & \text{if } x_i < l_i \\ u_i & \text{if } x_i > u_i \\ x_i & \text{otherwise} \end{cases}$$

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:

$$\begin{aligned}
\text{lower bounded}: \quad \mathcal{L}(x) &= \{i \ : \ x_i \leq l_i + \epsilon \ \wedge \ g(x)_i > 0\}, \\
\text{upper bounded}: \quad \mathcal{U}(x) &= \{i \ : \ x_i \geq u_i + \epsilon \ \wedge \ g(x)_i < 0\}, \\
\text{fixed}: \quad \mathcal{F}(x) &= \{i \ : \ l_i = u_i\}, \\
\text{active-set}: \quad \mathcal{A}(x) &= \{\mathcal{L}(x) \bigcup \mathcal{U}(x) \bigcup \mathcal{F}(x)\}, \\
\text{inactive-set}: \quad \mathcal{I}(x) &= \{1, 2, \ldots, n\} \setminus \mathcal{A}(x).
\end{aligned}$$

At each iteration, the bound tolerance is estimated as $\epsilon_{k+1} = \min(\epsilon_k, ||w_k||_2)$ with $w_k = x_k - \mathfrak{B}(x_k - \beta D_k g_k)$, where the diagonal matrix D_k is an approximation of the Hessian inverse H_k^{-1}. The initial bound tolerance ϵ_0 and the step length β 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/icc/jacobi/lmvm`. The `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. `icc` and `ilu` 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 its`. 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 ρ_k is a dynamically adjusted positive constant. The step is globalized using a projected Moré-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 (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 Quasi-Newton Line Search (BQNLS)

The 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`.

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 $\mathcal{O}(10)$ 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 Nonlinear Conjugate Gradient (BNCG)

BNCG extends the unconstrained nonlinear conjugate gradient algorithm to bound constraints via gradient projections and a bounded Moré-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_brdn)`, 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 ξ; 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`.

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 tolerances 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 $\nabla f(x_k)^T d_k \in [-10^{11}, -10^{-9}]$. 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.

Generally Constrained Solvers

Constrained solvers solve optimization problems that incorporate either or both equality and inequality constraints, and may optionally include bounds on solution variables.

Alternating Direction Method of Multipliers (ADMM)

The TAOADMM algorithm is intended to blend the decomposability of dual ascent with the superior convergence properties of the method of multipliers. [BPC+11] The algorithm solves problems in the form

$$\begin{aligned} \min_{x} \quad & f(x) + g(z) \\ \text{subject to} \quad & Ax + Bz = c \end{aligned}$$

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, ρ, 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 ρ, which may change in the case of `-tao_admm_dual_update (update_basic|update_adaptive|update_adaptive_relaxed)`. This issue can be prevented by `TaoADMMSetMinimumSpectralPenalty()`.

Augmented Lagrangian Method of Multipliers (ALMM)

The TAOALMM method solves generally constrained problems of the form

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g(x) = 0 \\ & h(x) \geq 0 \\ & l \leq x \leq u \end{aligned}$$

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 $k = 1, 2, \ldots$

$$\begin{aligned} \min_{x} \quad & L(x, \lambda_k) \\ \text{subject to} \quad & l \leq x \leq u \end{aligned}$$

where $L(x, \lambda_k)$ is the augmented Lagrangian merit function and λ_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 [Hes69] [Pow69] with inequality constrained converted to equality constraints via slack variables, and the slack-less Powell-Hestenes-Rockafellar formulation [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.

Primal-Dual Interior-Point Method (PDIPM)

The TAOPDIPM method (`-tao_type pdipm`) implements a primal-dual interior point method for solving general nonlinear programming problems of the form

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g(x) = 0 \\ & h(x) \geq 0 \\ & x^- \leq x \leq x^+ \end{aligned} \tag{2.6}$$

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.

PDIPM converts the inequality constraints to equalities using slack variables z and a log-barrier term, which transforms (2.6) to

$$\begin{aligned} \min \; & f(x) - \mu \sum_{i=1}^{nci} \ln z_i \\ \text{s.t.} \quad & \\ & ce(x) = 0 \\ & ci(x) - z = 0 \end{aligned} \tag{2.7}$$

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. μ is a parameter that is driven to zero as the optimization progresses.

The Lagrangian for (2.7)) is

$$L_\mu(x, \lambda_{ce}, \lambda_{ci}, z) = f(x) + \lambda_{ce}^T ce(x) - \lambda_{ci}^T (ci(x) - z) - \mu \sum_{i=1}^{nci} \ln z_i \tag{2.8}$$

where, λ_{ce} and λ_{ci} are the Lagrangian multipliers for the equality and inequality constraints, respectively.

The first order KKT conditions for optimality are as follows

$$\nabla L_\mu(x, \lambda_{ce}, \lambda_{ci}, z) = \begin{bmatrix} \nabla f(x) + \nabla ce(x)^T \lambda_{ce} - \nabla ci(x)^T \lambda_{ci} \\ ce(x) \\ ci(x) - z \\ Z\Lambda_{ci} e - \mu e \end{bmatrix} = 0 \tag{2.9}$$

(2.9) 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 μ is also updated after each Newton iteration. The Newton update is obtained by solving the second-order KKT system $Hd = -\nabla L_\mu$. 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`.

PDE-Constrained Optimization

TAO solves PDE-constrained optimization problems of the form

$$\begin{aligned} \min_{u,v} \quad & f(u, v) \\ \text{subject to} \quad & g(u, v) = 0, \end{aligned}$$

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 (LCL)

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

$$\begin{aligned} \min_{u,v} \quad & \tilde{f}_k(u, v) \\ \text{subject to} \quad & A_k(u - u_k) + B_k(v - v_k) + g_k = 0, \end{aligned}$$

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

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

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 ρ_k so that du is also a sufficient descent direction for the augmented Lagrangian merit function. We then find α 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,

$$\begin{aligned} u_{k+\frac{1}{2}} &= u_k + \alpha_k du \\ v_{k+\frac{1}{2}} &= v_k, \end{aligned}$$

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:

$$\begin{aligned} \min_{u,v} \quad & \tilde{f}_k(u,v) \\ \text{subject to} \quad & A_k(u - u_k) + B_k(v - v_k) + \alpha_k g_k = 0. \end{aligned}$$

We begin with the change of variables

$$\begin{aligned} \min_{du,dv} \quad & \tilde{f}_k(u_k + du, v_k + dv) \\ \text{subject to} \quad & A_k du + B_k dv + \alpha_k g_k = 0 \end{aligned}$$

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

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

which is equivalent to

$$\min_{dv} \quad \tilde{f}_k(u_{k+\frac{1}{2}} - A_k^{-1} B_k dv, v_{k+\frac{1}{2}} + dv).$$

We apply one step of a limited-memory quasi-Newton method to this problem. The direction is obtain by solving the quadratic problem

$$\min_{dv} \quad \tfrac{1}{2} dv^T \tilde{H}_k dv + \tilde{g}_{k+\frac{1}{2}}^T dv,$$

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.

$$\begin{aligned} \tilde{g}_{k+\frac{1}{2}} &= \nabla_v \tilde{f}_k(u_{k+\frac{1}{2}}, v_{k+\frac{1}{2}}) - \nabla_u \tilde{f}_k(u_{k+\frac{1}{2}}, v_{k+\frac{1}{2}}) A_k^{-1} B_k \\ &= d_{k+\frac{1}{2}} + c_{k+\frac{1}{2}} A_k^{-1} B_k \end{aligned}$$

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.

$$\min_{\beta \geq 0} \quad \tilde{f}_k(u_{k+\frac{1}{2}} + \beta du, v_{k+\frac{1}{2}} + \beta dv)$$

We enforce the Wolfe conditions during the search procedure. The new point is

$$\begin{aligned} u_{k+1} &= u_{k+\frac{1}{2}} + \beta_k du \\ v_{k+1} &= v_{k+\frac{1}{2}} + \beta_k dv. \end{aligned}$$

The reduced gradient at the new point is computed from

$$\begin{aligned} y_{k+1} &= A_k^{-T} c_{k+1} \\ \tilde{g}_{k+1} &= d_{k+1} - y_{k+1}^T B_k, \end{aligned}$$

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

$$f(x) = \|F(x)\|_2^2 = \sum_{i=1}^{m} F_i(x)^2. \tag{2.10}$$

The nonlinear equations F should be specified with the function `TaoSetResidual()`.

Bound-constrained Regularized Gauss-Newton (BRGN)

The TAOBRGN algorithms is a 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 α_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 (`-tao_type brgn`) implementation adds a regularization term $\beta(x)$ such that

$$\min_x \ \frac{1}{2}||R(x)||_2^2 + \lambda\beta(x),$$

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

- **L2-norm** - $\beta(x) = \frac{1}{2}||x_k||_2^2$
- **L2-norm Proximal Point** - $\beta(x) = \frac{1}{2}||x_k - x_{k-1}||_2^2$
- **L1-norm with Dictionary** - $\beta(x) = ||Dx||_1 \approx \sum_i \sqrt{y_i^2 + \epsilon^2} - \epsilon$ where $y = Dx$ and ϵ 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 $\mathcal{C}2$-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.

POUNDERS

One algorithm for solving the least squares problem ((2.10)) 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 (`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 $\|\nabla m_k(x_k)\|_2 >$`gtol`, the next point x_+ to be evaluated is obtained by solving the trust-region subproblem

$$\min \{m_k(x) : \|x - x_k\|_p \leq \Delta_k, \}, \tag{2.11}$$

where Δ_k is the current trust-region radius. By default we use a trust-region norm with $p = \infty$ and solve ((2.11)) with the BLMVM method described in *Bound-constrained Limited-Memory Variable-Metric Method (BLMVM)*. 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, a bounded Newton-Krylov Method should be used; the default is the BNTR algorithm. Note: BNTR uses its own internal trust region that may interfere with the infinity-norm trust region used in the model problem ((2.11)).

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, `valid`, on the model's quality of approximation on the trust region is then used to update the iterate,

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

and trust-region radius,

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

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

If $\rho_k \leq 0$ and `valid` is `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

$$q_k^{(i)}(x) = F_i(x_k) + (x - x_k)^T g_k^{(i)} + \frac{1}{2}(x - x_k)^T H_k^{(i)}(x - x_k), \qquad i = 1, \ldots, m \tag{2.12}$$

thus satisfy the interpolation conditions

$$q_k^{(i)}(y_j) = F_i(y_j), \qquad i = 1, \ldots, m;\ j = 1, \ldots, l_k$$

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 *(12)* are then used to construct the main model,

$$m_k(x) = f(x_k) + \tag{2.13}$$

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

The process of forming these models also computes the indicator `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:

`-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.

`-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.

`-tao_pounders_gqt`
Use the gqt algorithm to solve the subproblem ((2.11)) (uses $p = 2$) instead of BQPIP.

`-pounders_subsolver`
If the default BQPIP algorithm is used to solve the subproblem ((2.11)), the parameters of the subproblem solver can be accessed using the command line options prefix `-pounders_subsolver_`. For example,

```
-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 *Objective Function and Gradient Routines* and *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.

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, $\mathbf{x}^* \in [\ell, u]$ is a solution to MCP(F, ℓ, u) if for each $i \in \{1, \ldots, n\}$ we have at least one of the following:

$$\begin{aligned} F_i(x^*) &\geq 0 \quad \text{if } x_i^* = \ell_i \\ F_i(x^*) &= 0 \quad \text{if } \ell_i < x_i^* < u_i \\ F_i(x^*) &\leq 0 \quad \text{if } x_i^* = u_i. \end{aligned}$$

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(∇f, ℓ, 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,

$$\phi(a, b) = 0 \quad \Leftrightarrow \quad a \geq 0,\ b \geq 0,\ ab = 0,$$

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 Φ is not differentiable everywhere but satisfies a semismoothness property [Mif77] [Qi93] [QS93]. Furthermore, the natural merit function, $\Psi(x) := \frac{1}{2}\|\Phi(x)\|_2^2$, 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 Φ 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.

Active-Set Methods

TAO also contained two active-set semismooth methods for solving complementarity problems. These methods solve a reduced system constructed by block elimination of active constraints. The subdifferential in these cases enables this block elimination.

The first active-set semismooth algorithm available in TAO is not guaranteed to remain feasible with respect to the bounds, $[\ell, u]$, and is termed an infeasible active-set semismooth method. This method can be specified by using the `tao_asils` solver.

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_asfls` solver.

Quadratic Solvers

Quadratic solvers solve optimization problems of the form

$$\begin{array}{ll} \min\limits_{x} & \frac{1}{2}x^T Q x + c^T x \\ \text{subject to} & l \geq x \geq u \end{array}$$

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

Gradient Projection Conjugate Gradient Method (GPCG)

The GPCG [MoreT91] algorithm is much like the TRON algorithm, discussed in Section *Trust-Region Newton Method (TRON)*, 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 optio flag `-tao_type gpcg`.

Interior-Point Newton's Method (BQPIP)

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 bgpip`. 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()`.

Legacy and Contributed Solvers

Bundle Method for Regularized Risk Minimization (BMRM)

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. λ 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.

Orthant-Wise Limited-memory Quasi-Newton (OWLQN)

OWLQN [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, λ is a positive weight parameter, and $\|x\|_1$ is the ℓ_1 norm of x: $\sum_i |x_i|$. The algorithm only requires evaluating the value of f and its gradient.

Trust-Region Newton Method (TRON)

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.

This algorithm will be deprecated in the next version in favor of the Bounded Newton Trust Region (BNTR) algorithm.

Bound-constrained Limited-Memory Variable-Metric Method (BLMVM)

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.

2.8.4 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

```
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 *User-Guide*.

Monitors

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

```
TaoMonitorSet(Tao, PetscErrorCode (*mon)(Tao,void*), void*);
```

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 `TaoMonitorSet()` 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 [MoreT92], 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_max_funcs 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`, `TAOBQNKLS`, `TAOBQNKTR`, `TAOBQNKTL`), 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.

2.8.5 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 uses 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 `taoimpl.h`:

```
#include "petsc/private/taoimpl.h"
```

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.

```
typedef struct{

  PetscReal beta;
  PetscReal eta;
  PetscInt  ngradtseps;
  PetscInt  nresetsteps;
  Vec X_old;
```

(continues on next page)

(continued from previous page)

```
  Vec G_old;

} TAO_CG;
```

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 `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.

```
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;

  PetscFunctionBegin;

  PetscCall(TaoComputeObjectiveAndGradient(tao,x,&f,g));
  PetscCall(VecNorm(g,NORM_2,&gnorm));

  PetscCall(VecSet(s,0));

  cg->beta=0;
  gnormPrev = gnorm;

  /* Enter loop */
  while (1){

    /* Test for convergence */
    PetscCall(TaoMonitor(tao,iter,f,gnorm,0.0,step,&reason));
    if (reason!=TAO_CONTINUE_ITERATING) break;

    cg->beta=(gnorm*gnorm)/(gnormPrev*gnormPrev);
    PetscCall(VecScale(s,cg->beta));
    PetscCall(VecAXPY(s,-1.0,g));

    PetscCall(VecDot(s,g,&gdx));
    if (gdx>=0){     /* If not a descent direction, use gradient */
      PetscCall(VecCopy(g,s));
      PetscCall(VecScale(s,-1.0));
      gdx=-gnorm*gnorm;
    }

    /* Line Search */
```

(continues on next page)

(continued from previous page)

```
    gnormPrev = gnorm;  step=1.0;
    PetscCall(TaoLineSearchSetInitialStepLength(tao->linesearch,1.0));
    PetscCall(TaoLineSearchApply(tao->linesearch,x,&f,g,s,&steplength,&lsflag));
    PetscCall(TaoAddLineSearchCounts(tao));
    PetscCall(VecNorm(g,NORM_2,&gnorm));
    iter++;
  }

  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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 an 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

```
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

```
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.

```
PETSC_EXTERN PetscErrorCode TaoCreate_CG(Tao tao)
{
  TAO_CG *cg = (TAO_CG*)tao->data;
  const char *morethuente_type = TAOLINESEARCH_MT;

  PetscFunctionBegin;

  PetscCall(PetscNew(&cg));
  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;
  tao->ops->setfromoptions = TaoSetFromOptions_CG;
  tao->ops->destroy = TaoDestroy_CG;

  PetscCall(TaoLineSearchCreate(((PetscObject)tao)->comm, &tao->linesearch));
  PetscCall(TaoLineSearchSetType(tao->linesearch, morethuente_type));
  PetscCall(TaoLineSearchUseTaoRoutines(tao->linesearch, tao));

  PetscFunctionReturn(PETSC_SUCCESS);
}
EXTERN_C_END
```

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.

```
PetscErrorCode TaoDestroy_CG(TAO_SOLVER tao)
{
  TAO_CG *cg = (TAO_CG *) tao->data;

  PetscFunctionBegin;

  PetscCall(VecDestroy(&cg->X_old));
  PetscCall(VecDestroy(&cg->G_old));

  PetscFree(tao->data);
  tao->data = NULL;

  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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.

```
PetscErrorCode TaoSetUp_CG(Tao tao)
{
  TAO_CG *cg = (TAO_CG*)tao->data;
  PetscFunctionBegin;

  PetscCall(VecDuplicate(tao->solution,&tao->gradient));
  PetscCall(VecDuplicate(tao->solution,&tao->stepdirection));
  PetscCall(VecDuplicate(tao->solution,&cg->X_old));
  PetscCall(VecDuplicate(tao->solution,&cg->G_old));

  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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.

```
PetscErrorCode TaoSetFromOptions_CG(Tao tao, void *solver);
{
  TAO_CG *cg = (TAO_CG*)solver;
```

(continues on next page)

(continued from previous page)

```
  PetscFunctionBegin;
  PetscCall(PetscOptionsReal("-tao_cg_eta","restart tolerance","",cg->eta,&cg->eta,
↪0));
  PetscCall(PetscOptionsReal("-tao_cg_delta_min","minimum delta value","",cg->delta_
↪min,&cg->delta_min,0));
  PetscCall(PetscOptionsReal("-tao_cg_delta_max","maximum delta value","",cg->delta_
↪max,&cg->delta_max,0));
  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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.

```
PetscErrorCode TaoView_CG(Tao tao, PetscViewer viewer)
{
  TAO_CG *cg = (TAO_CG*)tao->data;

  PetscFunctionBegin;
  PetscCall(PetscViewerASCIIPushTab(viewer));
  PetscCall(PetscViewerASCIIPrintf(viewer,"Grad. steps: %d\n",cg->ngradsteps));
  PetscCall(PetscViewerASCIIPrintf(viewer,"Reset steps: %d\n",cg->nresetsteps));
  PetscCall(PetscViewerASCIIPopTab(viewer));
  PetscFunctionReturn(PETSC_SUCCESS);
}
```

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

```
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 PetscRegressor: Regression Solvers

The `PetscRegressor` component provides some basic infrastructure and a general API for supervised machine learning tasks at a higher level of abstraction than a purely algebraic "solvers" view. Methods are currently available for

- *Linear regressor*

Note that by "regressor" we mean an algorithm or implementation used to fit and apply a regression model, following standard parlance in the machine-learning community. Regressor here does NOT mean an independent (or predictor) variable, as it often does in the statistics community.

2.9.1 Basic Regressor Usage

`PetscRegressor` supports supervised learning tasks: Given a matrix of observed data X with size $n_{samples}$ by $n_{features}$, predict a vector of "target" values y (of size $n_{samples}$), where the ith entry of y corresponds to the observation (or "sample") stored in the ith row of X. Traditionally, when the target consists of continuous values this is called "regression", and when it consists of discrete values (or "labels"), this task is called "classification"; we use `PetscRegressor` to support both of these cases.

Before a regressor can be used to make predictions, the model must be fitted using an initial set of training data. Once a fitted model has been obtained, it can be used to predict target values for new observations. Every `PetscRegressor` implementation provides a `Fit()` and a `Predict()` method to support this workflow. Fitting (or "training") a model is a relatively computationally intensive task that generally involves solving an optimization problem (often using `TAO` solvers) to determine the model parameters, whereas making predictions (or performing "inference") is generally much simpler.

Here, we introduce a simple example to demonstrate `PetscRegressor` usage. Please read *Regression Solvers* for more in-depth discussion. The code presented *below* solves an ordinary linear regression problem, with various options for regularization.

In the simplest usage of a regressor, the user provides a training (or "design") matrix (`Mat`) and a target vector (`Vec`) against which to fit the model. Once the regressor is fitted, the user can then obtain a vector of predicted values for a set of new observations.

PETSc's default method for solving regression problems is ordinary least squares, `REGRESSOR_LINEAR_OLS`, which is a sub-type of linear regressor, `PETSCREGRESSORLINEAR`. By "linear" we mean that the model $f(x, \theta)$ is linear in its coefficients θ but not necessarily linear in its features x.

Note that data creation, option parsing, and cleaning stages are omitted here for clarity. The complete code is available in *ex3.c*.

Listing: `src/ml/regressor/tests/ex3.c`

```
#include <petscregressor.h>
int main(int argc, char **args)
{
  AppCtx         ctx;
  PetscRegressor regressor;
  PetscScalar    intercept;

  /* Initialize PETSc */
  PetscCall(PetscInitialize(&argc, &args, (char *)0, help));

  /* Initialize problem parameters and data */
  PetscCall(PetscNew(&ctx));
```

(continues on next page)

(continued from previous page)

```
  PetscCall(ConfigureContext(ctx));
  PetscCall(CreateData(ctx));

  /* Create Regressor solver with desired type and options */
  PetscCall(PetscRegressorCreate(PETSC_COMM_WORLD, &regressor));
  PetscCall(PetscRegressorSetType(regressor, PETSCREGRESSORLINEAR));
  PetscCall(PetscRegressorLinearSetType(regressor, REGRESSOR_LINEAR_OLS));
  PetscCall(PetscRegressorLinearSetFitIntercept(regressor, PETSC_FALSE));
  /* Testing prefix functions for Regressor */
  PetscCall(TestPrefixRegressor(regressor, ctx));
  /* Check for command line options */
  PetscCall(PetscRegressorSetFromOptions(regressor));
  /* Fit the regressor */
  PetscCall(PetscRegressorFit(regressor, ctx->X, ctx->y));
  /* Predict data with fitted regressor */
  PetscCall(PetscRegressorPredict(regressor, ctx->X, ctx->y_predicted));
  /* Get other desired output data */
  PetscCall(PetscRegressorLinearGetIntercept(regressor, &intercept));
  PetscCall(PetscRegressorLinearGetCoefficients(regressor, &ctx->coefficients));

  /* Testing Views, and GetTypes */
  PetscCall(TestRegressorViews(regressor, ctx));
  PetscCall(PetscRegressorDestroy(&regressor));
  PetscCall(DestroyCtx(&ctx));
  PetscCall(PetscFinalize());
return 0;}
```

To create a `PetscRegressor` instance, one must first call `PetscRegressorCreate()`:

```
PetscRegressorCreate(MPI_Comm comm, PetscRegressor *regressor);
```

To choose a regressor type, the user can either call

```
PetscRegressorSetType(PetscRegressor regressor, PetscRegressorType type);
```

or use the command-line option `-regressor_type <method>`; details regarding the available methods are presented in *Regression Solvers*. The application code can specify the options used by underlying linear, nonlinear, and optimization solver methods used in fitting the model by calling

```
PetscRegressorSetFromOptions(regressor);
```

which interfaces with the PETSc options database and enables convenient runtime selection of the type of regression algorithm and setting various various solver or problem parameters. This routine can also control all inner solver options in the `KSP`, and `Tao` modules, as discussed in *KSP: Linear System Solvers*, *TAO: Optimization Solvers*.

After having set these routines and options, the user can fit (or "train") the regressor by calling

```
PetscRegressorFit(PetscRegressor regressor, Mat X, Vec y);
```

where `X` is training data, and `y` is target values. Finally, after fitting the regressor, the user can compute model predictions, that is, perform inference, for a data matrix of unlabeled observations using the fitted regressor:

```
PetscRegressorPredict(PetscRegressor regressor, Mat X, Vec y_predicted);
```

Finally, after the user is done using the regressor, the user should destroy its PetscRegressor context with

```
PetscRegressorDestroy(PetscRegressor *regressor);
```

2.9.2 Regression Solvers

One can see the list of regressor types in Table *PETSc Regressor*. Currently, we only support one type, PETSCREGRESSORLINEAR, although we plan to add several others in the near future.

Table 2.19: PETSc Regressor

Method	PetscRegressorType	Options Name
Linear	PETSCREGRESSORLINEAR	linear

If the particular method being employed is one that supports regularization, the user can set regularizer's weight via

```
PetscRegressorSetRegularizerWeight(PetscRegressor regressor, PetscReal weight);
```

or with the option -regressor_regularizer_weight <weight>.

2.9.3 Linear regressor

The PETSCREGRESSORLINEAR (-regressor_type linear) implementation constructs a linear model to reduce the sum of squared differences between the actual target values ("observations") in the dataset and the target values estimated by the fitted model. By default, bound-constrained regularized Gauss-Newton TAOBRGN is used to solve the underlying optimization problem.

Currently, linear regressor has three types, which are described in Table *Linear Regressor types*.

Table 2.20: Linear Regressor types

Linear method	PetscRegressorLinearType	Options Name
Ordinary	REGRESSOR_LINEAR_OLS	ols
Lasso	REGRESSOR_LINEAR_LASSO	lasso
Ridge	REGRESSOR_LINEAR_RIDGE	ridge

If one wishes, the user can (when appropriate) use KSP to solve the problem, instead of Tao, via

```
PetscRegressorLinearSetUseKSP(PetscRegressor regressor, PetscBool flg);
```

or with the option -regressor_linear_use_ksp <true,false>.

Calculation of the intercept (also known as the "bias" or "offset") is performed separately from the rest of the model fitting process, because data sets are often already mean-centered and because it is generally undesirable to regularize the intercept term. By default, this step is omitted; if the user wishes to compute the intercept, this can be done by calling

```
PetscRegressorLinearSetFitIntercept(PetscRegressor regressor, PetscBool flg);
```

or by specifying the option `-regressor_linear_fit_intercept <true,false>`.

For a fitted regression, one can obtain the intercept and a vector of the model coefficients from a linear regression model via

```
PetscRegressorLinearGetCoefficients(PetscRegressor regressor, Vec *coefficients);
PetscRegressorLinearGetIntercept(PetscRegressor regressor, PetscScalar *intercept);
```

2.10 PetscDA: Data Assimilation

PETSc's `PetscDA` object coordinates data assimilation (DA) workflows.

This is new code, please independently verify all results you obtain using it.

Some planned work for `PetscDA` is available as GitLab Issue #1882

2.10.1 Ensemble-based Data Assimilation

Currently `PetscDA` only supports ensemble-based data assimilation with two `PetscDAType`: `PETSCDAETKF` and `PETSCDALETKF`. These focus on ensemble transform Kalman filter (ETKF)-style updates but are extensible to other assimilation techniques.

- *ETKF*
- *LETKF*

These centralize ensemble storage, observational metadata, and user-defined forecast/analysis operators so that algorithms can run independently of the MPI layout or the vector/matrix backends.

2.10.2 Lifecycle overview

A typical assimilation cycle alternates between forecast propagation and statistical analysis:

1. Initialize a `PetscDA` context and configure ensemble sizes and data structures.
2. Populate the ensemble state vectors and optional observation-error descriptions.
3. Advance each ensemble member with a model operator supplied by the application.
4. Combine forecasts with observations through `PetscDAEnsembleAnalysis()` to produce the posterior ensemble.
5. Repeat until the desired simulation horizon is complete, optionally extracting diagnostics after each phase.

Throughout this loop the `PetscDA` object abstracts the global vectors, scatters, and reductions needed to compute ensemble means, anomalies, and square-root transforms.

Here we introduce a simple example to demonstrate `PetscDA` usage with the `PETSCDALETKF` on the Lorenz-96 model. Please read *Implementations* for more in-depth discussion of the available implementations.

Listing: `src/ml/da/tutorials/ex1.c`

```
int main(int argc, char **argv)
{
  /* Configuration parameters */
  PetscInt  n                = DEFAULT_N;
  PetscInt  steps            = DEFAULT_STEPS;
  PetscInt  burn             = DEFAULT_BURN;
  PetscInt  obs_freq         = DEFAULT_OBS_FREQ;
  PetscInt  random_seed      = DEFAULT_RANDOM_SEED;
  PetscInt  ensemble_size    = DEFAULT_ENSEMBLE_SIZE;
  PetscReal F                = DEFAULT_F;
  PetscReal dt               = DEFAULT_DT;
  PetscReal obs_error_std    = DEFAULT_OBS_ERROR_STD;
  PetscReal ensemble_init_std = 1; /* Initial ensemble spread */

  /* PETSc objects */
  Lorenz96Ctx *l95_ctx = NULL, *truth_ctx = NULL;
  DM           da_state;
  PetscDA      da;
  Vec          x0, x_mean, x_forecast;
  Vec          truth_state, rmse_work;
  Vec          observation, obs_noise, obs_error_var;
  PetscRandom  rng;
  Mat          H = NULL; /* Observation operator matrix */

  /* Statistics tracking */
  PetscReal rmse_forecast = 0.0, rmse_analysis = 0.0;
  PetscReal sum_rmse_forecast = 0.0, sum_rmse_analysis = 0.0;
  PetscInt  n_stat_steps = 0;
  PetscInt  obs_count    = 0;
  PetscInt  step, progress_interval;

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));

  /* Parse command-line options */
  PetscOptionsBegin(PETSC_COMM_WORLD, NULL, "Lorenz-96 ETKF Quick Example", NULL);
  PetscCall(PetscOptionsInt("-n", "State dimension", "", n, &n, NULL));
  PetscCall(PetscOptionsInt("-steps", "Number of time steps", "", steps, &steps,␣
↪NULL));
  PetscCall(PetscOptionsInt("-burn", "Burn-in steps excluded from statistics", "",␣
↪burn, &burn, NULL));
  PetscCall(PetscOptionsInt("-obs_freq", "Observation frequency", "", obs_freq, &obs_
↪freq, NULL));
  PetscCall(PetscOptionsReal("-F", "Forcing parameter", "", F, &F, NULL));
  PetscCall(PetscOptionsReal("-dt", "Time step size", "", dt, &dt, NULL));
  PetscCall(PetscOptionsReal("-obs_error", "Observation error standard deviation", "",
↪ obs_error_std, &obs_error_std, NULL));
  PetscCall(PetscOptionsReal("-ensemble_init_std", "Initial ensemble spread standard␣
↪deviation", "", ensemble_init_std, &ensemble_init_std, NULL));
  PetscCall(PetscOptionsInt("-random_seed", "Random seed for ensemble perturbations",
↪"", random_seed, &random_seed, NULL));
  PetscOptionsEnd();

  /* Validate and constrain parameters */
  PetscCall(ValidateParameters(&n, &steps, &burn, &obs_freq, &ensemble_size, &dt, &F,␣
↪&obs_error_std));
```

(continues on next page)

(continued from previous page)

```
  /* Calculate progress reporting interval (avoid division by zero) */
  progress_interval = (steps >= PROGRESS_INTERVALS) ? (steps / PROGRESS_INTERVALS) :␣
↪1;

  /* Create 1D periodic DM for state space */
  PetscCall(DMDACreate1d(PETSC_COMM_WORLD, DM_BOUNDARY_PERIODIC, n, 1, 2, NULL, &da_
↪state));
  PetscCall(DMSetFromOptions(da_state));
  PetscCall(DMSetUp(da_state));
  PetscCall(DMDASetUniformCoordinates(da_state, 0.0, (PetscReal)n, 0.0, 0.0, 0.0, 0.
↪0));

  /* Create Lorenz96 context with reusable TS object */
  PetscCall(Lorenz96ContextCreate(da_state, n, F, dt, &l95_ctx));
  PetscCall(Lorenz96ContextCreate(da_state, n, F, dt, &truth_ctx));

  /* Initialize random number generator */
  PetscCall(PetscRandomCreate(PETSC_COMM_WORLD, &rng));
  PetscCall(PetscRandomSetSeed(rng, (unsigned long)random_seed));
  PetscCall(PetscRandomSetFromOptions(rng));
  PetscCall(PetscRandomSeed(rng));

  /* Initialize state vectors */
  PetscCall(DMCreateGlobalVector(da_state, &x0));
  PetscCall(PetscRandomSetInterval(rng, -.1 * F, .1 * F));
  PetscCall(VecSetRandom(x0, rng));
  PetscCall(PetscRandomSetInterval(rng, 0, 1));

  /* Initialize truth trajectory */
  PetscCall(VecDuplicate(x0, &truth_state));
  PetscCall(VecCopy(x0, truth_state));
  PetscCall(VecDuplicate(x0, &rmse_work));

  /* Spin up truth to get onto attractor */
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Spinning up truth for %" PetscInt_FMT "␣
↪steps...\n", (PetscInt)SPINUP_STEPS));
  for (PetscInt k = 0; k < SPINUP_STEPS; k++) PetscCall(Lorenz96Step(truth_state,␣
↪truth_state, truth_ctx));

  /* Initialize observation vectors */
  PetscCall(VecDuplicate(x0, &observation));
  PetscCall(VecDuplicate(x0, &obs_noise));
  PetscCall(VecDuplicate(x0, &obs_error_var));
  PetscCall(VecSet(obs_error_var, obs_error_std * obs_error_std));

  /* Initialize ensemble statistics vectors */
  PetscCall(VecDuplicate(x0, &x_mean));
  PetscCall(VecDuplicate(x0, &x_forecast));

  /* Create identity observation matrix H */
  PetscCall(CreateIdentityObservationMatrix(n, &H));

  /* Create and configure PetscDA for ensemble data assimilation */
  PetscCall(PetscDACreate(PETSC_COMM_WORLD, &da));
  PetscCall(PetscDASetType(da, PETSCDAETKF)); /* Set ETKF type */
  PetscCall(PetscDASetSizes(da, n, n));
```

(continues on next page)

(continued from previous page)

```
  PetscCall(PetscDAEnsembleSetSize(da, ensemble_size));
  PetscCall(PetscDASetFromOptions(da));
  PetscCall(PetscDAEnsembleGetSize(da, &ensemble_size));
  PetscCall(PetscDASetUp(da));
  PetscCall(PetscDAViewFromOptions(da, NULL, "-petscda_view"));
  PetscCall(PetscDASetObsErrorVariance(da, obs_error_var));

  /* Initialize ensemble members from spun-up truth state with appropriate spread */
  PetscCall(PetscDAEnsembleInitialize(da, truth_state, ensemble_init_std, rng));

  /* Print configuration summary */
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Lorenz-96 ETKF Example\n"));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "======================\n"));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD,
                        "  State dimension       : %" PetscInt_FMT "\n"
                        "  Ensemble size         : %" PetscInt_FMT "\n"
                        "  Forcing parameter (F) : %.4f\n"
                        "  Time step (dt)        : %.4f\n"
                        "  Total steps           : %" PetscInt_FMT "\n"
                        "  Burn-in steps         : %" PetscInt_FMT "\n"
                        "  Observation frequency : %" PetscInt_FMT "\n"
                        "  Observation noise std : %.3f\n"
                        "  Ensemble init std     : %.3f\n"
                        "  Random seed           : %" PetscInt_FMT "\n\n",
                        n, ensemble_size, (double)F, (double)dt, steps, burn, obs_
↪freq, (double)obs_error_std, (double)ensemble_init_std, random_seed));

  /* Main assimilation cycle: forecast and analysis steps */
  for (step = 0; step <= steps; step++) {
    PetscReal time = step * dt;

    /* Forecast step: compute ensemble mean and forecast RMSE */
    PetscCall(PetscDAEnsembleComputeMean(da, x_mean));
    PetscCall(VecCopy(x_mean, x_forecast));
    PetscCall(ComputeRMSE(x_forecast, truth_state, rmse_work, n, &rmse_forecast));
    rmse_analysis = rmse_forecast; /* Default to forecast RMSE if no analysis */

    /* Analysis step: assimilate observations when available */
    if (step % obs_freq == 0 && step > 0) {
      /* Generate synthetic noisy observations from truth */
      PetscCall(VecSetRandomGaussian(obs_noise, rng, 0.0, obs_error_std));
      PetscCall(VecWAXPY(observation, 1.0, obs_noise, truth_state));

      /* Perform ETKF analysis with observation matrix H */
      PetscCall(PetscDAEnsembleAnalysis(da, observation, H));

      /* Compute analysis RMSE */
      PetscCall(PetscDAEnsembleComputeMean(da, x_mean));
      PetscCall(ComputeRMSE(x_mean, truth_state, rmse_work, n, &rmse_analysis));
      obs_count++;
    }

    /* Accumulate statistics after burn-in period */
    if (step >= burn) {
      sum_rmse_forecast += rmse_forecast;
      sum_rmse_analysis += rmse_analysis;
```

(continues on next page)

(continued from previous page)

```
      n_stat_steps++;
    }

    /* Progress reporting */
    if ((step % progress_interval == 0) || (step == steps) || (step == 0)) {
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Step %4" PetscInt_FMT ", time %6.3f␣
↪RMSE_forecast %.5f  RMSE_analysis %.5f%s\n", step, (double)time, (double)rmse_
↪forecast, (double)rmse_analysis, (step < burn) ? " [burn-in]" : ""));
    }

    /* Propagate ensemble and truth trajectory */
    if (step < steps) {
      PetscCall(PetscDAEnsembleForecast(da, Lorenz96Step, l95_ctx));
      PetscCall(Lorenz96Step(truth_state, truth_state, truth_ctx));
    }
  }

  /* Report final statistics */
  if (n_stat_steps > 0) {
    PetscReal avg_rmse_forecast = sum_rmse_forecast / n_stat_steps;
    PetscReal avg_rmse_analysis = sum_rmse_analysis / n_stat_steps;
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nStatistics (%" PetscInt_FMT " post-
↪burn-in steps):\n", n_stat_steps));
    PetscCall(PetscPrintf(PETSC_COMM_WORLD,
↪"================================================\n"));
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Mean RMSE (forecast) : %.5f\n",␣
↪(double)avg_rmse_forecast));
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Mean RMSE (analysis) : %.5f\n",␣
↪(double)avg_rmse_analysis));
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Observations used    : %" PetscInt_FMT
↪"\n\n", obs_count));
  } else {
    PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nWarning: No post-burn-in statistics␣
↪collected (burn >= steps)\n\n"));
  }

  /* Test VecSetRandomGaussian to verify Gaussian distribution */
  {
    Vec          test_vec;
    PetscInt     test_size = 10000; /* Large sample for statistical testing */
    PetscScalar *array;
    PetscReal    mean_target = 2.0, std_target = 1.5;
    PetscReal    sample_mean = 0.0, sample_variance = 0.0, sample_std;
    PetscReal    skewness = 0.0, kurtosis = 0.0;
    PetscInt     i;
    PetscBool    test_gaussian = PETSC_FALSE;

    PetscCall(PetscOptionsGetBool(NULL, NULL, "-test_gaussian", &test_gaussian,␣
↪NULL));

    if (test_gaussian) {
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\
↪n================================================\n"));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "Testing VecSetRandomGaussian\n"));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD,
↪"================================================\n"));
```

(continues on next page)

(continued from previous page)

```
  /* Create test vector */
  PetscCall(VecCreate(PETSC_COMM_WORLD, &test_vec));
  PetscCall(VecSetSizes(test_vec, PETSC_DECIDE, test_size));
  PetscCall(VecSetFromOptions(test_vec));

  /* Generate Gaussian random numbers */
  PetscCall(VecSetRandomGaussian(test_vec, rng, mean_target, std_target));

  /* Get array for statistical analysis */
  PetscCall(VecGetArray(test_vec, &array));

  /* Compute sample mean */
  for (i = 0; i < test_size; i++) sample_mean += PetscRealPart(array[i]);
  sample_mean /= test_size;

  /* Compute sample variance and higher moments */
  for (i = 0; i < test_size; i++) {
    PetscReal diff  = PetscRealPart(array[i]) - sample_mean;
    PetscReal diff2 = diff * diff;
    sample_variance += diff2;
    skewness += diff * diff2;
    kurtosis += diff2 * diff2;
  }
  sample_variance /= (test_size - 1);
  sample_std = PetscSqrtReal(sample_variance);

  /* Normalize skewness and kurtosis */
  skewness = (skewness / test_size) / PetscPowReal(sample_std, 3.0);
  kurtosis = (kurtosis / test_size) / PetscPowReal(sample_std, 4.0) - 3.0; /*␣
↪Excess kurtosis */

  PetscCall(VecRestoreArray(test_vec, &array));

  /* Report results */
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nTarget parameters:\n"));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Mean      : %.6f\n", (double)mean_
↪target));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Std Dev   : %.6f\n", (double)std_
↪target));

  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nSample statistics (n=%" PetscInt_FMT
↪"):\n", (PetscInt)test_size));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Mean      : %.6f (error: %.6f)\n",␣
↪(double)sample_mean, (double)PetscAbsReal(sample_mean - mean_target)));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Std Dev   : %.6f (error: %.6f)\n",␣
↪(double)sample_std, (double)PetscAbsReal(sample_std - std_target)));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Skewness  : %.6f (expected ~0 for␣
↪Gaussian)\n", (double)skewness));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Kurtosis  : %.6f (expected ~0 for␣
↪Gaussian)\n", (double)kurtosis));

  /* Statistical tests with reasonable tolerances for finite samples */
  PetscReal mean_error     = PetscAbsReal(sample_mean - mean_target);
  PetscReal std_error      = PetscAbsReal(sample_std - std_target);
  PetscReal mean_tolerance = 3.0 * std_target / PetscSqrtReal((PetscReal)test_
```

(continues on next page)

(continued from previous page)

```
↪size); /* 3-sigma rule */
      PetscReal std_tolerance  = 0.1 * std_target;                                    ␣
↪   /* 10% tolerance for std dev */
      PetscReal skew_tolerance = 0.1;                                                 ␣
↪   /* Skewness should be near 0 */
      PetscReal kurt_tolerance = 0.5;                                                 ␣
↪   /* Excess kurtosis should be near 0 */

      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nStatistical tests:\n"));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Mean test     : %s (error %.6f <␣
↪tolerance %.6f)\n", mean_error < mean_tolerance ? "PASS" : "FAIL", (double)mean_
↪error, (double)mean_tolerance));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Std dev test  : %s (error %.6f <␣
↪tolerance %.6f)\n", std_error < std_tolerance ? "PASS" : "FAIL", (double)std_error,␣
↪(double)std_tolerance));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Skewness test : %s (|skewness| %.6f
↪< tolerance %.6f)\n", PetscAbsReal(skewness) < skew_tolerance ? "PASS" : "FAIL",␣
↪(double)PetscAbsReal(skewness), (double)skew_tolerance));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "  Kurtosis test : %s (|kurtosis| %.6f
↪< tolerance %.6f)\n", PetscAbsReal(kurtosis) < kurt_tolerance ? "PASS" : "FAIL",␣
↪(double)PetscAbsReal(kurtosis), (double)kurt_tolerance));

      /* Overall test result */
      PetscBool all_pass = (mean_error < mean_tolerance) && (std_error < std_
↪tolerance) && (PetscAbsReal(skewness) < skew_tolerance) && (PetscAbsReal(kurtosis)
↪< kurt_tolerance);
      PetscCall(PetscPrintf(PETSC_COMM_WORLD, "\nOverall result: %s\n", all_pass ?
↪"PASS - Distribution is Gaussian" : "FAIL - Distribution may not be Gaussian"));
      PetscCall(PetscPrintf(PETSC_COMM_WORLD,
↪"================================================\n\n"));

      PetscCall(VecDestroy(&test_vec));
    }
  }

  /* Cleanup */
  PetscCall(MatDestroy(&H));
  PetscCall(VecDestroy(&x_forecast));
  PetscCall(VecDestroy(&x_mean));
  PetscCall(VecDestroy(&obs_error_var));
  PetscCall(VecDestroy(&obs_noise));
  PetscCall(VecDestroy(&observation));
  PetscCall(VecDestroy(&rmse_work));
  PetscCall(VecDestroy(&truth_state));
  PetscCall(VecDestroy(&x0));
  PetscCall(PetscDADestroy(&da));
  PetscCall(DMDestroy(&da_state));
  PetscCall(Lorenz96ContextDestroy(&l95_ctx));
  PetscCall(Lorenz96ContextDestroy(&truth_ctx));
  PetscCall(PetscRandomDestroy(&rng));

  PetscCall(PetscFinalize());
return 0;}
```

2.10.3 Creating a PetscDA context

Create, configure, and destroy a `PetscDA` object with the standard PETSc object lifecycle:

```
PetscDA da;
PetscCall(PetscDACreate(PETSC_COMM_WORLD, &da));
PetscCall(PetscDASetType(da, PETSCDAETKF));
PetscCall(PetscDASetSizes(da, state_size, obs_size));
PetscCall(PetscDAEnsembleSetSize(da, ensemble_size));
PetscCall(PetscDASetFromOptions(da));
PetscCall(PetscDASetUp(da));

/* ... data assimilation loop ... */

PetscCall(PetscDADestroy(&da));
```

To create a `PetscDA` instance, call `PetscDACreate()`:

```
PetscDACreate(MPI_Comm comm, PetscDA *da);
```

To choose an implementation type, call

```
PetscDASetType(PetscDA da, PetscDAType type);
```

or use the command-line option `-petscda_type <name>`; details regarding the available implementations are presented in *Implementations*.

`PetscDASetSizes()` records the global state dimension and the number of observations:

```
PetscDASetSizes(PetscDA da, PetscInt state_size, PetscInt obs_size)
```

For MPI-parallel runs where the state or observation vectors are distributed, the local partition sizes can be set explicitly with

```
PetscDASetLocalSizes(PetscDA da, PetscInt local_state_size, PetscInt local_obs_size);
```

Pass `PETSC_DECIDE` for either argument to let PETSc choose the partition automatically.

`PetscDAEnsembleSetSize()` records the number of ensemble members requested:

```
PetscDAEnsembleSetSize(PetscDA da, PetscInt ensemble_size)
```

After having set these options, call `PetscDASetFromOptions()` to apply any command-line overrides, then `PetscDASetUp()` to allocate internal storage:

```
PetscDASetFromOptions(PetscDA da);
PetscDASetUp(PetscDA da);
```

Finally, after the user is done using the DA object, destroy it with

```
PetscDADestroy(PetscDA *da);
```

2.10.4 Degrees of freedom per grid point

When the state vector is laid out on a structured grid, the number of physical degrees of freedom per grid point can be recorded with

```
PetscDASetNDOF(PetscDA da, PetscInt ndof);
PetscDAGetNDOF(PetscDA da, PetscInt *ndof);
```

This metadata is used by some implementations and viewers; the default is `1`.

2.10.5 Managing ensembles

`PetscDA` stores ensemble members as PETSc `Vec` objects and exposes convenience helpers to access them safely.

Individual ensemble members can be read or overwritten using a get/restore ownership pattern analogous to `MatDenseGetColumnVec`:

```
/* Read-only view of member i; must be paired with Restore */
PetscDAEnsembleGetMember(PetscDA da, PetscInt i, Vec *member);
PetscDAEnsembleRestoreMember(PetscDA da, PetscInt i, Vec *member);

/* Inject an externally created vector into slot i */
PetscDAEnsembleSetMember(PetscDA da, PetscInt i, Vec member);
```

`PetscDAEnsembleGetMember()` / `PetscDAEnsembleRestoreMember()` map ensemble indices to `Vec` handles that participate in PETSc's reference counting. `PetscDAEnsembleSetMember()` lets applications inject externally created vectors into specific slots, which is useful when importing state snapshots from disk or another solver component.

Ensemble statistics are computed with:

```
/* Form the sample mean across all members */
PetscDAEnsembleComputeMean(PetscDA da, Vec mean);

/* Return a tall-and-skinny Mat whose columns are the mean-subtracted,
   normalized anomalies X = (E - mean) / sqrt(m-1) */
PetscDAEnsembleComputeAnomalies(PetscDA da, Vec mean, Mat *anomalies);
```

`PetscDAEnsembleComputeAnomalies()` returns a dense `Mat` whose columns are the mean-subtracted ensemble anomalies. Many square-root filters use this matrix to construct low-rank covariance factorizations. Pass `NULL` for `mean` to have the function compute it internally.

2.10.6 Observation error

Observation-error variances (or more general descriptions) are supplied through `PetscDASetObsErrorVariance()`. The associated vector is assumed to follow the global observation ordering; `PetscDAGetObsErrorVariance()` returns the stored object for later inspection or reuse.

```
PetscDASetObsErrorVariance(PetscDA da, Vec obs_error_var);
PetscDAGetObsErrorVariance(PetscDA da, Vec *obs_error_var);
```

2.10.7 Analysis step

`PetscDAEnsembleAnalysis()` performs the assimilation step given an observation vector and a linear observation operator matrix:

```
/* observation - Vec of length P (number of observations) */
/* H           - Mat of size P x N (observation operator mapping state to observation␣
↪space) */
PetscCall(PetscDAEnsembleAnalysis(da, observation, H));
```

`H` is a `Mat` (typically sparse AIJ) that maps the N-dimensional state vector to the P-dimensional observation space: `y ≈ H*x`. `PetscDAAnalysis()` handles all ensemble reductions, gain computations, and posterior updates.

2.10.8 Forecast step

`PetscDAEnsembleForecast()` wraps the forecast step. The user supplies a function that advances a single ensemble member:

Calling sequence for model:

- `input` - the vector to be evolved, forecasted, time-stepped, or otherwise advanced
- `output` - the forecast, evolved, or time-stepped result
- `ctx` - the context for the model function

```
/* Prototype for model forecast M(x) */
PetscErrorCode ModelForecast(Vec input, Vec output, PetscCtx ctx) {
  /* Advance input by dt to produce output */
  /* (e.g., step a TS object) */
  return PETSC_SUCCESS;
}

PetscCall(PetscDAEnsembleForecast(da, ModelForecast, ctx));
```

`ModelForecast` can call PETSc time integrators (*TS: Scalable ODE and DAE Solvers*), nonlinear solvers (*SNES: Nonlinear Solvers*), or bespoke device kernels. The `PetscDA` layer orchestrates calls across the entire ensemble, issuing them in rank-local loops while ensuring that ownership and recycling semantics remain correct.

2.10.9 Inflation

Covariance inflation counteracts ensemble collapse by artificially widening the prior spread before the analysis step. The inflation factor $\rho \geq 1$ scales each anomaly so that the effective prior covariance becomes $\rho^2 P^f$. Set and retrieve the factor with

```
PetscDAEnsembleSetInflation(PetscDA da, PetscReal inflation);
PetscDAEnsembleGetInflation(PetscDA da, PetscReal *inflation);
```

or at runtime with `-petscda_ensemble_inflation <value>` (default: `1.0`, i.e. no inflation).

2.10.10 Implementations

The available `PetscDA` implementations are listed in Table *PETSc Data Assimilation Methods*. Custom types can be registered with `PetscDARegister()` and selected at runtime with `-petscda_type <name>`.

Table 2.21: PETSc Data Assimilation Methods

Method	PetscDAType	Options Name
Ensemble Transform Kalman Filter	`PETSCDAETKF`	`etkf`
Local Ensemble Transform Kalman Filter	`PETSCDALETKF`	`letkf`

ETKF

The built-in square-root ETKF (`PETSCDAETKF`, `-petscda_type etkf`) is the default implementation. It implements Algorithm 6.4 in [ABN16] using a deterministic square-root update that avoids stochastic perturbations.

The ETKF supports two factorization strategies for the reduced-space T-matrix:

```
PetscDAEnsembleSetSqrtType(PetscDA da, PetscDASqrtType type);
PetscDAEnsembleGetSqrtType(PetscDA da, PetscDASqrtType *type);
```

Table 2.22: ETKF square-root types

PetscDASqrtType	Options string	Notes
`PETSCDA_SQRT_CHOL`	`cholesky`	$O(n^3/3)$; preferred when the reduced-space matrix is positive definite
`PETSCDA_SQRT_EIGE`	`eigen`	More robust for semi-definite matrices; handles small negative eigenvalues from round-off

Select at runtime with `-petscda_ensemble_sqrt_type {cholesky,eigen}` (default: `eigen`).

LETKF

The Local ETKF (`PETSCDALETKF`, `-petscda_type letkf`) performs the analysis update locally around each grid point, enabling scalable assimilation on large domains by avoiding the global ensemble covariance matrix. LETKF-specific configuration:

```
/* Number of observations associated with each grid vertex (default: 9) */
PetscDALETKFSetObsPerVertex(PetscDA da, PetscInt n_obs_vertex);
PetscDALETKFGetObsPerVertex(PetscDA da, PetscInt *n_obs_vertex);

/* Localization weight matrix Q (N x P) and observation operator matrix H (P x N) */
PetscDALETKFSetLocalization(PetscDA da, Mat Q, Mat H);
```

Set the observation count at runtime with `-petscda_letkf_obs_per_vertex <n>` (default: `9`).

2.10.11 Command-line options

The `PetscDA` object obeys standard PETSc options parsing. Commonly used switches include:

- `-petscda_type <name>` – select a registered `PetscDA` implementation (`etkf`, `letkf`).
- `-petscda_ensemble_inflation <value>` – set the covariance inflation factor (default: `1.0`).
- `-petscda_ensemble_sqrt_type {cholesky,eigen}` – select the T-matrix square-root algorithm for ETKF (default: `eigen`).
- `-petscda_letkf_obs_per_vertex <n>` – set the number of observations per grid vertex for LETKF (default: `9`).
- `-petscda_view` – inspect ensemble metadata and internal sizes.

Because `PetscDA` participates in the PETSc object registry, any prefix applied with `PetscDASetOptionsPrefix()` scopes these options.

2.10.12 Viewing and monitoring

`PetscDAView()` and `PetscDAViewFromOptions()` expose ensemble sizing, observation dimensions, and implementation-specific diagnostics . Views can be directed to ASCII, HDF5, or custom `PetscViewer` targets, enabling lightweight instrumentation of assimilation experiments. For advanced profiling, the `PetscDA` package registers with PETSc's logging infrastructure via `PetscDAInitializePackage()`/`PetscDAFinalizePackage()`, so standard `-log_view` outputs will include assimilation breakdown.

2.10.13 Further reading

- *TS: Scalable ODE and DAE Solvers* discusses PETSc time integrators that can supply the forecast operator passed to `PetscDAForecast()`.
- *Vectors and Parallel Data* documents vector assembly and parallel data management for the state and observation spaces.
- *SNES: Nonlinear Solvers* outlines nonlinear solvers that often participate in observation or model operators.
- *DM Basics* provides background on distributed mesh infrastructure that can coexist with PetscDA-managed ensembles.

CHAPTER

THREE

DM: INTERFACING BETWEEN SOLVERS AND MODELS/DISCRETIZATIONS

3.1 DM Basics

The previous chapters have focused on the core numerical solvers in PETSc. However, numerical solvers without efficient ways (in both human and machine time) of connecting the solvers to the mathematical models and discretizations, including grids (or meshes) that people wish to build their simulations on, will not get widely used. Thus PETSc provides a set of abstractions represented by the `DM` object to provide a powerful, comprehensive mechanism for translating the problem specification of a model and its discretization to the language and API of solvers. `DM` is an orphan initialism or orphan acronym, the letters have no meaning and never did.

Some of the model classes `DM` currently supports are PDEs on structured and staggered grids with finite difference methods (`DMDA` – *DMDA - Creating vectors for structured grids* and `DMSTAG` – *DMSTAG: Staggered, Structured Grid*), PDEs on unstructured grids with finite element and finite volume methods (`DMPLEX` – *DMPlex: Unstructured Grids*), PDEs on quad and octree-grids (`DMFOREST`), models on networks (graphs) such as the power grid or river networks (`DMNETWORK` – *Networks*), and particle-in-cell simulations (`DMSWARM`).

In previous chapters, we have demonstrated some simple usage of `DM` to provide the input for the solvers. In this chapter, and those that follow, we will dive deep into the capabilities of `DM`.

It is possible to create a `DM` with

```
DM dm;
DMCreate(MPI_Comm comm, DM *dm);
DMSetType(DM dm, DMType type);
```

but more commonly, a `DM` is created with a type-specific constructor; the construction process for each type of `DM` is discussed in the sections on each `DMType`. This chapter focuses on commonalities between all the `DM` so we assume the `DM` already exists and we wish to work with it.

As discussed earlier, a `DM` can construct vectors and matrices appropriate for a model and discretization and provide the mapping between the global and local vector representations.

```
DMCreateLocalVector(DM dm,Vec *l);
DMCreateGlobalVector(DM dm,Vec *g);
DMGlobalToLocal(dm,g,l,INSERT_VALUES);
DMLocalToGlobal(dm,l,g,ADD_VALUES);
DMCreateMatrix(dm,Mat *m);
```

The matrices produced may support `MatSetValuesLocal()` allowing one to work with the local numbering on each MPI rank. For `DMDA` one can also use `MatSetValuesStencil()` and for `DMSTAG` with `DMStagMatSetValuesStencil()`.

A given DM can be refined for certain DMTypes with DMRefine() or coarsened with DMCoarsen(). Mappings between DMs may be obtained with routines such as DMCreateInterpolation(), DMCreateRestriction() and DMCreateInjection().

One attaches a DM to a PETSc solver object, KSP, SNES, TS, or Tao with

```
KSPSetDM(KSP ksp,DM dm);
SNESSetDM(SNES snes,DM dm);
TSSetDM(TS ts,DM dm);
```

Once the DM is attached, the solver can utilize it to create and process much of the data that the solver needs to set up and implement its solve. For example, with PCMG simply providing a DM can allow it to create all the data structures needed to run geometric multigrid on your problem.

SNES Tutorial ex19 demonstrates how this may be done with DMDA.

See *DM Commonalities* for an advanced discussion of the commonalities between the various DM. That material should be read after having read the material below for each of the DM.

3.2 PetscSection: Connecting Grids to Data

The strongest links between solvers and discretizations are

- the relationship between the layout of data over a mesh (or similar structure) and the data layout in arrays and Vec used for computation,
- data partitioning, and
- ordering of data.

To enable modularity, we encode the operations above in simple data structures that can be understood by the linear algebraic and solver components of PETSc (Vec, Mat, KSP, PC, SNES, TS, Tao, PetscRegressor) without explicit reference to the mesh (topology) or discretization (analysis).

While PetscSection is currently only employed for DMPlex, DMForest, and DMNetwork mesh descriptions, much of its operation is general enough to be utilized for other types of discretizations. This section will explain the basic concepts of a PetscSection that are generalizable to other mesh descriptions.

3.2.1 General concept

Specific entries (or collections of entries) in a Vec (or a simple array) can be associated with a "location" on a mesh (or other types of data structure) using the PetscSection object. A **point** is a PetscInt that serves as an abstract "index" into arrays from iterable sets, such as k-cells in a mesh. Other iterable set examples can be as simple as the points of a finite difference grid, or cells of a finite volume grid, or as complex as the topological entities of an unstructured mesh (cells, faces, edges, and vertices).

At it's most basic, a PetscSection is a mapping between the mesh points and a tuple (ndof, offset), where ndof is the number of values stored at that mesh point and offset is the location in the array of that data. So given the tuple for a mesh point, its data can be accessed by array[offset + d], where d in [0, ndof) is the dof to access.

Charts: Defining mesh points

The mesh points for a `PetscSection` must be contiguously numbered and are defined to be in some range [pStart, pEnd), which is called a **chart**. The chart of a `PetscSection` is set via `PetscSectionSetChart()`. Note that even though the mesh points must be contiguously numbered, the indexes into the array (defined by each `(ndof, offset)` tuple) associated with the `PetscSection` need not be. In other words, there may be elements in the array that are not associated with any mesh points, though this is not often the case.

Defining the (ndof, offset) tuple

Defining the `(ndof, offset)` tuple for each mesh point generally first starts with setting the `ndof` for each point, which is done using `PetscSectionSetDof()`. This associates a set of degrees of freedom (dof), (a small space $\{e_k\}$ $0 < k < ndof$), with every point. If `ndof` is not set for a mesh point, it is assumed to be 0.

The offset for each mesh point is usually set automatically by `PetscSectionSetUp()`. This will concatenate each mesh point's dofs together in the order of the mesh points. This concatenation can be done in a different order by setting a permutation, which is described in *Permutation: Changing the order of array data*.

Alternatively, the offset for each mesh point can be set manually by `PetscSectionSetOffset()`, though this is not commonly needed.

Once the tuples are created, the `PetscSection` is ready to use.

Basic Setup Example

To summarize, the sequence for constructing a basic `PetscSection` is the following:

1. Specify the range of points, or chart, with `PetscSectionSetChart()`.
2. Specify the number of dofs per point with `PetscSectionSetDof()`. Any values not set will be zero.
3. Set up the `PetscSection` with `PetscSectionSetUp()`.

3.2.2 Multiple Fields

In many discretizations, it is useful to differentiate between different kinds of dofs present on a mesh. For example, a dof attached to a cell point might represent pressure while dofs on vertices might represent velocity or displacement. A `PetscSection` can represent this additional structure with what are called **fields**. **Fields** are indexed contiguously from `[0, num_fields)`. To set the number of fields for a `PetscSection`, call `PetscSectionSetNumFields()`.

Internally, each field is stored in a separate `PetscSection`. In fact, all the concepts and functions presented in *General concept* were actually applied onto the **default field**, which is indexed as `0`. The fields inherit the same chart as the "parent" `PetscSection`.

Setting Up Multiple Fields

Setup for a `PetscSection` with multiple fields is nearly identical to setup for a single field.

The sequence for constructing such a `PetscSection` is the following:

1. Specify the range of points, or chart, with `PetscSectionSetChart()`. All fields share the same chart.
2. Specify the number of fields with `PetscSectionSetNumFields()`.
3. Set the number of dof for each point on each field with `PetscSectionSetFieldDof()`. Any values not set will be zero.
4. Set the **total** number of dof for each point with `PetscSectionSetDof()`. Thus value must be greater than or equal to the sum of the values set with `PetscSectionSetFieldDof()` at that point. Again, values not set will be zero.
5. Set up the `PetscSection` with `PetscSectionSetUp()`.

Point Major or Field Major

A `PetscSection` with one field and and offsets set in `PetscSectionSetUp()` may be thought of as defining a two dimensional array indexed by point in the outer dimension with a variable length inner dimension indexed by the dof at that point: v[pStart <= $point$ < pEnd][0 <= dof < ndof][1].

With multiple fields, this array is now three dimensional, with the outer dimensions being both indexed by mesh points and field points. Thus, there is a choice on whether to index by points first, or by fields first. In other words, will the array be laid out in a point-major or field-major fashion.

Point-major ordering corresponds to v[pStart <= $point$ < pEnd][0 <= $field$ < num_fields][0 <= dof < ndof]. All the dofs for each mesh point are stored contiguously, meaning the fields are **interlaced**. Field-major ordering corresponds to v[0 <= $field$ < num_fields][pStart <= $point$ < pEnd][0 <= dof < ndof]. The all the dofs for each field are stored contiguously, meaning the points are **interlaced**.

Consider a `PetscSection` with 2 fields and 2 points (from 0 to 2). Let the 0th field have `ndof=1` for each point and the 1st field have `ndof=2` for each point. Denote each array entry (p_i, f_i, d_i) for p_i being the ith point, f_i being the ith field, and d_i being the ith dof.

Point-major order would result in:

$$[(p_0, f_0, d_0), (p_0, f_1, d_0), (p_0, f_1, d_1),$$
$$(p_1, f_0, d_0), (p_1, f_1, d_0), (p_1, f_1, d_1)]$$

Conversely, field-major ordering would result in:

$$[(p_0, f_0, d_0), (p_1, f_0, d_0),$$
$$(p_0, f_1, d_0), (p_0, f_1, d_1), (p_1, f_1, d_0), (p_1, f_1, d_1)]$$

Note that dofs are always contiguous, regardless of the outer dimensional ordering.

Setting the which ordering is done with `PetscSectionSetPointMajor()`, where `PETSC_TRUE` sets point-major and `PETSC_FALSE` sets field major.

NOTE: The current default is for point-major, and many operations on `DMPlex` will only work with this ordering. Field-major ordering is provided mainly for compatibility with external packages, such as LibMesh.

[1] 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. Conversely, `PetscSection` associates multiple unit vectors with every mesh point (one for each dof) and divides the mesh points between processes using a `PetscSF` to define the distribution.

3.2.3 Working with data

Once a `PetscSection` has been created one can use `PetscSectionGetStorageSize()` to determine the total number of entries that can be stored in an array or `Vec` accessible by the `PetscSection`. This is most often used when creating a new `Vec` for a `PetscSection` such as:

```
PetscSectionGetStorageSize(s, &n);
VecSetSizes(localVec, n, PETSC_DETERMINE);
VecSetFromOptions(localVec);
```

The memory locations in the associated array are found using an **offset** which can be obtained with:

Single-field `PetscSection`:

```
PetscSectionGetOffset(PetscSection, PetscInt point, PetscInt &offset);
```

Multi-field `PetscSection`:

```
PetscSectionGetFieldOffset(PetscSection, PetscInt point, PetscInt field, PetscInt &
↪offset);
```

The value in the array is then accessed with `array[offset + d]`, where `d` in `[0, ndof)` is the dof to access.

3.2.4 Global Sections: Constrained and Distributed Data

To handle distributed data and data with constraints, we use a pair of `PetscSections` called the `localSection` and `globalSection`. Their use for each is described below.

Distributed Data

`PetscSection` can also be applied to distributed problems as well. This is done using the same local/global system described in *Local/global vectors and communicating between vectors*. To do this, we introduce three new concepts; a `localSection`, `globalSection`, `pointSF`, and `sectionSF`.

Assume the mesh points of the "global" mesh are partitioned among processes and that some mesh points are shared between multiple processes (i.e there is an overlap in the partitions). The shared mesh points define the ghost/halo points needed in many PDE problems. For each shared mesh point, appoint one process to be the owner of that mesh point. To describe this parallel mesh point layout, we use a `PetscSF` and call it the `pointSF`. The `pointSF` describes which processes "own" which mesh points and which process is the owner of each shared mesh point.

Next, for each process define a `PetscSection` that describes the mapping between that process's partition (including shared mesh points) and the data stored on it and call it the `localSection`. The `localSection` describes the layout of the local vector. To generate the `globalSection` we use `PetscSectionCreateGlobalSection()`, which takes the `localSection` and `pointSF` as inputs. The global section returns $-(dof+1)$ for the number of dofs on an unowned (ghost) point, and traditionally $-(off+1)$ for its offset on the owning process. This behavior of the offsets is controlled via an argument to `PetscSectionCreateGlobalSection()`. The `globalSection` can be used to create global vectors, just as the local section is used to create local vectors.

To perform the global-to-local and local-to-global communication, we define `sectionSF` to be the `PetscSF` describing the mapping between the local and global vectors. This is generated via `PetscSFSetGraphSection()`. Using `PetscSFBcastBegin()` will send data from the global vector to the local vector, while `PetscSFReduceBegin()` will send data from the local vector to the global vector.

If using DM, this entire process is done automatically. The localSection, globalSection, pointSF, and sectionSF on a DM can be obtained via DMGetLocalSection(), DMGetGlobalSection(), DMGetPointSF(), and DMGetSectionSF(), respectively. Additionally, communication from global to local vectors and vice versa can be done via DMGlobalToLocal() and DMLocalToGlobal() as described in *Local/global vectors and communicating between vectors*. Note that not all DM types use this system, such as DMDA (see *DMDA - Creating vectors for structured grids*).

Constrained Data

In addition to describing parallel data, the localSection/globalSection pair can be used to describe *constrained* dofs These constraints usually represent essential (Dirichlet) boundary conditions, or algebraic constraints. They are dofs that have a given fixed value, so they are present in local vectors for finite element/volume assembly or finite difference stencil application purposes, but generally absent from global vectors since they are not unknowns in the algebraic solves.

Constraints should be indicated in the localSection. Use PetscSectionSetConstraintDof() to set the number of constrained dofs for a given point, and PetscSectionSetConstraintIndices() to indicate which dofs on the given point are constrained. This must be done before PetscSectionCreateGlobalSection() is called to create the globalSection.

Note that it is possible to have constraints set in a localSection, but have the globalSection be generated to include those constraints. This is useful when doing some form of post-processing of a solution where you want to access all data (see DMGetOutputDM() for example). See PetscSectionCreateGlobalSection() for more details on this.

3.2.5 Permutation: Changing the order of array data

By default, when PetscSectionSetUp() is called, the data laid out in the associated array is assumed to be in the same order of the grid points. For example, the DoFs associated with grid point 0 appear directly before grid point 1, which appears before grid point 2, etc.

It may be desired to have a different the ordering of data in the array than the order of grid points defined by a section. For example, one may want grid points associated with the boundary of a domain to appear before points associated with the interior of the domain.

This can be accomplished by either changing the indexes of the grid points themselves, or by informing the section of the change in array ordering. Either method uses an IS to define the permutation.

To change the indices of the grid points, call PetscSectionPermute() to generate a new PetscSection with the desired grid point permutation.

To just change the array layout without changing the grid point indexing, call PetscSectionSetPermutation(). This must be called before PetscSectionSetUp() and will only affect the calculation of the offsets for each grid point.

3.2.6 DMPlex Specific Functionality: Obtaining data from the array

A vanilla `PetscSection` (what's been described up till now) gives a relatively naive perspective on the underlying data; it doesn't describe how DoFs attached to a single grid point are ordered or how different grid points relate to each other. A `PetscSection` can store and use this extra information in the form of **closures**, **symmetries**, and **closure permutations**. These features currently target `DMPlex` and other unstructured grid descriptions. A description of those features will be left to *DMPlex: Unstructured Grids*.

3.3 DMPlex: Unstructured Grids

This chapter introduces the `DMPLEX` subclass of `DM`, which allows the user to handle unstructured grids (or meshes) 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.

3.3.1 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 the same way. This allows the interface to be 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 (vertices, edges, faces, and cells) are treated as *points* (or mesh entities), which are each identified by a `PetscInt`. 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). This structure is known as 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. 3.1,

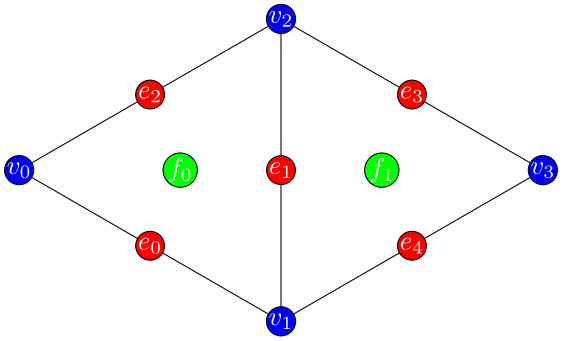


Fig. 3.1: A 2D doublet mesh, two triangles sharing an edge.

which can also be represented as the DAG in Fig. 3.2.

To use the PETSc API, we 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

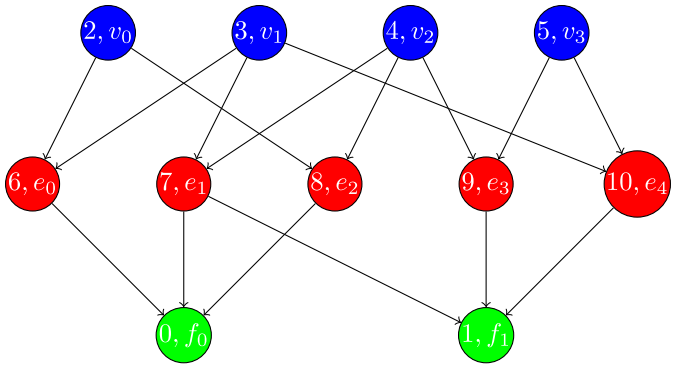


Fig. 3.2: The Hasse diagram for our 2D doublet mesh, expressed as a DAG.

faces, vertices, and then edges. In terms of the labels in Fig. 3.1, these numberings are

$$f_0 \mapsto 0, f_1 \mapsto 1,$$
$$v_0 \mapsto 2, v_1 \mapsto 3, v_2 \mapsto 4, v_3 \mapsto 5,$$
$$e_0 \mapsto 6, e_1 \mapsto 7, e_2 \mapsto 8, e_3 \mapsto 9, e_4 \mapsto 10$$

First, we declare the set of points present in a mesh,

```
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 provide sizes,

```
/* DMPlexSetConeSize(dm, point, number of points that cover the point); */
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 (recall each point is an integer that represents a single geometric entity, a cell, face, edge, or vertex),

```
/* DMPlexSetCone(dm, point, [points that cover the point]); */
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 providing the dual relation, using `DMPlexSetSupportSize()` and `DMPlexSetSupport()`, but this can be calculated automatically using the provided `DMPlexSetConeSize()` and `DMPlexSetCone()` information and then calling

```
DMPlexSymmetrize(dm);
```

The "symmetrization" is in the sense of the DAG. Each point knows its covering (cone) and each point knows what it covers (support). Note that when using automatic symmetrization, cones will be ordered but supports will not. The user can enforce an ordering on supports by rewriting them after symmetrization using `DMPlexSetSupport()`.

In order to support efficient queries, we construct fast search structures and indices for the different types of points using

```
DMPlexStratify(dm);
```

3.3.2 Grid Point Orientations

TODO: fill this out with regard to orientations.

Should probably reference Hapla and summarize what's there.

3.3.3 Dealing with Periodicity

Plex allows you to represent periodic domains is two ways. Using the default scheme, periodic topology can be represented directly. This ensures that all topological queries can be satisfied, but then care must be taken in representing functions over the mesh, such as the coordinates. The second method is to use a non-periodic topology, but connect certain mesh points using the local-to-global map for that DM. This allows a more general set of mappings to be implemented, such as partial twists, but topological queries on the periodic boundary cease to function.

For the default scheme, a call to `DMLocalizeCoordinates()` (which usually happens automatically on mesh creation) creates a second, discontinuous coordinate field. These values can be accessed using `DMGetCellCoordinates()` and `DMGetCellCoordinatesLocal()`. Plex provides a convenience method, `DMPlexGetCellCoordinates()`, that extracts cell coordinates correctly, depending on the periodicity of the mesh. An example of its use is shown below:

```
const PetscScalar *array;
PetscScalar       *coords = NULL;
PetscInt           numCoords;
PetscBool          isDG;

PetscCall(DMPlexGetCellCoordinates(dm, cell, &isDG, &numCoords, &array, &coords));
for (PetscInt cc = 0; cc < numCoords/dim; ++cc) {
  if (cc > 0) PetscCall(PetscPrintf(PETSC_COMM_SELF, " -- "));
  PetscCall(PetscPrintf(PETSC_COMM_SELF, "("));
  for (PetscInt d = 0; d < dim; ++d) {
    if (d > 0) PetscCall(PetscPrintf(PETSC_COMM_SELF, ", "));
    PetscCall(PetscPrintf(PETSC_COMM_SELF, "%g", (double)PetscRealPart(coords[cc *␣
↪dim + d])));
  }
  PetscCall(PetscPrintf(PETSC_COMM_SELF, ")"));
}
PetscCall(PetscPrintf(PETSC_COMM_SELF, "\n"));
PetscCall(DMPlexRestoreCellCoordinates(dm, cell, &isDG, &numCoords, &array, &coords));
```

3.3.4 Connecting Grids to Data Using PetscSection:

A `PetscSection` is used to describe the connection between the grid and data associated with the grid. Specifically, it assigns a number of dofs to each mesh entity on the DAG. See *PetscSection: Connecting Grids to Data* for more details. Using the mesh from Fig. 3.1, we provide an example of creating a `PetscSection` for a single field. We can lay out data for a continuous Galerkin P_3 finite element method,

```
PetscInt pStart, pEnd, cStart, cEnd, c, vStart, vEnd, v, eStart, eEnd, e;

DMPlexGetChart(dm, &pStart, &pEnd);
DMPlexGetHeightStratum(dm, 0, &cStart, &cEnd);   // cells
DMPlexGetHeightStratum(dm, 1, &eStart, &eEnd);   // edges
DMPlexGetHeightStratum(dm, 2, &vStart, &vEnd);   // vertices, equivalent to␣
↪DMPlexGetDepthStratum(dm, 0, &vStart, &vEnd);
PetscSectionSetChart(s, pStart, pEnd);
for(c = cStart; c < cEnd; ++c)
    PetscSectionSetDof(s, c, 1);
for(v = vStart; v < vEnd; ++v)
    PetscSectionSetDof(s, v, 1);
for(e = eStart; e < eEnd; ++e)
    PetscSectionSetDof(s, e, 2); // two dof on each edge
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)` returns the depth of the DAG, hence `DMPlexGetDepthStratum(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,

```
PetscSectionGetStorageSize(s, &n);
VecSetSizes(localVec, n, PETSC_DETERMINE);
VecSetFromOptions(localVec);
```

When working with `DMPLEX` and `PetscFE` (see below) one can simply get the sections (and related vectors) with

```
DMSetLocalSection(dm, s);
DMGetLocalVector(dm, &localVec);
DMGetGlobalVector(dm, &globalVec);
```

DMPlex-specific PetscSection Features:

The following features are built into `PetscSection`. However, their usage and purpose is best understood through `DMPLEX`.

Closure:

The `DMPlexGetTransitiveClosure()` method returns all points in the topological closure of a given point. This is equivalent to the transitive closure in the Hasse Diagram, which is a DAG. The default transitive closure order is breadth-first over the DAG. This means the k-cell itself, then its cone $(k-1)$-faces, then their cone $(k-2)$-faces until we get down to the vertices.

This provides an ordering on points in the closure. If we map each point to a set of dofs using a section, then we have an order on the dofs in the closure of a point. If a closure permutation is applied, then we permute the entire dof closure after it has been generated.

`DMPlexVecGetClosure()` can extract values associated with the closure of a given point from a local vector, using the layout in the local section, in the closure order. This means the data is ordered first by section field, then by closure point, and finally by dof on that point. The closure order can be directly changed using `PetscSectionSetClosurePermutation()`, but `DMPLEX` has helper functions for common reorderings such as `DMPlexSetClosurePermutationTensor()` and `DMPlexSetClosurePermutationLexicographic()`.

If multiple dofs are associated to a point, we must order them in some way. The local section has a canonical order for these dofs, but if the point has an orientation in the closure, we must reorder those dofs to respect that orientation. See *Symmetries: Accessing data from different orientations* for a discussion of `PetscSectionSym` which implements these transformations.

Closure information can be attached to a `PetscSection` to allow for more efficient closure information queries. This information can either be set directly using the `PetscSectionSetClosureIndex()` method, or it can be generated automatically for a `DMPLEX` via `DMPlexCreateClosureIndex()`.

Symmetries: Accessing data from different orientations

While mesh point orientation information specifies how one mesh point is oriented with respect to another, it does not describe how the dofs associated with that mesh point should be permuted for that orientation. This information is supplied via a `PetscSectionSym` object that is attached to the `PetscSection`. It is a map from orientation numbers to permutation indices for the dofs on that point. Orientation 0 means the canonical orientation, and negative orientations indicate a reversal. Each integer maps to a member of the dihedral group for the cell. More information on orientations is given in *Grid Point Orientations*. Generally the setup and usage of this information is handled automatically by PETSc during setup of a Plex using `PetscFE`.

Closure Permutation:

A permutation of the dof closure of a k-cell may be specified. This allows data to be returned in an order that might be more efficiently processed than the default (breadth-first search) ordering. For example, for tensor cells such as quadrilaterals, closure data can be permuted to lexicographic order (i.e. a tensor-product ordering). This is most commonly done via `DMPlexSetClosurePermutationTensor()`. Custom permutations can be set using `PetscSectionSetClosurePermutation()`.

3.3.5 Data Layout using DMPLEX and PetscFE

A DM can automatically create the local section if given a description of the discretization, for example using a PetscFE object. We demonstrate this by creating a PetscFE that can be configured from the command line. It is a single, scalar field, and is added to the DM using DMSetField(). When a local or global vector is requested, the DM builds the local and global sections automatically.

```
DMPlexIsSimplex(dm, &simplex);
PetscFECreateDefault(PETSC_COMM_SELF, dim, 1, simplex, NULL, -1, &fe);
DMSetField(dm, 0, NULL, (PetscObject) fe);
DMCreateDS(dm);
```

Here the call to DMSetField() declares the discretization will have one field with the integer label 0 that has one degree of freedom at each point on the DMPlex. To get the P_3 section above, we can either give the option -petscspace_degree 3, or call PetscFECreateLagrange() and set the degree directly.

Suppose we have a P_3 triangle, and call DMPlexVecGetClosure(). First it will retrieve the closure points and orientations

$$\{c_0, o_{c0}, e_0, o_{e0}, e_1, o_{e1}, e_2, o_{e2}, v_0, 0, v_1, 0, v_2, 0\}$$

since vertices only have 0 orientation. Suppose that $o_{e0} = 0$, $o_{e1} = -1$ and $o_{e2} = 0$. Now we extract that local vector values in this order, using the local section. Suppose that the section map is

point		size	offset
c_0	$\to$	1	0
e_0	$\to$	2	6
e_1	$\to$	2	8
e_2	$\to$	2	10
v_0	$\to$	1	2
v_1	$\to$	1	3
v_2	$\to$	1	4

where the offset is into a local vector, not an array over the closure. Then we will have array values

$$\{a_0, a_6, a_7, a_9, a_8, a_{10}, a_{11}, a_2, a_3, a_4\}$$

Notice that the values for e_1 are reversed since it had orientation -1.

3.3.6 Partitioning and Ordering

In the same way as MatPartitioning or MatGetOrdering(), give the results of a partitioning or ordering of a graph defined by a sparse matrix, PetscPartitionerDMPlexPartition or DMPlexPermute are encoded in an IS. However, the graph is not the adjacency graph of the matrix 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 represents 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[1] 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`.

3.3.7 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 current 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 from mesh traversal routines to data extraction routines and back. In this way, the `PetscSection` which structures the data inside a `Vec` does not need to know anything about the mesh inside a `DMPLEX`.

The most common mesh traversal is the transitive closure of a point, which is exactly the transitive closure of a point in the DAG using the covering relation. In other words, the transitive closure consists of all points that cover the given point (generally a cell) plus all points that cover those points, etc. So in 2d the transitive closure for a cell consists of edges and vertices while in 3d it consists of faces, edges, and vertices. Note that this closure can be calculated orienting the arrows in either direction. For example, in a finite element calculation, we calculate an integral over each element, and then sum up the contributions to the basis function coefficients. The closure of the element can be expressed discretely as the transitive closure of the element point in the mesh DAG, where each point also has an orientation. Then we can retrieve the data using `PetscSection` methods,

```
PetscScalar *a;
PetscInt     numPoints, *points = NULL, p;

VecGetArrayRead(u,&a);
DMPlexGetTransitiveClosure(dm,cell,PETSC_TRUE,&numPoints,&points);
for (p = 0; p <= numPoints*2; p += 2) {
  PetscInt dof, off, d;
```

(continues on next page)

[1] In three dimensions, the boundary of a cell (sometimes called an element) is its faces, the boundary of a face is its edges and the boundary of an edge is the two vertices.

(continued from previous page)

```
  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);
VecRestoreArrayRead(u, &a);
```

This operation is so common that we have built a convenience method around it which returns the values in a contiguous array, correctly taking into account the orientations of various mesh points:

```
const PetscScalar *values;
PetscInt           csize;

DMPlexVecGetClosure(dm, section, u, cell, &csize, &values);
// Do integral in quadrature loop putting the result into r[]
DMPlexVecRestoreClosure(dm, section, u, cell, &csize, &values);
DMPlexVecSetClosure(dm, section, residual, cell, &r, ADD_VALUES);
```

A simple example of this kind of calculation is in `DMPlexComputeL2Diff_Plex()` (source). Note that there is no restriction on the type of cell or dimension of the mesh in the code above, so it will work for polyhedral cells, hybrid meshes, and meshes of any dimension, without change. We can also reverse the covering relation, so that the code works for finite volume methods where we want the data from neighboring cells for each face:

```
PetscScalar *a;
PetscInt     points[2*2], numPoints, p, dofA, offA, dofB, offB;

VecGetArray(u,  &a);
DMPlexGetTransitiveClosure(dm, cell, PETSC_FALSE, &numPoints, &points);
assert(numPoints == 2);
PetscSectionGetDof(section, points[0*2], &dofA);
PetscSectionGetDof(section, points[1*2], &dofB);
assert(dofA == dofB);
PetscSectionGetOffset(section, points[0*2], &offA);
PetscSectionGetOffset(section, points[1*2], &offB);
myfunc(a[offA], a[offB]);
VecRestoreArray(u, &a);
```

This kind of calculation is used in TS Tutorial ex11.

3.3.8 Saving and Loading DMPlex Data with HDF5

PETSc allows users to save/load DMPLEXs representing meshes, PetscSections representing data layouts on the meshes, and Vecs defined on the data layouts to/from an HDF5 file in parallel, where one can use different number of processes for saving and for loading.

Saving

The simplest way to save DM data is to use options for configuration. This requires only the code

```
DMViewFromOptions(dm, NULL, "-dm_view");
VecViewFromOptions(vec, NULL, "-vec_view")
```

along with the command line options

```
$ ./myprog -dm_view hdf5:myprog.h5 -vec_view hdf5:myprog.h5::append
```

Options prefixes can be used to separately control the saving and loading of various fields. However, the user can have finer grained control by explicitly creating the PETSc objects involved. To save data to "example.h5" file, we can first create a PetscViewer of type PETSCVIEWERHDF5 in FILE_MODE_WRITE mode as:

```
PetscViewer  viewer;

PetscViewerHDF5Open(PETSC_COMM_WORLD, "example.h5", FILE_MODE_WRITE, &viewer);
```

As dm is a DMPLEX object representing a mesh, we first give it a *mesh name*, "plexA", and save it as:

```
PetscObjectSetName((PetscObject)dm, "plexA");
PetscViewerPushFormat(viewer, PETSC_VIEWER_HDF5_PETSC);
DMView(dm, viewer);
PetscViewerPopFormat(viewer);
```

The DMView() call is shorthand for the following sequence

```
DMPlexTopologyView(dm, viewer);
DMPlexCoordinatesView(dm, viewer);
DMPlexLabelsView(dm, viewer);
```

If the *mesh name* is not explicitly set, the default name is used. In the above PETSC_VIEWER_HDF5_PETSC format was used to save the entire representation of the mesh. This format also saves global point numbers attached to the mesh points. In this example the set of all global point numbers is $X = [0, 11)$.

The data layout, s, needs to be wrapped in a DM object for it to be saved. Here, we create the wrapping DM, sdm, with DMClone(), give it a *dm name*, "dmA", attach s to sdm, and save it as:

```
DMClone(dm, &sdm);
PetscObjectSetName((PetscObject)sdm, "dmA");
DMSetLocalSection(sdm, s);
DMPlexSectionView(dm, viewer, sdm);
```

If the *dm name* is not explicitly set, the default name is to be used. In the above, instead of using DMClone(), one could also create a new DMSHELL object to attach s to. The first argument of DMPlexSectionView() is a DMPLEX object that represents the mesh, and the third argument is a DM object that carries the data layout that we would like to save. They are, in general, two different objects, and the former carries a *mesh name*, while the latter carries a *dm name*. These names are used to construct a group structure in the

HDF5 file. Note that the data layout points are associated with the mesh points, so each of them can also be tagged with a global point number in X; `DMPlexSectionView()` saves these tags along with the data layout itself, so that, when the mesh and the data layout are loaded separately later, one can associate the points in the former with those in the latter by comparing their global point numbers.

We now create a local vector associated with `sdm`, e.g., as:

```
Vec  vec;

DMGetLocalVector(sdm, &vec);
```

After setting values of `vec`, we name it "vecA" and save it as:

```
PetscObjectSetName((PetscObject)vec, "vecA");
DMPlexLocalVectorView(dm, viewer, sdm, vec);
```

A global vector can be saved in the exact same way with trivial changes.

After saving, we destroy the `PetscViewer` with:

```
PetscViewerDestroy(&viewer);
```

The output file "example.h5" now looks like the following:

```
$ h5dump --contents example.h5
HDF5 "example.h5" {
FILE_CONTENTS {
 group      /
 group      /topologies
 group      /topologies/plexA
 group      /topologies/plexA/dms
 group      /topologies/plexA/dms/dmA
 dataset    /topologies/plexA/dms/dmA/order
 group      /topologies/plexA/dms/dmA/section
 dataset    /topologies/plexA/dms/dmA/section/atlasDof
 dataset    /topologies/plexA/dms/dmA/section/atlasOff
 group      /topologies/plexA/dms/dmA/vecs
 group      /topologies/plexA/dms/dmA/vecs/vecA
 dataset    /topologies/plexA/dms/dmA/vecs/vecA/vecA
 group      /topologies/plexA/labels
 group      /topologies/plexA/topology
 dataset    /topologies/plexA/topology/cells
 dataset    /topologies/plexA/topology/cones
 dataset    /topologies/plexA/topology/order
 dataset    /topologies/plexA/topology/orientation
 }
}
```

Saving in the new parallel HDF5 format

Since PETSc 3.19, we offer a new format which supports parallel loading. To write in this format, you currently need to specify it explicitly using the option

```
-dm_plex_view_hdf5_storage_version 3.0.0
```

The storage version is stored in the file and set automatically when loading (described below). You can check the storage version of the HDF5 file with

```
$ h5dump -a /dmplex_storage_version example.h5
```

To allow for simple and efficient implementation, and good load balancing, the 3.0.0 format changes the way the mesh topology is stored. Different strata (sets of mesh entities with an equal dimension; or vertices, edges, faces, and cells) are now stored separately. The new structure of `/topologies/<mesh_name>/topology` is following:

```
$ h5dump --contents example.h5
HDF5 "example.h5" {
FILE_CONTENTS {
 ...
 group      /topologies/plexA/topology
 dataset    /topologies/plexA/topology/permutation
 group      /topologies/plexA/topology/strata
 group      /topologies/plexA/topology/strata/0
 dataset    /topologies/plexA/topology/strata/0/cone_sizes
 dataset    /topologies/plexA/topology/strata/0/cones
 dataset    /topologies/plexA/topology/strata/0/orientations
 group      /topologies/plexA/topology/strata/1
 dataset    /topologies/plexA/topology/strata/1/cone_sizes
 dataset    /topologies/plexA/topology/strata/1/cones
 dataset    /topologies/plexA/topology/strata/1/orientations
 group      /topologies/plexA/topology/strata/2
 dataset    /topologies/plexA/topology/strata/2/cone_sizes
 dataset    /topologies/plexA/topology/strata/2/cones
 dataset    /topologies/plexA/topology/strata/2/orientations
 group      /topologies/plexA/topology/strata/3
 dataset    /topologies/plexA/topology/strata/3/cone_sizes
 dataset    /topologies/plexA/topology/strata/3/cones
 dataset    /topologies/plexA/topology/strata/3/orientations
 }
}
```

Group `/topologies/<mesh_name>/topology/strata` contains a subgroup for each stratum depth (dimension; 0 for vertices up to 3 for cells). DAG points (mesh entities) have an implicit global numbering, given by the position in `orientations` (or `cone_sizes`) plus the stratum offset. The stratum offset is given by a sum of lengths of all previous strata with respect to the order stored in `/topologies/<mesh_name>/topology/permutation`. This global numbering is compatible with the explicit numbering in dataset `topology/order` of previous versions.

For a DAG point with index `i` at depth `s`, `cone_sizes[i]` gives a size of this point's cone (set of adjacent entities with depth `s-1`). Let `o = sum(cone_sizes[0:i])` (in Python syntax). Points forming the cone are then given by `cones[o:o+cone_sizes[i]]` (in numbering relative to the depth `s-1`). The orientation of the cone with respect to point `i` is then stored in `orientations[i]`.

Loading

To load data from "example.h5" file, we create a `PetscViewer` of type `PETSCVIEWERHDF5` in `FILE_MODE_READ` mode as:

```
PetscViewerHDF5Open(PETSC_COMM_WORLD, "example.h5", FILE_MODE_READ, &viewer);
```

We then create a `DMPLEX` object, give it a *mesh name*, "plexA", and load the mesh as:

```
DMCreate(PETSC_COMM_WORLD, &dm);
DMSetType(dm, DMPLEX);
PetscObjectSetName((PetscObject)dm, "plexA");
PetscViewerPushFormat(viewer, PETSC_VIEWER_HDF5_PETSC);
DMLoad(dm, viewer);
PetscViewerPopFormat(viewer);
```

where `PETSC_VIEWER_HDF5_PETSC` format was again used. The user can have more control by replace the single load call with

```
PetscSF  sf0;

DMCreate(PETSC_COMM_WORLD, &dm);
DMSetType(dm, DMPLEX);
PetscObjectSetName((PetscObject)dm, "plexA");
PetscViewerPushFormat(viewer, PETSC_VIEWER_HDF5_PETSC);
DMPlexTopologyLoad(dm, viewer, &sf0);
DMPlexCoordinatesLoad(dm, viewer, sf0);
PetscViewerPopFormat(viewer);
```

The object returned by `DMPlexTopologyLoad()`, `sf0`, is a `PetscSF` that pushes forward X to the loaded mesh, `dm`; this `PetscSF` is constructed with the global point number tags that we saved along with the mesh points.

As the `DMPLEX` mesh just loaded might not have a desired distribution, it is common to redistribute the mesh for a better distribution using `DMPlexDistribute()`, e.g., as:

```
DM        distributedDM;
PetscInt  overlap = 1;
PetscSF   sfDist, sf;

DMPlexDistribute(dm, overlap, &sfDist, &distributedDM);
if (distributedDM) {
  DMDestroy(&dm);
  dm = distributedDM;
  PetscObjectSetName((PetscObject)dm, "plexA");
}
PetscSFCompose(sf0, sfDist, &sf);
PetscSFDestroy(&sf0);
PetscSFDestroy(&sfDist);
```

Note that the new `DMPLEX` does not automatically inherit the *mesh name*, so we need to name it "plexA" once again. `sfDist` is a `PetscSF` that pushes forward the loaded mesh to the redistributed mesh, so, composed with `sf0`, it makes the `PetscSF` that pushes forward X directly to the redistributed mesh, which we call `sf`.

We then create a new `DM`, `sdm`, with `DMClone()`, give it a *dm name*, "dmA", and load the on-disk data layout into `sdm` as:

```
PetscSF  globalDataSF, localDataSF;

DMClone(dm, &sdm);
PetscObjectSetName((PetscObject)sdm, "dmA");
DMPlexSectionLoad(dm, viewer, sdm, sf, &globalDataSF, &localDataSF);
```

where we could also create a new `DMSHELL` object instead of using `DMClone()`. Each point in the on-disk data layout being tagged with a global point number in X, `DMPlexSectionLoad()` internally constructs a `PetscSF` that pushes forward the on-disk data layout to X. Composing this with `sf`, `DMPlexSectionLoad()` internally constructs another `PetscSF` that pushes forward the on-disk data layout directly to the redistributed mesh. It then reconstructs the data layout `s` on the redistributed mesh and attaches it to `sdm`. The objects returned by this function, `globalDataSF` and `localDataSF`, are `PetscSF`s that can be used to migrate the on-disk vector data into local and global `Vec`s defined on `sdm`.

We now create a local vector associated with `sdm`, e.g., as:

```
Vec  vec;

DMGetLocalVector(sdm, &vec);
```

We then name `vec` "vecA" and load the on-disk vector into `vec` as:

```
PetscObjectSetName((PetscObject)vec, "vecA");
DMPlexLocalVectorLoad(dm, viewer, sdm, localDataSF, localVec);
```

where `localDataSF` knows how to migrate the on-disk vector data into a local `Vec` defined on `sdm`. The on-disk vector can be loaded into a global vector associated with `sdm` in the exact same way with trivial changes.

After loading, we destroy the `PetscViewer` with:

```
PetscViewerDestroy(&viewer);
```

The above infrastructure works seamlessly in distributed-memory parallel settings, in which one can even use different number of processes for saving and for loading; a more comprehensive example is found in DMPlex Tutorial ex12.

3.3.9 Metric-based mesh adaptation

`DMPLEX` supports mesh adaptation using the *Riemannian metric framework*. The idea is to use a Riemannian metric space within the mesher. The metric space dictates how mesh resolution should be distributed across the domain. Using this information, the remesher transforms the mesh such that it is a *unit mesh* when viewed in the metric space. That is, the image of each of its elements under the mapping from Euclidean space into the metric space has edges of unit length.

One of the main advantages of metric-based mesh adaptation is that it allows for fully anisotropic remeshing. That is, it provides a means of controlling the shape and orientation of elements in the adapted mesh, as well as their size. This can be particularly useful for advection-dominated and directionally-dependent problems.

See [Ala10] for further details on metric-based anisotropic mesh adaptation.

The two main ingredients for metric-based mesh adaptation are an input mesh (i.e. the `DMPLEX`) and a Riemannian metric. The implementation in PETSc assumes that the metric is piecewise linear and continuous across elemental boundaries. Such an object can be created using the routine

```
DMPlexMetricCreate(DM dm, PetscInt field, Vec *metric);
```

A metric must be symmetric positive-definite, so that distances may be properly defined. This can be checked using

```
DMPlexMetricEnforceSPD(DM dm, Vec metricIn, PetscBool restrictSizes, PetscBool␣
↪restrictAnisotropy, Vec metricOut, Vec determinant);
```

This routine may also be used to enforce minimum and maximum tolerated metric magnitudes (i.e. cell sizes), as well as maximum anisotropy. These quantities can be specified using

```
DMPlexMetricSetMinimumMagnitude(DM dm, PetscReal h_min);
DMPlexMetricSetMaximumMagnitude(DM dm, PetscReal h_max);
DMPlexMetricSetMaximumAnisotropy(DM dm, PetscReal a_max);
```

or the command line arguments

```
-dm_plex_metric_h_min h_min
-dm_plex_metric_h_max h_max
-dm_plex_metric_a_max a_max
```

One simple way to combine two metrics is by simply averaging them entry-by-entry. Another is to *intersect* them, which amounts to choosing the greatest level of refinement in each direction. These operations are available in PETSc through the routines

```
DMPlexMetricAverage(DM dm, PetscInt numMetrics, PetscReal weights[], Vec metrics[],␣
↪Vec metricAvg);
DMPlexMetricIntersection(DM dm, PetscInt numMetrics, Vec metrics[], Vec metricInt);
```

However, before combining metrics, it is important that they are scaled in the same way. Scaling also allows the user to control the number of vertices in the adapted mesh (in an approximate sense). This is achieved using the L^p normalization framework, with the routine

```
DMPlexMetricNormalize(DM dm, Vec metricIn, PetscBool restrictSizes, PetscBool␣
↪restrictAnisotropy, Vec metricOut, Vec determinant);
```

There are two important parameters for the normalization: the normalization order p and the target metric complexity, which is analogous to the vertex count. They are controlled using

```
DMPlexMetricSetNormalizationOrder(DM dm, PetscReal p);
DMPlexMetricSetTargetComplexity(DM dm, PetscReal target);
```

or the command line arguments

```
-dm_plex_metric_p p
-dm_plex_metric_target_complexity target
```

Two different metric-based mesh adaptation tools are available in PETSc:

- Pragmatic;
- Mmg/ParMmg.

Mmg is a serial package, whereas ParMmg is the MPI version. Note that surface meshing is not currently supported and that ParMmg works only in three dimensions. Mmg can be used for both two and three dimensional problems. Pragmatic, Mmg and ParMmg may be specified by the command line arguments

```
-dm_adaptor (pragmatic|mmg|parmmg)
```

Once a metric has been constructed, it can be used to perform metric-based mesh adaptation using the routine

```
DMAdaptMetric(DM dm, Vec metric, DMLabel bdLabel, DMLabel rgLabel, DM dmAdapt);
```

where `bdLabel` and `rgLabel` are boundary and interior tags to be preserved under adaptation, respectively.

3.4 DMSTAG: Staggered, Structured Grid

For structured (aka "regular") grids with staggered data (degrees of freedom potentially living on elements, faces, edges, and/or vertices), the `DMSTAG` object is available. This can be useful for problems in many domains, including fluid flow, MHD, and seismology.

It is possible, though extremely cumbersome, to implement a staggered-grid code using multiple `DMDA` objects, or a single multi-component `DMDA` object where some degrees of freedom are unused. `DMSTAG` was developed for two main purposes:

1. To help manage some of the burden of choosing and adhering to the complex indexing conventions needed for staggered grids (in parallel)
2. To provide a uniform abstraction for which scalable solvers and preconditioners may be developed (in particular, using `PCFIELDSPLIT` and `PCMG`).

`DMSTAG` is design to behave much like *DMDA*, with a couple of important distinctions, and borrows some terminology from *DMPLEX*.

3.4.1 Terminology

Like a `DMPLEX` object, a `DMSTAG` represents a cell complex, distributed in parallel over the ranks of an `MPI_Comm`. It is, however, a very regular complex, consisting of a structured grid of d-dimensional cells, with $d \in \{1, 2, 3\}$, which are referred to as *elements*, $d-1$ dimensional cells defining boundaries between these elements, and the boundaries of the domain, and in 2 or more dimensions, boundaries of *these* cells, all the way down to 0 dimensional cells referred to as *vertices*. In 2 dimensions, the 1-dimensional element boundaries are referred to as *edges* or *faces*. In 3 dimensions, the 2-dimensional element boundaries are referred to as *faces* and the 1-dimensional boundaries between faces are referred to as *edges* The set of cells of a given dimension is referred to as a *stratum* (which one can think of as a level in DAG representation of the mesh); a `DMSTAG` object of dimension d represents a complete cell complex with $d+1$ *strata* (levels).

In the description of any:`ch_unstructured` the cells at each level are referred to as *points*. Thus we adopt that terminology uniformly in PETSc and so furthermore in this document, point will refer to a cell.

Each stratum has a constant number of unknowns (which may be zero) associated with each point (cell) on that level. The distinct unknowns associated with each point are referred to as *components*. For a `DMPLEX` there may be a different number of unknowns for each point on the same level.

The structured grid, is like with `DMDA`, decomposed via a Cartesian product of decompositions in each dimension, giving a rectangular local subdomain on each rank. This is extended by an element-wise stencil width of *ghost* elements to create an atlas of overlapping patches.

3.4.2 Working with vectors and operators (matrices)

DMSTAG allows the user to reason almost entirely about a global indexing of elements. Element indices are simply 1-3 PetscInt values, starting at 0, in the back, bottom, left corner of the domain. For instance, element $(1, 2, 3)$, in 3D, is the element second from the left, third from the bottom, and fourth from the back (regardless of how many MPI ranks are used).

To refer to points (elements, faces, edges, and vertices), a value of DMStagStencilLocation is used, relative to the element index. The element itself is referred to with DMSTAG_ELEMENT, the top right vertex (in 2D) or the top right edge (in 3D) with DMSTAG_UP_RIGHT, the back bottom left corner in 3D with DMSTAG_BACK_DOWN_LEFT, and so on.

Fig. 3.3 gives a few examples in 2D.

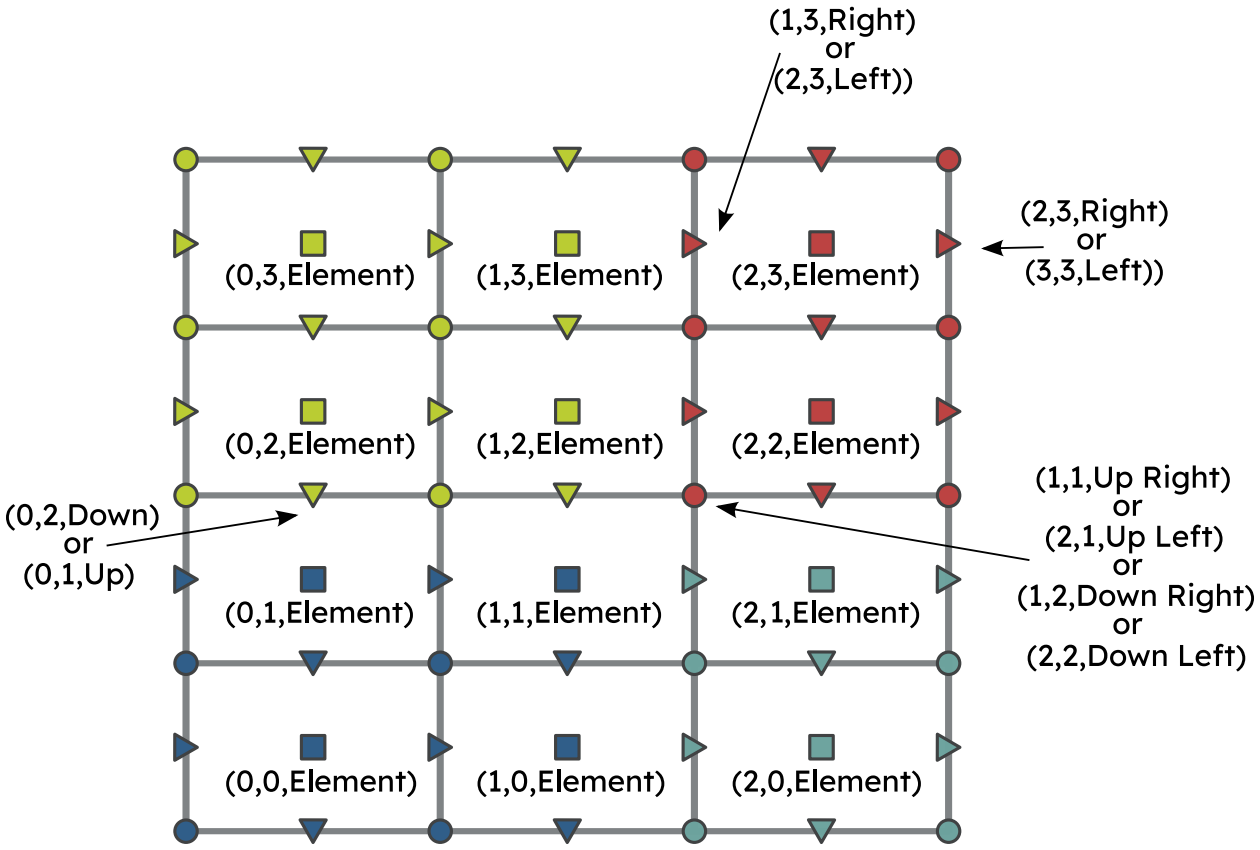


Fig. 3.3: Locations in DMSTAG are indexed according to global element indices (here, two in 2D) and a location name. Elements have unique names but other locations can be referred to in more than one way. Element colors correspond to a parallel decomposition, but locations on the grid have names which are invariant to this. Note that the face on the top right can be referred to as being to the left of a "dummy" element $(3, 3)$ outside the physical domain.

Crucially, this global indexing scheme does not include any "ghost" or "padding" unknowns outside the physical domain. This is useful for higher-level operations such as computing norms or developing physics-based solvers. However (unlike DMDA), this implies that the global Vec do not have a natural block structure, as different strata have different numbers of points (e.g. in 1D there is an "extra" vertex on the right). This regular block structure is, however, very useful for the *local* representation of the data, so in that case *dummy* DOF are included, drawn as grey in Fig. 3.4.

For working with Vec data, this approach is used to allow direct access to a multi-dimensional, regular-

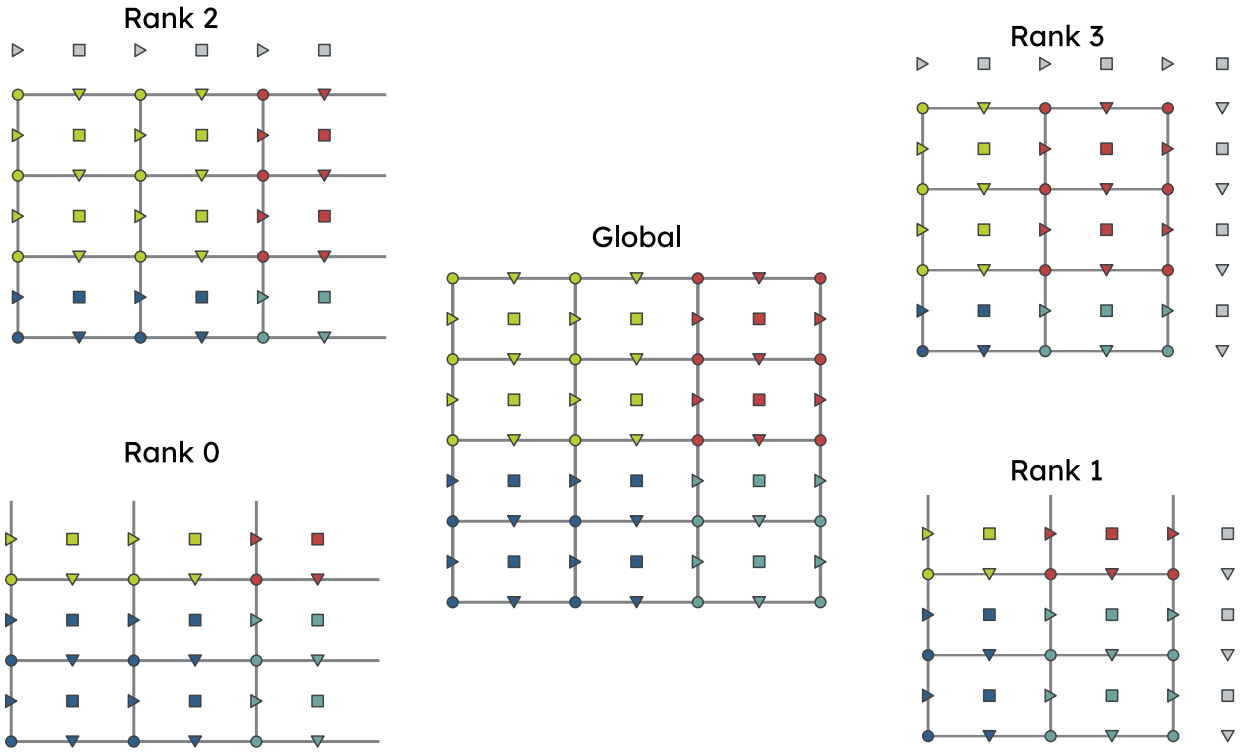


Fig. 3.4: Local and global representations for a 2D DMSTAG object, 3 by 4 elements, with one degree of freedom on each of the three strata: element (squares), faces (triangles), and vertices (circles). The cell complex is parallelized across 4 MPI ranks. In the global representation, the colors correspond to which rank holds the native representation of the unknown. The 4 local representations are shown, with an (elementwise) stencil "box" stencil width of 1. Unknowns are colored by their native rank. Dummy unknowns, which correspond to no global degree of freedom, are colored grey. Note that the local representations have a natural block size of 4, and the global representation has no natural block size.

blocked array. To avoid the user having to know about the *internal numbering conventions used*, helper functions are used to produce the proper final integer index for a given location and component, referred to as a "slot". Similarly to `DMDAVecGetArrayDOF()`, this uses a $d+1$ dimensional array in d dimensions. The following snippet give an example of this usage.

```
  /* Set the second component of all vertex dof to 2.0 */
  PetscCall(DMStagGetCorners(dm, &s_x, &s_y, &s_z, &n_x, &n_y, &n_z, &n_e_x, &n_e_y, &
↪n_e_z));
  PetscCall(DMStagGetLocationSlot(dm, location_vertex, 1, &slot_vertex_2));
  PetscCall(DMStagVecGetArray(dm, x, &x_array));
  for (PetscInt k = s_z; k < s_z + n_z + n_e_z; ++k) {
    for (PetscInt j = s_y; j < s_y + n_y + n_e_y; ++j) {
      for (PetscInt i = s_x; i < s_x + n_x + n_e_x; ++i) x_array[k][j][i][slot_vertex_
↪2] = 2.0;
    }
  }
  PetscCall(DMStagVecRestoreArray(dm, x, &x_array));
```

`DMSTAG` provides a stencil-based method for getting and setting entries of `Mat` and `Vec` objects. The follow excerpt from DMSTAG Tutorial ex1 demonstrates the idea. For more, see the manual page for `DMStagMatSetValuesStencil()`.

```
    /* Velocity is either a BC or an interior point */
    if (isFirstRank && e == start) {
      DMStagStencil row;
      PetscScalar   val;

      row.i   = e;
      row.loc = LEFT;
      row.c   = 0;
      val     = 1.0;
      PetscCall(DMStagMatSetValuesStencil(dmSol, A, 1, &row, 1, &row, &val, INSERT_
↪VALUES));
```

The array-based approach for `Vec` is likely to be more efficient than the stencil-based method just introduced above.

3.4.3 Coordinates

`DMSTAG`, unlike `DMDA`, supports two approaches to defining coordinates. This is captured by which type of `DM` is used to represent the coordinates. No default is imposed, so the user must directly or indirectly call `DMStagSetCoordinateDMType()`.

If a second `DMSTAG` object is used to represent coordinates in "explicit" form, behavior is much like with `DMDA` - the coordinate `DM` has d DOF on each stratum corresponding to coordinates associated with each point.

If `DMPRODUCT` is used instead, coordinates are represented by a `DMPRODUCT` object referring to a Cartesian product of 1D `DMSTAG` objects, each of which features explicit coordinates as just mentioned.

Navigating these nested `DM` in `DMPRODUCT` can be tedious, but note the existence of helper functions like `DMStagSetUniformCoordinatesProduct()` and `DMStagGetProductCoordinateArrays()`.

3.4.4 Numberings and internal data layout

While DMSTAG aims to hide the details of its internal data layout, for debugging, optimization, and customization purposes, it can be important to know how DMSTAG internally numbers unknowns.

Internally, each point is canonically associated with an element (top-level point (cell)). For purposes of local, regular-blocked storage, an element is grouped with lower-dimensional points left of, below ("down"), and behind ("back") it. This means that "canonical" values of DMStagStencilLocation are DMSTAG_ELEMENT, plus all entries consisting only of "LEFT", "DOWN", and "BACK". In general, these are the most efficient values to use, unless convenience dictates otherwise, as they are the ones used internally.

When creating the decomposition of the domain to local ranks, and extending these local domains to handle overlapping halo regions and boundary ghost unknowns, this same per-element association is used. This has the advantage of maintaining a regular blocking, but may not be optimal in some situations in terms of data movement.

Numberings are, like DMDA, based on a local "x-fastest, z-slowest" or "PETSc" ordering of elements (see *Application Orderings*), with ordering of locations canonically associated with each element decided by considering unknowns on each point to be located at the center of their point, and using a nested ordering of the same style. Thus, in 3-D, the ordering of the 8 canonical DMStagStencilLocation values associated with an element is

```
DMSTAG_BACK_DOWN_LEFT
DMSTAG_BACK_DOWN
DMSTAG_BACK_LEFT
DMSTAG_BACK
DMSTAG_DOWN_LEFT
DMSTAG_DOWN
DMSTAG_LEFT
DMSTAG_ELEMENT
```

Multiple DOF associated with a given point are stored sequentially (as with DMDA).

For local Vecs, this gives a regular-blocked numbering, with the same number of unknowns associated with each element, including some "dummy" unknowns which to not correspond to any (local or global) unknown in the global representation. See Fig. 3.6 for an example.

In the global representation, only physical unknowns are numbered (using the same "Z" ordering for unknowns which are present), giving irregular numbers of unknowns, depending on whether a domain boundary is present. See Fig. 3.5 for an example.

It should be noted that this is an *interlaced* (AoS) representation. If a segregated (SoA) representation is required, one should use DMCOMPOSITE collecting several DMSTAG objects, perhaps using DMStagCreateCompatibleDMStag() to quickly create additional DMSTAG objects from an initial one.

3.5 Networks

The DMNETWORK class provides abstractions for representing general unstructured networks such as communication networks, power grid, computer networks, transportation networks, electrical circuits, graphs, and others.

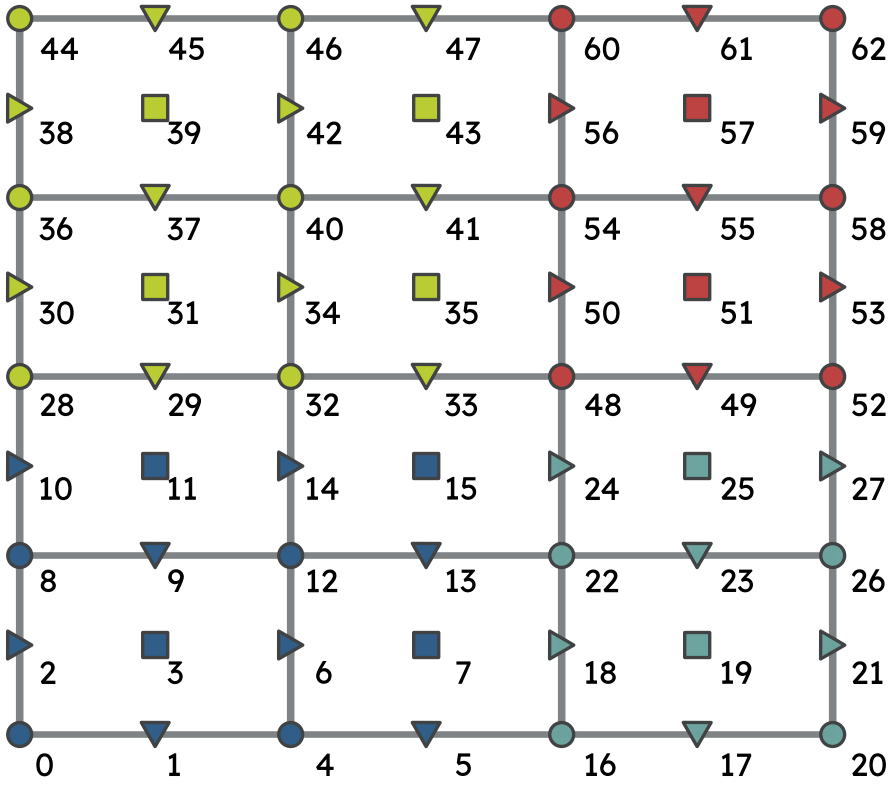


Fig. 3.5: Global numbering scheme for a 2D DMSTAG object with one DOF per stratum. Note that the numbering depends on the parallel decomposition (over 4 ranks, here).

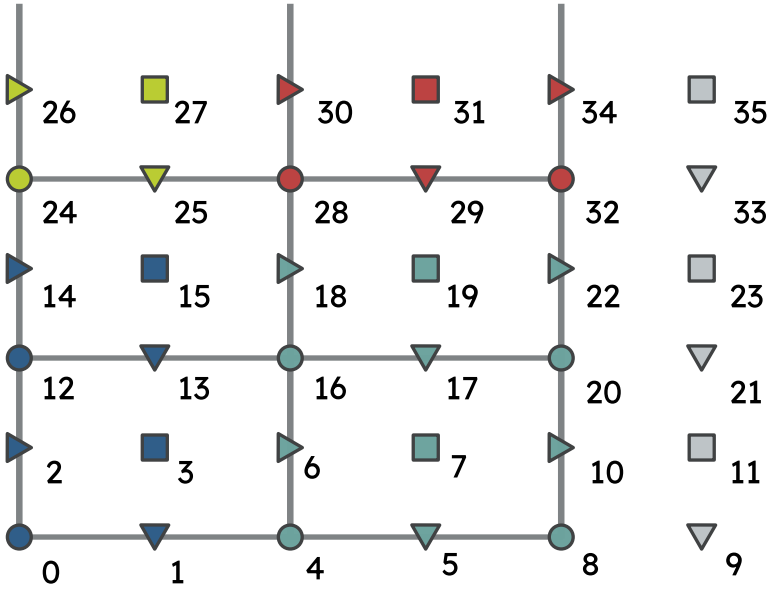


Fig. 3.6: Local numbering scheme on rank 1 (Cf. Fig. 3.4) for a 2D DMSTAG object with one DOF per stratum. Note that dummy locations (grey) are used to give a regular block size (here, 4).

3.5.1 Application flow

The general flow of an application code using DMNETWORK is as follows:

1. Create a network object.

```
DMNetworkCreate(MPI_Comm comm, DM *dm);
```

2. Create components and register them with the network. A "component" is specific application data at a vertex/edge of the network required for its residual evaluation. For example, components could be resistor/inductor data for circuit applications, edge weights for graph problems, or generator/transmission line data for power grids. Components are registered by calling

```
DMNetworkRegisterComponent(DM dm, const char *name, size_t size, PetscInt␣
↪*compkey);
```

   Here, name is the component name, size is the size of component data, and compkey is an integer key that can be used for setting/getting the component at a vertex or an edge.

3. A DMNETWORK can consist of one or more physical subnetworks. Each subnetwork has its own mathematical model. When multiple subnetworks are used one can (optionally) provide coupling information between subnetworks. That is vertices that are *shared* between multiple subnetworks; edges can only belong to a single subnetwork. The number of subnetwork is set by calling

```
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

```
DMNetworkAddSubnetwork(DM dm, const char* name, PetscInt ne, PetscInt edgelist[],␣
↪PetscInt *netnum);
```

   Here name is the subnetwork name, ne is the number of 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 comprised 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] = {0,1,1,2,2,3}

   edgelist[1] = {1,2,2,0}

   The coupling is done by calling

```
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

```
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

```
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 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.

```
DMSetUp(DM dm);
```

8. Distribute the network (also moves components attached with vertices/edges) to multiple processors.

```
DMNetworkDistribute(DM dm, const char partitioner[], PetscInt overlap, DM␣
↪*distDM);
```

9. Associate the DM with a PETSc solver:

```
KSPSetDM(KSP ksp, DM dm) or SNESSetDM(SNES snes, DM dm) or TSSetDM(TS ts, DM dm).
```

3.5.2 Utility functions

DMNETWORK provides several utility functions for operations on the network. The most commonly used functions are: obtaining iterators for vertices/edges,

```
DMNetworkGetEdgeRange(DM dm, PetscInt *eStart, PetscInt *eEnd);
```

```
DMNetworkGetVertexRange(DM dm, PetscInt *vStart, PetscInt *vEnd);
```

```
DMNetworkGetSubnetwork(DM dm, PetscInt netnum, PetscInt *nv, PetscInt *ne, const␣
↪PetscInt **vtx, const PetscInt **edge);
```

checking the status of a vertex,

```
DMNetworkIsGhostVertex(DM dm, PetscInt p, PetscBool *isghost);
```

```
DMNetworkIsSharedVertex(DM dm, PetscInt p, PetscBool *isshared);
```

and retrieving local/global indices of vertex/edge component variables for inserting elements in vectors/matrices,

```
DMNetworkGetLocalVecOffset(DM dm, PetscInt p, PetscInt compnum, PetscInt *offset);
```

```
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.

```
DMNetworkGetConnectedVertices(DM dm, PetscInt edge, const PetscInt *vertices[]);
```

```
DMNetworkGetSupportingEdges(DM dm, PetscInt vertex, PetscInt *nedges, const PetscInt␣
↪*edges[]).
```

3.5.3 Retrieving components and number of variables

The components and the corresponding number of variables set at a vertex/edge can be accessed by

```
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:

```
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);
  }
}
```

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.

3.6 DM Commonalities

We have introduced a variety of seemingly very different DM. Here, we will try to explore the commonalities between them to emphasize that despite superficial differences they all tackle the three same basic problems:

1. How to map between the indices of (geometric) entities, in some physical space such as R^3 or perhaps an abstract space, and the storage location (offsets) of numerical values associated with said entities in PETSc vectors or arrays. In PETSc, these geometric entities are referred to as points.
2. How to iterate over the entities of interest (for example, points) to produce updates to the numerical values associated with the entities in the PETSc vectors or arrays.
3. How to store/access the "connectivity information" that is needed at each entity (point) and provides the correct data dependencies needed to compute the numerical updates.

For several DM we will devote a short paragraph for each of the three problems.

3.6.1 DMDA simple structured grids

For structured grids, *DMDA - Creating vectors for structured grids*, the indexing is trivial, the points are represented as tuples (i, j, k), where $l_i \leq i \leq u_i$, $l_j \leq j \leq u_j$, and $l_k \leq k \leq u_k$. `DMDAVecGetArray()` returns a multidimensional array that trivially provides the mapping from said points to the numerical values. Note that when the programming language gives access to the values in a multi-dimensional array, internally it computes the `offset` from the beginning of the array using a formula based on the value of i, j, and k and the array dimensions.

To iterate over the local points, one uses `DMDAGetCorners()` or `DMDAGetGhostCorners()` and iterates over the tuples within the bounds. Specific points, for example boundary points, can be skipped or processed differently based on the index values.

For finite difference methods on structured grids using a stencil formula, the "connectivity information" is defined implicitly by the stencil needed by the given discretization and is the same for all grid points (except maybe boundaries or other special points). For example, for the standard seven point stencil computed at the (i, j, k) entity one needs the numerical values at the $(i \pm 1, j, k)$, $(i, j \pm 1, k)$, and $(i, j, k \pm 1)$ entities.

3.6.2 DMSTAG simple stagger grids

A staggered grid, *DMSTAG: Staggered, Structured Grid*, extends the idea of a simple structured grid by allowing not only entities associated with grid vertices (or equivalently cells) as with DMDA but also with grid edges, grid faces, and grid cells (also called elements). As with DMDA each type of entity must have the same number of associated numerical values. As with simple structured grids, each cell can be represented as a (i, j, k) tuple. But, in addition we need to represent what vertex, edge, or face of the cell we are referring to. This is done using `DMStagStencilLocation`. `DMStagVecGetArray()` returns a multidimensional array indexed by (i, j, k) plus a `slot` that tells us which entity on the cell is being accessed. Since a staggered grid can have any problem-dependent number of numerical values associated with a given entity type, the function `DMStagGetLocationSlot()` provides the final index needed for the array access. After this the programming language than computes the `offset` from the beginning of the array from the provided indices using a simple formula.

To iterate over the local points, one uses `DMStagGetCorners()` or `DMStagGetGhostCorners()` and iterates over the tuples within the bounds with an inner iteration of the point entities desired for the application. For example, for a discretization with cell-centered pressures and edge-based velocity the application would process each of these entities.

For finite difference methods on staggered structured grids using a stencil formula the "connectivity information" is again defined implicitly by the stencil needed by the given discretization and is the same for all grid

points (except maybe boundaries or other special points). In addition, any required cross coupling between different entities needs to be encoded for the given problem. For example, how do the velocities affect the pressure equation. This information is generally embedded directly in lines of code that implement the finite difference formula and is not represented in the data structures.

3.6.3 DMPLEX unstructured meshes

For general unstructured grids, *DMPlex: Unstructured Grids*, there is no formula for computing the `offset` into the numerical values for an entity on the grid from its `point` value, since each `point` can have any number of values which is determined at runtime based on the grid, PDE, and discretization. Hence all the offsets must be managed by `DMPLEX`. This is the job of `PetscSection`, *PetscSection: Connecting Grids to Data*. The process of building a `PetscSection` computes and stores the offsets and then using the `PetscSection` gives access to the needed offsets.

For unstructured grids, one does not in general iterate over all the entities on all the points. Rather it iterates over a subset of the points representing a particular entity. For example, when using the finite element method, the application iterates all the points representing elements (cells) using `DMPlexGetHeightStratum()` to access the chart (beginning and end indices) of the cell entities. Then one uses an associated `PetscSection` to determine the offsets into vectors or arrays for said points. If needed one can then have an inner iteration over the fields associated with the cell.

For `DMPLEX`, the connectivity information is defined by a graph (and stored explicitly in a data structure used to store graphs), and the connectivity of a point (entity) is obtained by `DMPlexGetTransitiveClosure()`.

3.6.4 DMNETWORK computations on graphs of nodes and connecting edges

For networks, *Networks*, the entities are nodes and edges that connect two nodes, each of which can have any number of submodels. Again, in general, there is no formula that can produce the appropriate `offset` into the numerical values for a given `point` (node or edge) directly. The routines `DMNetworkGetLocalVecOffset()` and `DMNetworkGetGlobalVecOffset()` are used to obtain the needed offsets from a given point (node or edge) and submodel at that point. Internally a `PetscSection` is used to manage the storage of the `offset` information but the user-level API does not refer to `PetscSection` directly, rather one thinks about a collection of submodels at each node and edge of the graph.

To iterate over graph vertices (nodes) one uses `DMNetworkGetVertexRange()` to provide its chart (the starting and end indices) and `DMNetworkGetEdgeRange()` to provide the chart of the edges. One can then iterate over the models on each point To iterate over sub-networks one can call `DMNetworkGetSubnetwork()` for each network which returns lists of the vertex and edge points in said network.

For `DMNETWORK`, the connectivity information is defined by a graph, which is is query-able at each entity by `DMNetworkGetSupportingEdges()` and `DMNetworkGetConnectedVertices()`.

3.7 Is it a programming language issue?

Regarding problem 1. Does the need for these various approaches for mapping between the entities and the related array offsets and the large amount of code (in particular `PetscSection` come from the limitation of programming languages when working with complex multidimensional jagged arrays. Both in constructing such arrays at runtime, that is supplying all the jagged information which depends on the exact problem, and then providing simple syntax to produce the correct offset into the memory for accessing the numerical values when the simple array access methods do not work.

3.8 PetscDT: Discretization Technology in PETSc

This chapter discusses the low-level infrastructure which supports higher-level discretizations in PETSc, which includes things such as quadrature and probability distributions.

3.8.1 Quadrature

3.8.2 Probability Distributions

A probability distribution function (PDF) returns the probability density at a given location $P(x)$, so that the probability for an event at location in $[x, x+dx]$ is $P(x)dx$. This means that we must have the normalization condition,

$$\int_\Omega P(x)dx = 1.$$

where :math:Omega is the domain for x. This requires that the PDF must have units which are the inverse of the volume form dx, meaning that it is homogeneous of order d under scaling

$$pdf(x) = \lambda^d P(\lambda x).$$

We can check this using the normalization condition,

$$\begin{aligned}\int_\Omega P(x)dx &= \int_\Omega P(\lambda s)\lambda^d ds \\ &= \int_\Omega P(s)\lambda^{-d}\lambda^d ds \\ &= \int_\Omega P(s)ds \\ &= 1\end{aligned}$$

The cumulative distribution function (CDF) is the incomplete integral of the PDF,

$$C(x) = \int_{x_}^{x} P(s)ds$$

where x_- is the lower limit of our domain. We can work out the effect of scaling on the CDF using this definition,

$$\begin{aligned} C(\lambda x) &= \int_{x_-}^{\lambda x} P(s)ds \\ &= \int_{x_-}^{x} \lambda^d P(\lambda t)dt \\ &= \int_{x_-}^{x} P(t)dt \\ &= C(x) \end{aligned}$$

so the CDF itself is scale invariant and unitless.

We do not add a scale argument to the PDF in PETSc, since all variables are assuming to be dimensionless. This means that inputs to the PDF and CDF should be scaled by the appropriate factor for the units of x, and the output can be rescaled if it is used outside the library.

3.9 PetscFE: Finite Element Infrastructure in PETSc

This chapter introduces the `PetscFE` class, and related subclasses `PetscSpace` and `PetscDualSpace`, which are used to represent finite element discretizations. It details there interaction with the `DMPLEX` class to assemble functions and operators over computational meshes, and produce optimal solvers by constructing multilevel iterations, for example using `PCPATCH`. The idea behind these classes is not to encompass all of computational finite elements, but rather to establish an interface and infrastructure that will allow PETSc to leverage the excellent work done in packages such as Firedrake, FEniCS, LibMesh, and Deal.II.

3.9.1 Using Pointwise Functions to Specify Finite Element Problems

See the paper about Unified Residual Evaluation, which explains the use of pointwise evaluation functions to describe weak forms.

3.9.2 Describing a particular finite element problem to PETSc

A finite element problem is presented to PETSc in a series of steps. This is both to facilitate automation, and to allow multiple entry points for user code and external packages since so much finite element software already exists. First, we tell the `DM`, usually a `DMPLEX` or `DMFOREST`, that we have a set of finite element fields which we intended to solve for in our problem, using

```
DMAddField(dm, NULL, presDisc);
DMAddField(dm, channelLabel, velDisc);
```

The second argument is a `DMLabel` object indicating the support of the field on the mesh, with `NULL` indicating the entire domain. Once we have a set of fields, we calls

A `PetscDS` (Discrete System) encodes a set of equations posed in a discrete space, which represents a set of nonlinear continuum equations. The equations can have multiple fields, each field having a different discretization. In addition, different pieces of the domain can have different field combinations and equations.

The DS provides the user a description of the approximation space on any given cell. It also gives pointwise functions representing the equations.

Each field is associated with a DMLabel, marking the cells on which it is supported. Note that a field can be supported on the closure of a cell not in the label due to overlap of the boundary of neighboring cells. The DM then creates a PetscDS for each set of cells with identical approximation spaces. When assembling, the user asks for the space associated with a given cell. DMPLEX uses the labels associated with each PetscDS in the default integration loop.

```
DMCreateDS(dm);
```

This divides the computational domain into subdomains, called *regions* in PETSc, each with a unique set of fields supported on it. These subdomain are identified by labels, and each one has a PetscDS object describing the discrete system on that subdomain. There are query functions to get the set of PetscDS objects for the DM, but it is usually easiest to get the proper PetscDS for a given cell using

```
DMGetCellDS(dm, cell, &ds, NULL);
```

Each PetscDS object has a set of fields, each with a PetscFE or PetscFV discretization. This allows it to calculate the size of the local discrete approximation, as well as allocate scratch space for all the associated computations. The final thing needed is to specify the actual equations to be enforced on each region. The PetscDS contains a PetscWeakForm object that holds callback function pointers that define the equations. A simplified, top-level interface through PetscDS allows users to quickly define problems for a single region. For example, in SNES Tutorial ex13, we define the Poisson problem using

```
DMLabel  label;
PetscInt f = 0, id = 1;

PetscDSSetResidual(ds, f, f0_trig_inhomogeneous_u, f1_u);
PetscDSSetJacobian(ds, f, f, NULL, NULL, NULL, g3_uu);
PetscDSSetExactSolution(ds, f, trig_inhomogeneous_u, user);
DMGetLabel(dm, "marker", &label);
DMAddBoundary(dm, DM_BC_ESSENTIAL, "wall", label, 1, &id, f, 0, NULL, (PetscVoidFn *)␣
↪ex, NULL, user, NULL);
```

where the pointwise functions are

```
static PetscErrorCode trig_inhomogeneous_u(PetscInt dim, PetscReal time, const␣
↪PetscReal x[], PetscInt Nc, PetscScalar *u, PetscCtx ctx)
{
  PetscInt d;
  *u = 0.0;
  for (d = 0; d < dim; ++d) *u += PetscSinReal(2.0*PETSC_PI*x[d]);
  return 0;
}

static void f0_trig_inhomogeneous_u(PetscInt dim, PetscInt Nf, PetscInt NfAux,
                      const PetscInt uOff[], const PetscInt uOff_x[], const␣
↪PetscScalar u[], const PetscScalar u_t[], const PetscScalar u_x[],
                      const PetscInt aOff[], const PetscInt aOff_x[], const␣
↪PetscScalar a[], const PetscScalar a_t[], const PetscScalar a_x[],
                      PetscReal t, const PetscReal x[], PetscInt numConstants, const␣
↪PetscScalar constants[], PetscScalar f0[])
{
  PetscInt d;
  for (d = 0; d < dim; ++d) f0[0] += -4.0*PetscSqr(PETSC_PI)*PetscSinReal(2.0*PETSC_
↪PI*x[d]);
}

static void f1_u(PetscInt dim, PetscInt Nf, PetscInt NfAux,
```

(continues on next page)

(continued from previous page)

```
                  const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar␣
↪u[], const PetscScalar u_t[], const PetscScalar u_x[],
                  const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar␣
↪a[], const PetscScalar a_t[], const PetscScalar a_x[],
                  PetscReal t, const PetscReal x[], PetscInt numConstants, const␣
↪PetscScalar constants[], PetscScalar f1[])
{
  PetscInt d;
  for (d = 0; d < dim; ++d) f1[d] = u_x[d];
}

static void g3_uu(PetscInt dim, PetscInt Nf, PetscInt NfAux,
                  const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar␣
↪u[], const PetscScalar u_t[], const PetscScalar u_x[],
                  const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar␣
↪a[], const PetscScalar a_t[], const PetscScalar a_x[],
                  PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt␣
↪numConstants, const PetscScalar constants[], PetscScalar g3[])
{
  PetscInt d;
  for (d = 0; d < dim; ++d) g3[d*dim+d] = 1.0;
}
```

Notice that we set boundary conditions using `DMAddBoundary`, which will be described later in this chapter. Also we set an exact solution for the field. This can be used to automatically calculate mesh convergence using the `PetscConvEst` object described later in this chapter.

For more complex cases with multiple regions, we need to use the `PetscWeakForm` interface directly. The weak form object allows you to set any number of functions for a given field, and also allows functions to be associated with particular subsets of the mesh using labels and label values. We can reproduce the above problem using the *SetIndex* variants which only set a single function at the specified index, rather than a list of functions. We use a `NULL` label and value, meaning that the entire domain is used.

```
PetscInt f = 0, val = 0;

PetscDSGetWeakForm(ds, &wf);
PetscWeakFormSetIndexResidual(ds, NULL, val, f, 0, 0, f0_trig_inhomogeneous_u, 0, f1_
↪u);
PetscWeakFormSetIndexJacobian(ds, NULL, val, f, f, 0, 0, NULL, 0, NULL, 0, NULL, 0,␣
↪g3_uu);
```

In SNES Tutorial ex23, we define the Poisson problem over the entire domain, but in the top half we also define a pressure. The entire problem can be specified as follows

```
DMGetRegionNumDS(dm, 0, &label, NULL, &ds, NULL);
PetscDSGetWeakForm(ds, &wf);
PetscWeakFormSetIndexResidual(wf, label, 1, 0, 0, 0, f0_quad_u, 0, f1_u);
PetscWeakFormSetIndexJacobian(wf, label, 1, 0, 0, 0, 0, NULL, 0, NULL, 0, NULL, 0, g3_
↪uu);
PetscDSSetExactSolution(ds, 0, quad_u, user);
DMGetRegionNumDS(dm, 1, &label, NULL, &ds, NULL);
PetscDSGetWeakForm(ds, &wf);
PetscWeakFormSetIndexResidual(wf, label, 1, 0, 0, 0, f0_quad_u, 0, f1_u);
PetscWeakFormSetIndexJacobian(wf, label, 1, 0, 0, 0, 0, NULL, 0, NULL, 0, NULL, 0, g3_
↪uu);
PetscWeakFormSetIndexResidual(wf, label, 1, 1, 0, 0, f0_quad_p, 0, NULL);
```

(continues on next page)

(continued from previous page)

```
PetscWeakFormSetIndexJacobian(wf, label, 1, 1, 1, 0, 0, g0_pp, 0, NULL, 0, NULL, 0,␣
↪NULL);
PetscDSSetExactSolution(ds, 0, quad_u, user);
PetscDSSetExactSolution(ds, 1, quad_p, user);
DMGetLabel(dm, "marker", &label);
DMAddBoundary(dm, DM_BC_ESSENTIAL, "wall", label, 1, &id, 0, 0, NULL, (PetscVoidFn *)␣
↪quad_u, NULL, user, NULL);
```

In the PyLith software we use this capability to combine bulk elasticity with a fault constitutive model integrated over the embedded manifolds corresponding to earthquake faults.

3.9.3 Assembling finite element residuals and Jacobians

Once the pointwise functions are set in each `PetscDS`, mesh traversals can be automatically determined from the `DMLabel` and value specifications in the keys. This default traversal strategy can be activated by attaching the `DM` and default callbacks to a solver

```
SNESSetDM(snes, dm);
DMPlexSetSNESLocalFEM(dm, &user, &user, &user);

TSSetDM(ts, dm);
DMTSSetBoundaryLocal(dm, DMPlexTSComputeBoundary, &user);
DMTSSetIFunctionLocal(dm, DMPlexTSComputeIFunctionFEM, &user);
DMTSSetIJacobianLocal(dm, DMPlexTSComputeIJacobianFEM, &user);
```

CHAPTER

FOUR

ADDITIONAL INFORMATION

4.1 PETSc for Fortran Users

Make sure the suffix of your Fortran files is .F90, not .f or .f90.

4.1.1 Fortran and MPI

By default PETSc uses the MPI Fortran module `mpi`. To use `mpi_f08` run `./configure` with `--with-mpi-ftn-module=mpi_f08`.

We do not recommend it but it is possible to write Fortran code that works with both `use mpi` or `use mpi_f08`. You must declare MPI objects using `MPIU_XXX` (for example `MPIU_Comm`, which the PETSc include files map to either `integer4` or `type(MPI_XXX)`. In addition, you must handle `MPIU_Status` declarations and access to entries using the Fortran preprocessor. For example,

```
#if defined(PETSC_USE_MPI_F08)
  MPIU_Status status
#else
  MPIU_Status status(MPI_STATUS_SIZE)
#endif
```

and then

```
#if defined(PETSC_USE_MPI_F08)
      tag = status%MPI_TAG
#else
      tag = status(MPI_TAG)
#endif
```

4.1.2 Numerical Constants

Since Fortran compilers do not automatically change the length of numerical constant arguments (integer and real) in subroutine calls to the expected length PETSc provides parameters that indicate the constants' kind.

```
   interface
     subroutine SampleSubroutine(real, complex, integer, MPIinteger)
       PetscReal real         ! real(PETSC_REAL_KIND)
       PetscComplex complex   ! real(PETSC_REAL_KIND) or complex(PETSC_REAL_KIND)
       PetscInt integer       ! integer(PETSC_INT_KIND)
```

(continues on next page)

(continued from previous page)

```
      PetscMPIInt MPIinteger ! integer(PETSC_MPIINT_KIND)
    end subroutine
  end interface
  ...
  ! Fortran () automatically sets the complex KIND to correspond to the KIND of␣
↪constant arguments
  call SampleSubroutine(real = 1.0_PETSC_REAL_KIND, complex = (0.1_PETSC_REAL_KIND,␣
↪0.0_PETSC_REAL_KIND), integer = 1_PETSC_INT_KIND, MPIinteger = 1_PETSC_MPIINT_KIND)
  ! For variable arguments one must set the complex kind explicitly or it defaults␣
↪to single precision
  PetscReal a = 0.1, b = 0.0
  call SampleSubroutine(real = 1.0_PETSC_REAL_KIND, complex = cmplx(a, b, PETSC_REAL_
↪KIND), integer = 1_PETSC_INT_KIND, MPIinteger = 1_PETSC_MPIINT_KIND)
```

4.1.3 Basic Fortran API Differences

Modules and Include Files

You must use both PETSc include files and modules. At the beginning of every function and module definition you need something like

```
#include "petsc/finclude/petscXXX.h"
  use petscXXX
```

The Fortran include files for PETSc are located in the directory `$PETSC_DIR/$PETSC_ARCH/include/petsc/finclude` and the module files are located in `$PETSC_DIR/$PETSC_ARCH/include`

The include files are nested, that is, for example, `petsc/finclude/petscmat.h` automatically includes `petsc/finclude/petscvec.h` and so on. The modules are also nested. One can use

```
use petsc
```

to include all of them.

Declaring PETSc Object Variables

You can declare PETSc object variables using either of the following:

```
XXX variablename
```

```
type(tXXX) variablename
```

For example,

```
#include "petsc/finclude/petscvec.h"
  use petscvec

  Vec b
  type(tVec) x
```

PETSc types like `PetscInt` and `PetscReal` are simply aliases for basic Fortran types and cannot be written as `type(tPetscInt)`

PETSc objects are always automatically initialized when declared so you do not need to (and should not) do

```
type(tXXX) x = PETSC_NULL_XXX
XXX x = PETSC_NULL_XXX
```

To make a variable no longer point to its previously assigned PETSc object use, for example,

```
   Vec x, y
   PetscInt one = 1
   PetscCallA(VecCreateMPI(PETSC_COMM_WORLD, one, PETSC_DETERMINE, x, ierr))
   y = x
   PetscCallA(VecDestroy(x, ierr))
   PetscObjectNullify(y)
```

Otherwise `y` will be a dangling pointer whose access will cause a crash.

Calling Sequences

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*.

The key differences in handling arguments when calling PETSc functions from Fortran are

- One cannot pass a scalar variable to a function expecting an array, *Passing Arrays To PETSc Functions*.
- One must use type specific `PETSC_NULL` arguments, such as `PETSC_NULL_INTEGER`, *Passing null pointers to PETSc functions*.
- One must pass pointers to arrays for arguments that output an array, for example `PetscScalar, pointer \:\: a(\:)`, *Output Arrays from PETSc functions*.
- `PETSC_DECIDE` and friends need to match the argument type, for example `PETSC_DECIDE_INTEGER`.

When passing floating point numbers into PETSc Fortran subroutines, always make sure you have them marked as double precision (e.g., pass in `10.d0` instead of `10.0` or declare them as PETSc variables, e.g. `PetscScalar one = 1.0`). Otherwise, the compiler interprets the input as a single precision number, which can cause crashes or other mysterious problems. We **highly** recommend using the `implicit none` option at the beginning of each Fortran subroutine and declaring all variables.

Error Checking

In the Fortran version, each PETSc routine has as its final argument an integer error variable. The error code is 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 `PetscErrorCode` error variable:

```
call KSPSolve(ksp, b, x, ierr) ! Fortran
ierr = KSPSolve(ksp, b, x);    // C
```

For proper error handling one should not use the above syntax instead one should use

```
PetscCall(KSPSolve(ksp, b, x, ierr))   ! Fortran subroutines
PetscCallA(KSPSolve(ksp, b, x, ierr))  ! Fortran main program
PetscCall(KSPSolve(ksp, b, x))         // C
```

Passing Arrays To PETSc Functions

Many PETSc functions take arrays as arguments; in Fortran they must be passed as arrays even if the "array" is of length one (unlike Fortran 77 where one can pass scalars to functions expecting arrays). When passing a single value one can use the Fortran [] notation to pass the scalar as an array, for example

```
PetscCall(VecSetValues(v, one, [i], [val], ierr))
```

This trick can only be used for arrays used to pass data into a PETSc routine, it cannot be used for arrays used to receive data from a PETSc routine. For example,

```
PetscCall(VecGetValues(v, one, idx, [val], ierr))
```

is invalid and will not set `val` with the correct value.

Passing null pointers to PETSc functions

Many PETSc C functions have the option of passing a `NULL` 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`, `SCALAR`, `VEC`, `MAT`, etc depending on the argument type). For example, when no options prefix is desired in the routine `PetscOptionsGetInt()`, one must use the following command in Fortran:

```
PetscCall(PetscOptionsGetInt(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, PETSC_NULL_
↪CHARACTER, '-name', N, flg, ierr))
```

Where the code expects an array, then use `PETSC_NULL_XXX_ARRAY`. For example:

```
PetscCall(MatCreateSeqDense(comm, m, n, PETSC_NULL_SCALAR_ARRAY, A))
```

When a PETSc function returns multiple arrays, such as `DMDAGetOwnershipRanges()` and the user does not need certain arrays they must pass `PETSC_NULL_XXX_POINTER` as the argument. For example,

```
PetscInt, pointer :: lx(:), ly(:)
PetscCallA(DMDAGetOwnershipRanges(dm, lx, ly, PETSC_NULL_INTEGER_POINTER, ierr))
PetscCallA(DMDARestoreOwnershipRanges(dm, lx, ly, PETSC_NULL_INTEGER_POINTER, ierr))
```

Arguments that are fully defined Fortran derived types (C structs), such as `MatFactorInfo` or `PetscSFNode`, cannot be passed as null from Fortran. A properly defined variable must be passed in for those arguments.

Finally when a subroutine returns a `PetscObject` through an argument, to check if it is `NULL` you must use:

```
if (PetscObjectIsNull(dm)) then
if (.not. PetscObjectIsNull(dm)) then
```

you cannot use

```
if (dm .eq. PETSC_NULL_DM) then
```

Note that

```
if (PetscObjectIsNull(PETSC_NULL_VEC)) then
```

will always return true, for any PETSc object.

These specializations with `NULL` types are required because of Fortran's strict type checking system and lack of a concept of `NULL`, the Fortran compiler will often warn you if the wrong `NULL` type is passed.

Output Arrays from PETSc functions

For PETSc routine arguments that return an array of `PetscInt`, `PetscScalar`, `PetscReal` or of PETSc objects, one passes in a pointer to an array and the PETSc routine returns an array containing the values. For example,

```
PetscScalar *a;
Vec         v;
VecGetArray(v, &a);
```

is in Fortran,

```
PetscScalar, pointer :: a(:)
Vec,         v
VecGetArray(v, a, ierr)
```

For PETSc routine arguments that return a character string (array), e.g. `const char *str[]` pass a string long enough to hold the result. For example,

```
character*(80)  str
PetscCall(KSPGetType(ksp,str,ierr))
```

The result is copied into `str`.

Similarly, for PETSc routines where the user provides a character array (to be filled) followed by the array's length, e.g. `char name[], size_t nlen`. In Fortran pass a string long enough to hold the result, but not the separate length argument. For example,

```
character*(80)  str
PetscCall(PetscGetHostName(name,ierr))
```

Matrix, Vector and IS Indices

All matrices, vectors and `IS` in PETSc use zero-based indexing in the PETSc API regardless of whether C or Fortran is being used. For example, `MatSetValues()` and `VecSetValues()` always use zero indexing. See *Basic Matrix Operations* for further details.

Indexing into Fortran arrays, for example obtained with `VecGetArray()`, uses the Fortran convention and generally begin with 1 except for special routines such as `DMDAVecGetArray()` which uses the ranges provided by `DMDAGetCorners()`.

Setting Routines and Contexts

Some PETSc functions take as arguments user-functions and contexts for the function. For example

```
external func
SNESSetFunction(snes, r, func, ctx, ierr)
SNES snes
Vec r
PetscErrorCode ierr
```

where `func` has the calling sequence

```
subroutine func(snes, x, f, ctx, ierr)
SNES snes
Vec x,f
PetscErrorCode ierr
```

and `ctx` can be almost anything (represented as `void *` in C).

In Fortran, it has to be a derived type as in

```
subroutine func(snes, x, f, ctx, ierr)
SNES snes
Vec x,f
type (AppCtx)   ctx
PetscErrorCode ierr
...

external func
SNESSetFunction(snes, r, func, ctx, ierr)
SNES snes
Vec r
PetscErrorCode ierr
type (AppCtx)   ctx
```

or a PETSc object

```
subroutine func(snes, x, f, ctx, ierr)
SNES snes
Vec x,f
Vec ctx
PetscErrorCode ierr
...

external func
SNESSetFunction(snes, r, func, ctx, ierr)
SNES snes
Vec r
PetscErrorCode ierr
Vec ctx
```

or nothing

```
subroutine func(snes, x, f, dummy, ierr)
SNES snes
Vec x,f
integer dummy(*)
PetscErrorCode ierr
```

(continues on next page)

(continued from previous page)

```
...

external func
SNESSetFunction(snes, r, func, 0, ierr)
SNES snes
Vec r
PetscErrorCode ierr
```

Certain PETSc functions return a context in an argument, for example, `SNESGetApplicationContext()`. In C, they are handled as a `void *` pointer so one can write code such as

```
AppCtx *ctx;
SNESGetApplicationContext(snes, &ctx);
```

In Fortran, they must be declared as a pointer with, for example,

```
type(AppCtx), pointer :: ctx
call SNESGetApplicationContext(snes, ctx)
```

But sadly, this alone will not work. One must also specifically tell the Fortran compiler in an interface definition that `SNESGetApplicationContext()` expects the `ctx` argument to be a pointer to `type(AppCtx)` since Fortran forbids pointers to unknown types. PETSc provides macros to provide this information easily using, for example,

```
Interface_SNESGetApplicationContext(AppCtx)
end interface
```

One must insert these lines into the Fortran source code where one inserts interface definitions, see, for example, src/snes/tests/ex590.F90. For those interested, the source code of these macros may be found in the generated Fortran include files located at `$PETSC_DIR/$PETSC_ARCH/include/petsc/finclude/*.h`.

When a function pointer (declared as external in Fortran) is passed as an argument to a PETSc function, it is assumed that this function references a routine written in the same language as the PETSc interface function that was called. For instance, if `SNESSetFunction()` is called from C, the function must be a C function. Likewise, if it is called from Fortran, the function must be (a subroutine) written in Fortran.

If you are using Fortran classes that have bound functions (methods) as in src/snes/tests/ex18f90.F90, the context cannot be passed to function pointer setting routines, such as `SNESSetFunction()`. Instead, one must use `SNESSetFunctionNoInterface()`, and define the interface directly in the user code, see ex18f90.F90 for a full demonstration.

Compiling and Linking Fortran Programs

See *Writing C/C++ or Fortran Applications*.

4.1.4 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.F90`

```
!
!
#include <petsc/finclude/petscvec.h>
program main
  use petscvec
  implicit none
!
!  This example demonstrates basic use of the PETSc Fortran interface
!  to vectors.
!
  PetscInt n
  PetscErrorCode ierr
  PetscBool flg
  PetscScalar dot
  PetscScalar, parameter :: one = 1.0, two = 2.0, three = 3.0
  PetscReal norm, rdot
  Vec x, y, w
  PetscOptions options

  PetscCallA(PetscInitialize(ierr))
  PetscCallA(PetscOptionsCreate(options, ierr))
  n = 20
  PetscCallA(PetscOptionsGetInt(options, PETSC_NULL_CHARACTER, '-n', n, flg, ierr))
  PetscCallA(PetscOptionsDestroy(options, ierr))

! Create a vector, then duplicate it
  PetscCallA(VecCreate(PETSC_COMM_WORLD, x, ierr))
  PetscCallA(VecSetSizes(x, PETSC_DECIDE, n, ierr))
  PetscCallA(VecSetFromOptions(x, ierr))
  PetscCallA(VecDuplicate(x, y, ierr))
  PetscCallA(VecDuplicate(x, w, ierr))

  PetscCallA(VecSet(x, one, ierr))
  PetscCallA(VecSet(y, two, ierr))

  PetscCallA(VecDot(x, y, dot, ierr))
  rdot = PetscRealPart(dot)
  write (6, 100) rdot
100 format('Result of inner product ', f10.4)

  PetscCallA(VecScale(x, two, ierr))
  PetscCallA(VecNorm(x, NORM_2, norm, ierr))
```

(continues on next page)

(continued from previous page)

```
  write (6, 110) norm
110 format('Result of scaling ', f10.4)

  PetscCallA(VecCopy(x, w, ierr))
  PetscCallA(VecNorm(w, NORM_2, norm, ierr))
  write (6, 120) norm
120 format('Result of copy ', f10.4)

  PetscCallA(VecAXPY(y, three, x, ierr))
  PetscCallA(VecNorm(y, NORM_2, norm, ierr))
  write (6, 130) norm
130 format('Result of axpy ', f10.4)

  PetscCallA(VecDestroy(x, ierr))
  PetscCallA(VecDestroy(y, ierr))
  PetscCallA(VecDestroy(w, ierr))
  PetscCallA(PetscFinalize(ierr))
end
```

Listing: **src/vec/vec/tutorials/ex4f.F90**

```
!
!
!  Description:  Illustrates the use of VecSetValues() to set
!  multiple values at once; demonstrates VecGetArray().
!
! -----------------------------------------------------------------------
#include <petsc/finclude/petscvec.h>
program main
  use petscvec
  implicit none

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PetscInt, parameter :: n = 6
  PetscScalar xwork(n)
  PetscScalar, pointer :: xx_v(:), yy_v(:)
  PetscInt i, loc(n)
  PetscErrorCode ierr
  Vec x, y

  PetscCallA(PetscInitialize(ierr))

! Create initial vector and duplicate it
  PetscCallA(VecCreateSeq(PETSC_COMM_SELF, n, x, ierr))
  PetscCallA(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 i = 1, n
    loc(i) = i - 1
```

(continues on next page)

(continued from previous page)

```
    xwork(i) = 10.0*real(i)
  end do

!  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).
  PetscCallA(VecSetValues(x, n, loc, xwork, INSERT_VALUES, ierr))

! Assemble vector
  PetscCallA(VecAssemblyBegin(x, ierr))
  PetscCallA(VecAssemblyEnd(x, ierr))

! View vector
  PetscCallA(PetscObjectSetName(x, 'initial vector:', ierr))
  PetscCallA(VecView(x, PETSC_VIEWER_STDOUT_SELF, ierr))
  PetscCallA(VecCopy(x, y, ierr))

!  Get a pointer to vector data.
!    - For default PETSc vectors, VecGetArray() returns a pointer to
!      the data array.  Otherwise, the routine is implementation dependent.
!    - You MUST call VecRestoreArray() when you no longer need access to
!      the array.
!    - Note that the Fortran interface to VecGetArray() differs from the
!      C version.  See the users manual for details.
  PetscCallA(VecGetArray(x, xx_v, ierr))
  PetscCallA(VecGetArray(y, yy_v, ierr))

! Modify vector data
  do i = 1, n
    xx_v(i) = 100.0*real(i)
    yy_v(i) = 1000.0*real(i)
  end do

! Restore vectors
  PetscCallA(VecRestoreArray(x, xx_v, ierr))
  PetscCallA(VecRestoreArray(y, yy_v, ierr))

! View vectors
  PetscCallA(PetscObjectSetName(x, 'new vector 1:', ierr))
  PetscCallA(VecView(x, PETSC_VIEWER_STDOUT_SELF, ierr))

  PetscCallA(PetscObjectSetName(y, 'new vector 2:', ierr))
  PetscCallA(VecView(y, PETSC_VIEWER_STDOUT_SELF, ierr))

! Free work space.  All PETSc objects should be destroyed when they
! are no longer needed.
  PetscCallA(VecDestroy(x, ierr))
  PetscCallA(VecDestroy(y, ierr))
  PetscCallA(PetscFinalize(ierr))
end
```

Listing: **src/sys/classes/draw/tests/ex5f.F90**

```
!
!
#include <petsc/finclude/petscsys.h>
#include <petsc/finclude/petscdraw.h>
program main
  use petscsys
  use petscdraw
  implicit none
!
!  This example demonstrates basic use of the Fortran interface for
!  PetscDraw routines.
!
  PetscDraw draw
  PetscDrawLG lg
  PetscDrawAxis axis
  PetscErrorCode ierr
  PetscBool flg
  integer4, parameter :: x = 0, y = 0
  integer4 width, height
  PetscReal xd, yd
  PetscInt i, n, w, h

  PetscCallA(PetscInitialize(ierr))

!  GetInt requires a PetscInt so have to do this ugly setting
  w = 400
  PetscCallA(PetscOptionsGetInt(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-width', w,
↪ flg, ierr))
  width = int(w, kind=kind(width))
  h = 300
  PetscCallA(PetscOptionsGetInt(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-height',␣
↪h, flg, ierr))
  height = int(h, kind=kind(height))
  n = 15
  PetscCallA(PetscOptionsGetInt(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-n', n,␣
↪flg, ierr))

  PetscCallA(PetscDrawCreate(PETSC_COMM_WORLD, PETSC_NULL_CHARACTER, PETSC_NULL_
↪CHARACTER, x, y, width, height, draw, ierr))
  PetscCallA(PetscDrawSetFromOptions(draw, ierr))

  PetscCallA(PetscDrawLGCreate(draw, 1_PETSC_INT_KIND, lg, ierr))
  PetscCallA(PetscDrawLGGetAxis(lg, axis, ierr))
  PetscCallA(PetscDrawAxisSetColors(axis, PETSC_DRAW_BLACK, PETSC_DRAW_RED, PETSC_
↪DRAW_BLUE, ierr))
  PetscCallA(PetscDrawAxisSetLabels(axis, 'toplabel', 'xlabel', 'ylabel', ierr))

  do i = 0, n - 1
    xd = real(i) - 5.0
    yd = xd**2
    PetscCallA(PetscDrawLGAddPoint(lg, xd, yd, ierr))
  end do
```

Listing: **src/snes/tutorials/ex1f.F90**

```
!
!  Description: Uses the Newton method to solve a two-variable system.
!
#include <petsc/finclude/petsc.h>
module ex1fmodule
  use petscvec
  use petscsnesdef
  use petscvec
  use petscmat
  implicit none

contains
!
! ---------------------------------------------------------------------
!
!  FormFunction - Evaluates nonlinear function, F(x).
!
!  Input Parameters:
!  snes - the SNES context
!  x - input vector
!  dummy - optional user-defined context (not used here)
!
!  Output Parameter:
!  f - function vector
!
  subroutine FormFunction(snes, x, f, dummy, ierr)
    SNES snes
    Vec x, f
    PetscErrorCode, intent(out) :: ierr
    integer dummy(*)

!  Declarations for use with local arrays
    PetscScalar, pointer :: lx_v(:), lf_v(:)

!  Get pointers to vector data.
!    - VecGetArray() returns a pointer to the data array.
!    - You MUST call VecRestoreArray() when you no longer need access to
!      the array.

    PetscCall(VecGetArrayRead(x, lx_v, ierr))
    PetscCall(VecGetArray(f, lf_v, ierr))

!  Compute function

    lf_v(1) = lx_v(1)*lx_v(1) + lx_v(1)*lx_v(2) - 3.0
    lf_v(2) = lx_v(1)*lx_v(2) + lx_v(2)*lx_v(2) - 6.0

!  Restore vectors

    PetscCall(VecRestoreArrayRead(x, lx_v, ierr))
    PetscCall(VecRestoreArray(f, lf_v, ierr))

  end

! ---------------------------------------------------------------------
!
!  FormJacobian - Evaluates Jacobian matrix.
```

(continues on next page)

(continued from previous page)

```
!
!  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 matrix used to construct the preconditioner
!
  subroutine FormJacobian(snes, X, jac, B, dummy, ierr)

    SNES snes
    Vec X
    Mat jac, B
    PetscScalar A(4)
    PetscErrorCode, intent(out) :: ierr
    PetscInt idx(2)
    integer dummy(*)

!  Declarations for use with local arrays

    PetscScalar, pointer :: lx_v(:)

!  Get pointer to vector data

    PetscCall(VecGetArrayRead(x, lx_v, 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 = [0, 1]
    A = [2.0*lx_v(1) + lx_v(2), lx_v(1), lx_v(2), lx_v(1) + 2.0*lx_v(2)]
    PetscCall(MatSetValues(B, 2_PETSC_INT_KIND, idx, 2_PETSC_INT_KIND, idx, A, INSERT_
↪VALUES, ierr))

!  Restore vector

    PetscCall(VecRestoreArrayRead(x, lx_v, ierr))

!  Assemble matrix

    PetscCall(MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY, ierr))
    PetscCall(MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY, ierr))
    if (B /= jac) then
      PetscCall(MatAssemblyBegin(jac, MAT_FINAL_ASSEMBLY, ierr))
      PetscCall(MatAssemblyEnd(jac, MAT_FINAL_ASSEMBLY, ierr))
    end if

  end

  subroutine MyLineSearch(linesearch, lctx, ierr)
```

(continues on next page)

(continued from previous page)

```
    SNESLineSearch linesearch
    SNES snes
    integer lctx
    Vec x, f, g, y, w
    PetscReal ynorm, gnorm, xnorm
    PetscErrorCode, intent(out) :: ierr

    PetscScalar, parameter :: mone = -1.0

    PetscCall(SNESLineSearchGetSNES(linesearch, snes, ierr))
    PetscCall(SNESLineSearchGetVecs(linesearch, x, f, y, w, g, ierr))
    PetscCall(VecNorm(y, NORM_2, ynorm, ierr))
    PetscCall(VecAXPY(x, mone, y, ierr))
    PetscCall(SNESComputeFunction(snes, x, f, ierr))
    PetscCall(VecNorm(f, NORM_2, gnorm, ierr))
    PetscCall(VecNorm(x, NORM_2, xnorm, ierr))
    PetscCall(VecNorm(y, NORM_2, ynorm, ierr))
    PetscCall(SNESLineSearchSetNorms(linesearch, xnorm, gnorm, ynorm, ierr))
  end
end module

program main
  use petsc
  use ex1fmodule
  implicit none

!  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                  Variable declarations
!  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     snes        - nonlinear solver
!     ksp        - linear solver
!     pc          - preconditioner context
!     ksp         - Krylov subspace method context
!     x, r        - solution, residual vectors
!     J           - Jacobian matrix
!     its         - iterations for convergence
!
  SNES snes
  PC pc
  KSP ksp
  Vec x, r
  Mat J
  SNESLineSearch linesearch
  PetscErrorCode ierr
  PetscInt its
  PetscMPIInt size, rank
  PetscScalar, parameter :: pfive = 0.5
  PetscReal, parameter :: tol = 1.e-4
  PetscBool setls
  PetscReal, pointer :: rhistory(:)
  PetscInt, pointer :: itshistory(:)
  PetscInt nhistory
#if defined(PETSC_USE_LOG)
  PetscViewer viewer
```

(continues on next page)

(continued from previous page)

```
#endif
  double precision threshold, oldthreshold

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PetscCallA(PetscInitialize(ierr))
  PetscCallA(PetscLogNestedBegin(ierr))
  threshold = 1.0
  PetscCallA(PetscLogSetThreshold(threshold, oldthreshold, ierr))
  PetscCallMPIA(MPI_Comm_size(PETSC_COMM_WORLD, size, ierr))
  PetscCallMPIA(MPI_Comm_rank(PETSC_COMM_WORLD, rank, ierr))
  PetscCheckA(size == 1, PETSC_COMM_SELF, PETSC_ERR_WRONG_MPI_SIZE, 'Uniprocessor␣
↪example')

! - - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Create nonlinear solver context
! - - - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -

  PetscCallA(SNESCreate(PETSC_COMM_WORLD, snes, ierr))

  PetscCallA(SNESSetConvergenceHistory(snes, PETSC_NULL_REAL_ARRAY, PETSC_NULL_
↪INTEGER_ARRAY, PETSC_DECIDE, PETSC_FALSE, ierr))

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Create matrix and vector data structures; set corresponding routines
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Create vectors for solution and nonlinear function

  PetscCallA(VecCreateSeq(PETSC_COMM_SELF, 2_PETSC_INT_KIND, x, ierr))
  PetscCallA(VecDuplicate(x, r, ierr))

!  Create Jacobian matrix data structure

  PetscCallA(MatCreate(PETSC_COMM_SELF, J, ierr))
  PetscCallA(MatSetSizes(J, PETSC_DECIDE, PETSC_DECIDE, 2_PETSC_INT_KIND, 2_PETSC_INT_
↪KIND, ierr))
  PetscCallA(MatSetFromOptions(J, ierr))
  PetscCallA(MatSetUp(J, ierr))

!  Set function evaluation routine and vector

  PetscCallA(SNESSetFunction(snes, r, FormFunction, 0, ierr))

!  Set Jacobian matrix data structure and Jacobian evaluation routine

  PetscCallA(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
```

(continues on next page)

(continued from previous page)

```
!  directly call any KSP, KSP, and PC routines to set various options.

  PetscCallA(SNESGetKSP(snes, ksp, ierr))
  PetscCallA(KSPGetPC(ksp, pc, ierr))
  PetscCallA(PCSetType(pc, PCNONE, ierr))
  PetscCallA(KSPSetTolerances(ksp, tol, PETSC_CURRENT_REAL, PETSC_CURRENT_REAL, 20_
↪PETSC_INT_KIND, 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.

  PetscCallA(SNESSetFromOptions(snes, ierr))

  PetscCallA(PetscOptionsHasName(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-setls',␣
↪setls, ierr))

  if (setls) then
    PetscCallA(SNESGetLineSearch(snes, linesearch, ierr))
    PetscCallA(SNESLineSearchSetType(linesearch, 'shell', ierr))
    PetscCallA(SNESLineSearchShellSetApply(linesearch, MyLineSearch, 0, ierr))
  end if

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Evaluate initial guess; then solve nonlinear system
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Note: The user should initialize the vector, x, with the initial guess
!  for the nonlinear solver prior to calling SNESSolve().  In particular,
!  to employ an initial guess of zero, the user should explicitly set
!  this vector to zero by calling VecSet().

  PetscCallA(VecSet(x, pfive, ierr))
  PetscCallA(SNESSolve(snes, PETSC_NULL_VEC, x, ierr))

  PetscCallA(SNESGetConvergenceHistory(snes, rhistory, itshistory, nhistory, ierr))
  PetscCallA(SNESRestoreConvergenceHistory(snes, rhistory, itshistory, nhistory,␣
↪ierr))

! View solver converged reason; we could instead use the option -snes_converged_reason
  PetscCallA(SNESConvergedReasonView(snes, PETSC_VIEWER_STDOUT_WORLD, ierr))

  PetscCallA(SNESGetIterationNumber(snes, its, ierr))
  if (rank == 0) then
    write (6, 100) its
  end if
100 format('Number of SNES iterations = ', i5)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  PetscCallA(VecDestroy(x, ierr))
```

(continues on next page)

(continued from previous page)

```
  PetscCallA(VecDestroy(r, ierr))
  PetscCallA(MatDestroy(J, ierr))
  PetscCallA(SNESDestroy(snes, ierr))
#if defined(PETSC_USE_LOG)
  PetscCallA(PetscViewerASCIIOpen(PETSC_COMM_WORLD, 'filename.xml', viewer, ierr))
  PetscCallA(PetscViewerPushFormat(viewer, PETSC_VIEWER_ASCII_XML, ierr))
  PetscCallA(PetscLogView(viewer, ierr))
  PetscCallA(PetscViewerDestroy(viewer, ierr))
#endif
  PetscCallA(PetscFinalize(ierr))
end
```

Calling Fortran Routines from C (and C Routines from Fortran)

The information here applies only if you plan to call your **own** C functions from Fortran or Fortran functions from C. Different compilers have different methods of naming Fortran routines called from C (or C routines called from Fortran). Most Fortran compilers change the capital letters in Fortran routines to all lowercase. With some compilers, the Fortran compiler appends an underscore to the end of each Fortran routine name; for example, the Fortran routine `Dabsc()` would be called from C with `dabsc_()`. Other compilers 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 will be defined in the file `$PETSC_DIR/$PETSC_ARCH/include/petscconf.h` based on the compilers conventions. The macros are used, for example, as follows:

```
#if defined(PETSC_HAVE_FORTRAN_CAPS)
#define dabsc_ DABSC
#elif !defined(PETSC_HAVE_FORTRAN_UNDERSCORE)
#define dabsc_ dabsc
#endif
.....
dabsc_( &n,x,y); /* call the Fortran function */
```

4.2 Checking the PETSc version

The PETSc version is defined in `$PETSC_DIR/include/petscversion.h` with the three macros `PETSC_VERSION_MAJOR`, `PETSC_VERSION_MINOR`, and `PETSC_VERSION_SUBMINOR`.

The shell commands `make getversion` or `$PETSC_DIR/lib/petsc/bin/petscversion` prints out the PETSc version. The command

```
$ $PETSC_DIR/lib/petsc/bin/petscversion <eq,gt,lt,ge,le> major.minor<.subminor>
```

allows one to add tests to make files, CMake files, configure scripts etc, to ensure the PETSc version is compatible with your applications. For example,

```
$ $PETSC_DIR/lib/petsc/bin/petscversion eq 3.22
```

returns 1 if the PETSc version is 3.22 (any subminor version is allowed). While

```
$ $PETSC_DIR/lib/petsc/bin/petscversion ge 3.21
```

returns 1 if the PETSc version is 3.21 or higher.

Though we try to avoid making changes to the PETSc API, they are inevitable; thus we provide tools to help manage one's application to be robust to such changes.

4.2.1 During configure/make time

The command

```
$ $PETSC_DIR/lib/petsc/bin/petscversion eq xxx.yyy[.zzz]
```

prints out 1 if the PETSc version matches `xxx.yyy[.zzz]` and 0 otherwise. The command works in a similar way for `lt`, `le`, `gt`, and `ge`. This allows your application configure script, or `makefile` or `CMake` file to check if the PETSc version is compatible with application even before beginning to compile your code.

4.2.2 During compile time

The CPP macros

- `PETSC_VERSION_EQ(MAJOR,MINOR,SUBMINOR)`
- `PETSC_VERSION_LT(MAJOR,MINOR,SUBMINOR)`
- `PETSC_VERSION_LE(MAJOR,MINOR,SUBMINOR)`
- `PETSC_VERSION_GT(MAJOR,MINOR,SUBMINOR)`
- `PETSC_VERSION_GE(MAJOR,MINOR,SUBMINOR)`

may be used in the source code to choose different code paths or error out depending on the PETSc version.

4.2.3 At Runtime

The command

```
char  version(lengthofversion);
PetscErrorCode PetscGetVersion(char version[], size_t lengthofversion)
```

gives access to the version at runtime.

4.3 Using MATLAB with PETSc

There are three basic ways to use MATLAB with PETSc:

1. *Dumping Data for MATLAB* into files to be read into MATLAB,
2. *Sending Data to an Interactive MATLAB Session* 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* to send 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).

For the latter two approaches one must `./configure` PETSc with the argument `--with-matlab [--with-matlab-dir=matlab_root_directory]`.

4.3.1 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 `PETSCVIEWERASCII` (see `PetscViewerASCIIOpen()`, `PETSC_VIEWER_STDOUT_WORLD`, `PETSC_VIEWER_STDOUT_SELF`, or `PETSC_VIEWER_STDOUT_(MPI_Comm)`). Before calling the viewer set the output type with, for example,

```
PetscViewerPushFormat(PETSC_VIEWER_STDOUT_WORLD,PETSC_VIEWER_ASCII_MATLAB);
VecView(A,PETSC_VIEWER_STDOUT_WORLD);
PetscViewerPopFormat(PETSC_VIEWER_STDOUT_WORLD);
```

The name of each PETSc variable printed for MATLAB may be set with

```
PetscObjectSetName((PetscObject)A,"name");
```

If no name is specified, the object is given a default name using `PetscObjectName()`.

Dumping Binary Data for MATLAB

One can also read PETSc binary files (see *Viewers: Looking at PETSc Objects*) directly into MATLAB via the scripts available in `$PETSC_DIR/share/petsc/matlab`. This requires less disk space and is recommended for all but the smallest data sizes. One can also use

```
PetscViewerPushFormat(viewer,PETSC_VIEWER_BINARY_MATLAB)
```

to dump both a PETSc binary file and a corresponding `.info` file which `PetscReadBinaryMatlab.m` will use to format the binary file in more complex cases, such as using a `DMDA`. For an example, see DM Tutorial ex7. In MATLAB one may then generate a useful structure. For example:

```
setenv('PETSC_DIR','~/petsc');
setenv('PETSC_ARCH','arch-darwin-double-debug');
addpath('~/petsc/share/petsc/matlab');
gridData=PetscReadBinaryMatlab('output_file');
```

4.3.2 Sending Data to an Interactive MATLAB Session

One creates a viewer to MATLAB via

```
PetscViewerSocketOpen(MPI_Comm,char *machine,int port,PetscViewer *v);
```

(`port` is usually set to `PETSC_DEFAULT`; use `NULL` for the machine if the MATLAB interactive session is running on the same machine as the PETSc program) and then sends matrices or vectors via

```
VecView(Vec A,v);
MatView(Mat B,v);
```

See *Viewers: Looking at PETSc Objects* for more on PETSc viewers. One may start the MATLAB program manually or use the PETSc command `PetscStartMatlab(MPI_Comm,char *machine,char *script,FILE **fp)`; where `machine` and `script` may be `NULL`. It is also possible to start your PETSc program from MATLAB via `launch()`.

To receive the objects in MATLAB, make sure that `$PETSC_DIR/$PETSC_ARCH/lib/petsc/matlab` and `$PETSC_DIR/share/petsc/matlab` are in the MATLAB path. Use `p = PetscOpenSocket();` (or `p = PetscOpenSocket(portnum)` if you provided a port number in your call to `PetscViewerSocketOpen()`), and then `a = PetscBinaryRead(p);` returns the object passed from PETSc. `PetscBinaryRead()` may be called any number of times. Each call should correspond on the PETSc side with viewing a single vector or matrix. `close()` closes the connection from MATLAB. On the PETSc side, one should destroy the viewer object with `PetscViewerDestroy()`.

For an example, which includes sending data back to PETSc, see Vec Tutorial ex42 and the associated `.m` file.

4.3.3 Using the MATLAB Compute Engine

One creates access to the MATLAB engine via

```
PetscMatlabEngineCreate(MPI_Comm comm,char *machine,PetscMatlabEngine *e);
```

where `machine` is the name of the machine hosting MATLAB (`NULL` may be used for localhost). One can send objects to MATLAB via

```
PetscMatlabEnginePut(PetscMatlabEngine e,PetscObject obj);
```

One can get objects via

```
PetscMatlabEngineGet(PetscMatlabEngine e,PetscObject obj);
```

Similarly, one can send arrays via

```
PetscMatlabEnginePutArray(PetscMatlabEngine e,int m,int n,PetscScalar *array,char␣
↪*name);
```

and get them back via

```
PetscMatlabEngineGetArray(PetscMatlabEngine e,int m,int n,PetscScalar *array,char␣
↪*name);
```

One cannot use MATLAB interactively in this mode but one can send MATLAB commands via

```
PetscMatlabEngineEvaluate(PetscMatlabEngine,"format",...);
```

where `format` has the usual `printf()` format. For example,

```
PetscMatlabEngineEvaluate(PetscMatlabEngine,"x = \%g *y + z;",avalue);
```

The name of each PETSc variable passed to MATLAB may be set with

```
PetscObjectSetName((PetscObject)A,"name");
```

Text responses can be returned from MATLAB via

```
PetscMatlabEngineGetOutput(PetscMatlabEngine,char **);
```

or

```
PetscMatlabEnginePrintOutput(PetscMatlabEngine,FILE*).
```

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 be 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.

4.3.4 Licensing the MATLAB Compute Engine on a cluster

To activate MATLAB on head node which does not have access to the internet.[1]

First ssh into the head node using the command: ssh node_name

Obtain the Host Id using the command: ip addr | grep ether[2] You will see something like this: link/ether xx:xx:xx:xx:xx:xx ABC yy:yy:yy:yy:yy:yy Note the value: xx:xx:xx:xx:xx:xx

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.

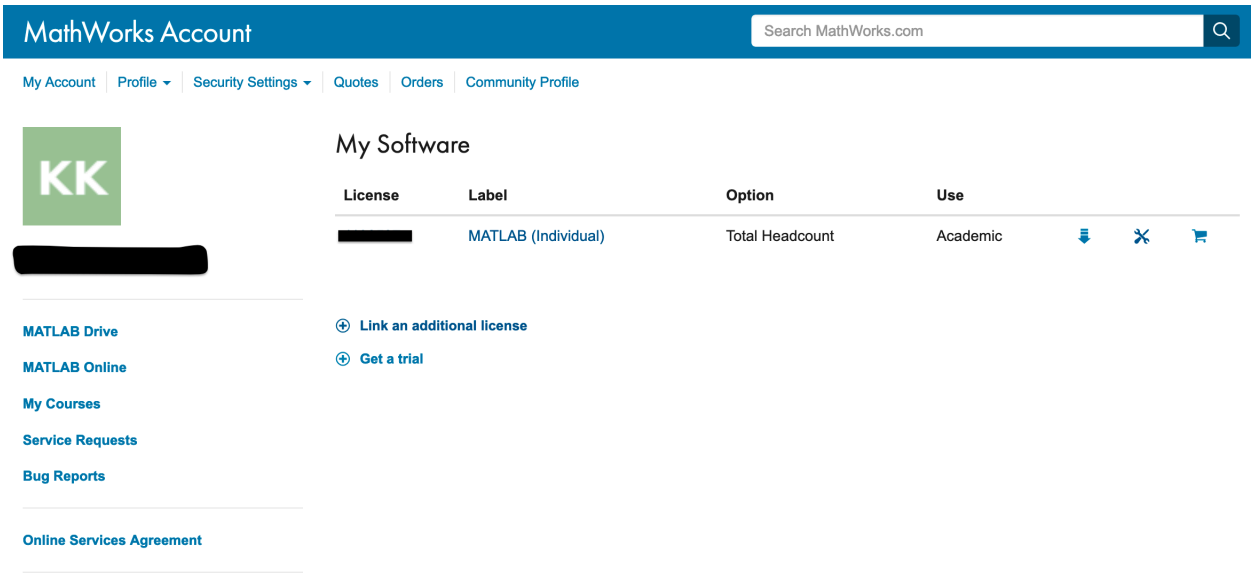
An option to download the License file will appear. Download it and copy the license file to the cluster (your home directory). Now, launch MATLAB where you have sshed into your head node.

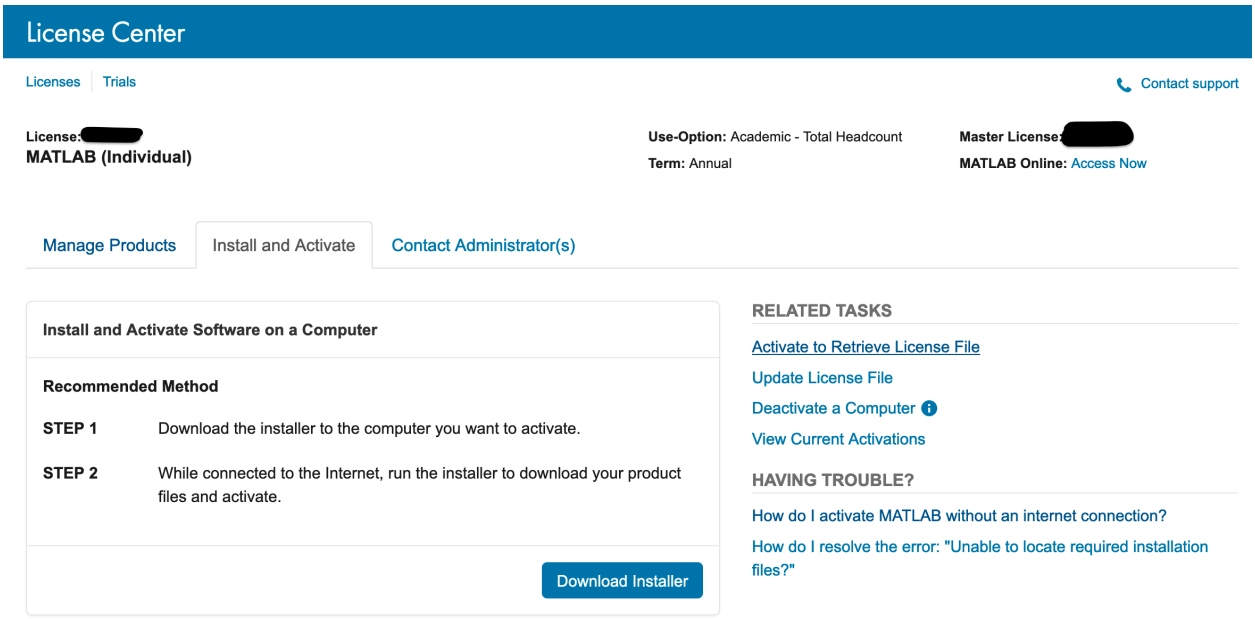
Select the Activate manually without the internet option and click Next >. Browse and locate the license file.

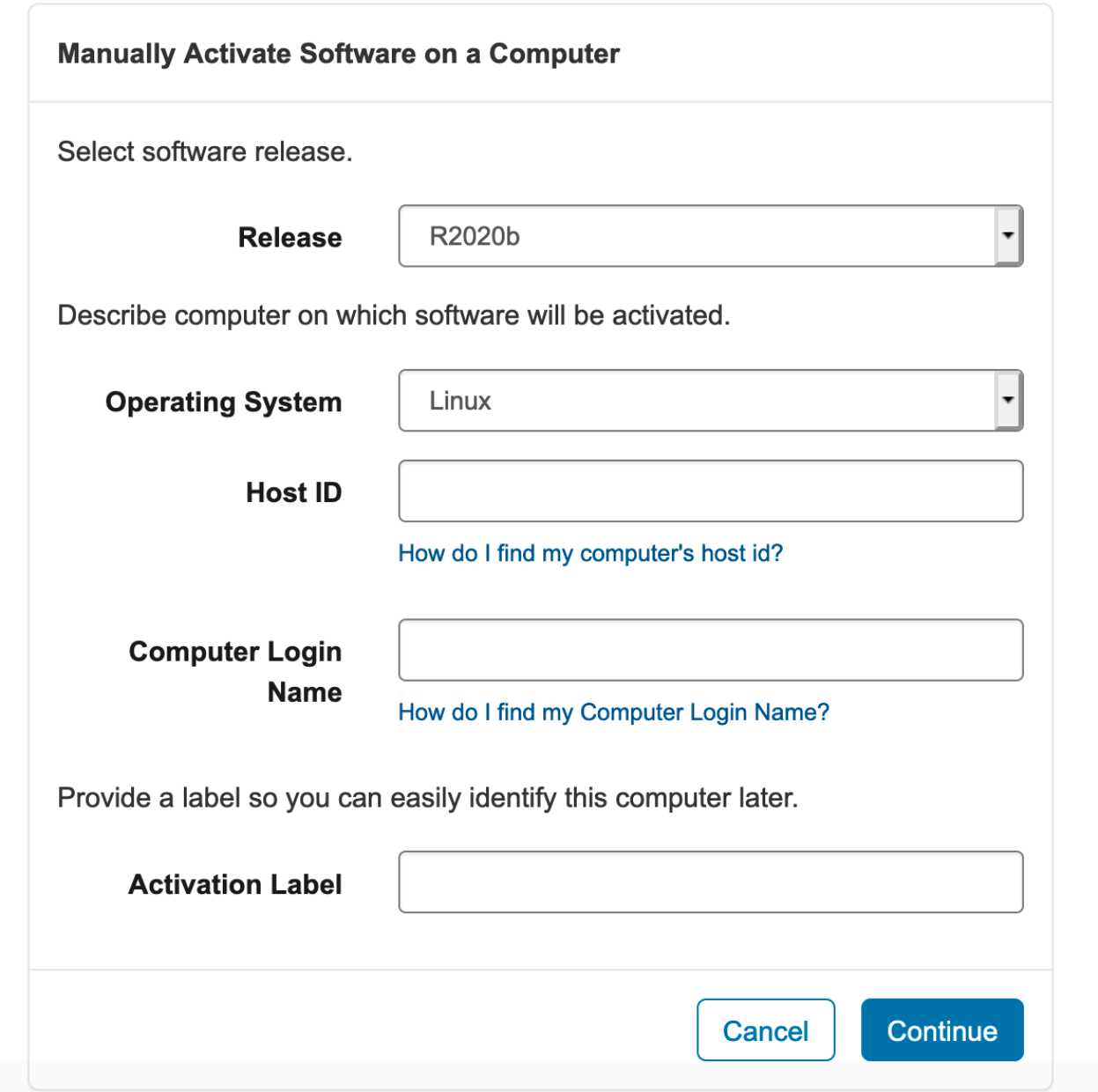
MATLAB is activated and ready to use.

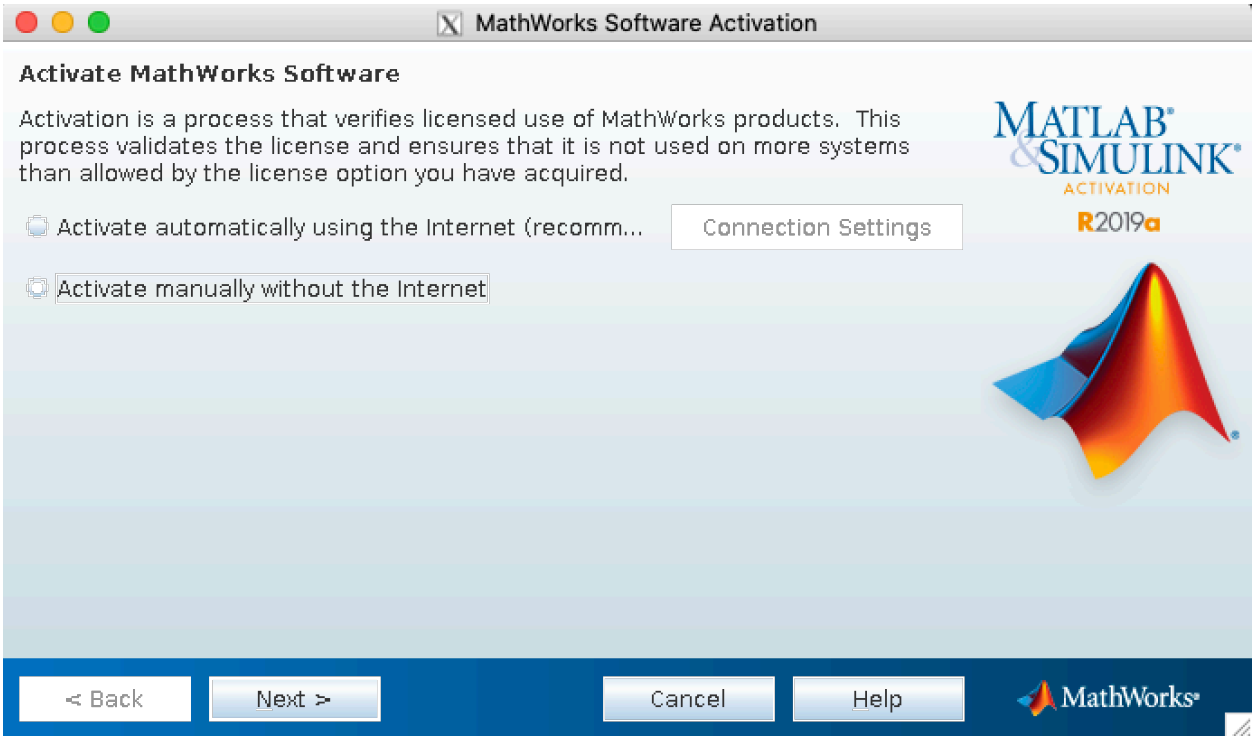
---

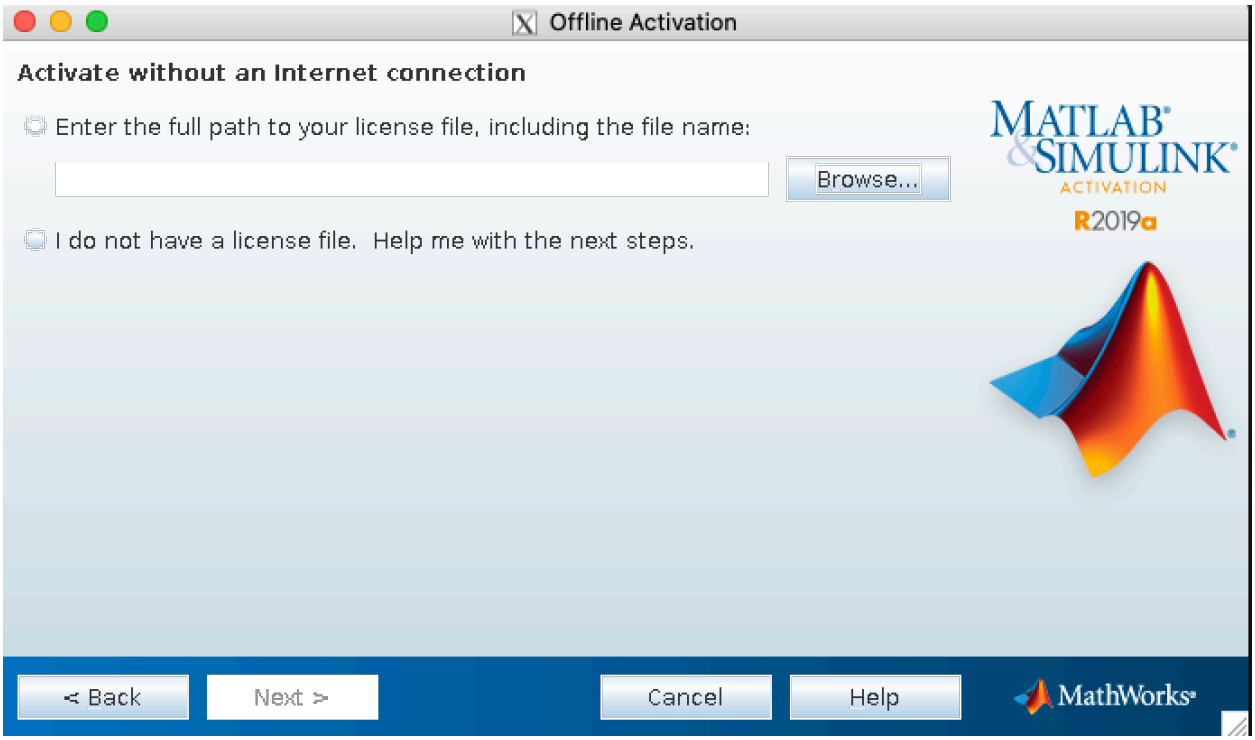
[1] https://www.mathworks.com/matlabcentral/answers/259627-how-do-i-activate-matlab-or-other-mathworks-products-without-an-internet-c
[2] http://www.mathworks.com/matlabcentral/answers/101892


MathWorks Account
Search MathWorks.com
My Account
Profile
Security Settings
Quotes
Orders
Community Profile
KK
[redacted]
MATLAB Drive
MATLAB Online
My Courses
Service Requests
Bug Reports
Online Services Agreement
My Software
License
Label
Option
Use
[redacted]
MATLAB (Individual)
Total Headcount
Academic
Link an additional license
Get a trial


License Center
Licenses
Trials
Contact support
License:[redacted]
MATLAB (Individual)
Use-Option: Academic - Total Headcount
Term: Annual
Master License:[redacted]
MATLAB Online: Access Now
Manage Products
Install and Activate
Contact Administrator(s)
Install and Activate Software on a Computer
Recommended Method
STEP 1
Download the installer to the computer you want to activate.
STEP 2
While connected to the Internet, run the installer to download your product files and activate.
Download Installer
RELATED TASKS
Activate to Retrieve License File
Update License File
Deactivate a Computer
View Current Activations
HAVING TROUBLE?
How do I activate MATLAB without an internet connection?
How do I resolve the error: "Unable to locate required installation files?"


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
Cancel
Continue


MathWorks Software Activation
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 (recomm...
Connection Settings
Activate manually without the Internet
MATLAB & SIMULINK ACTIVATION R2019a
< Back
Next >
Cancel
Help
MathWorks


Offline Activation
Activate without an Internet connection
Enter the full path to your license file, including the file name:
Browse...
I do not have a license file. Help me with the next steps.
MATLAB & SIMULINK ACTIVATION R2019a
< Back
Next >
Cancel
Help
MathWorks

4.4 Profiling

PETSc includes a consistent, lightweight scheme for profiling 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.

4.4.1 Basic Profiling Information

The profiling options include the following:

- `-log_view [:filename]` - Prints an ASCII version of performance data at the 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. See *Interpreting -log_view Output: The Basics*
- `-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. See *Interpreting -info Output: Informative Messages*
- `-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. See `PetscLogTraceBegin()`.

As discussed in *Using -log_mpe with Jumpshot*, additional profiling can be done with MPE.

Interpreting `-log_view` Output: The Basics

As shown in *listing* in *Profiling Programs*, the option `-log_view [:filename]` activates printing of profile data to standard output or an ASCII file at the conclusion of a program. See `PetscLogView()` for all the possible output options.

We print performance data for each routine, organized by PETSc libraries, followed by any user-defined events (discussed in *Profiling Application Codes*). For each routine, the output data include the maximum time and floating point operation (flop) rate over all processes. Information about parallel performance is also included, as discussed in the following section.

For the purpose of PETSc floating point operation counting, we define one *flop* as one operation of any of the following types: multiplication, division, addition, or subtraction. For example, one `VecAXPY()` operation, which computes $y = \alpha x + y$ for vectors of length N, requires $2N$ flop (consisting of N additions and N multiplications). Bear in mind that flop rates present only a limited view of performance, since memory loads and stores are the real performance barrier.

For simplicity, the remainder of this discussion focuses on interpreting profile data for the `KSP` library, which provides the linear solvers at the heart of the PETSc package. Recall the hierarchical organization of the PETSc library, as shown in *Numerical Libraries in PETSc*. Each `KSP` solver is composed of a `PC` (preconditioner) and a `KSP` (Krylov subspace) part, which are in turn built on top of the `Mat` (matrix) and `Vec` (vector) modules. Thus, operations in the `KSP` module are composed of lower-level operations in these packages. Note also that the nonlinear solvers library, `SNES`, is built on top of the `KSP` module, and the timestepping library, `TS`, is in turn built on top of `SNES`.

We briefly discuss interpretation of the sample output in *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 matrix
      - `MatILUFactorSymbolic` - Symbolic factorization phase
      - `MatLUFactorNumeric` - Numeric factorization phase
- `KSPSolve` - Solve linear system
  - `PCApply` - Apply preconditioner
    - `MatSolve` - Forward/backward triangular solves
  - `KSPGMRESOrthog` - Orthogonalization in GMRES
    - `VecDot` or `VecMDot` - Inner products
    - `VecAXPY` or `VecMAXPY` - vector updates
  - `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 *listing* and *listing*, which presents the 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 obtained from datafiles, see `petsc_repositories`.

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 object creation and destruction is presented.

```
mpiexec -n 4 ./ex10 -f0 medium -f1 arco6 -ksp_gmres_classicalgramschmidt -log_view -
↪mat_type baij \
           -matload_block_size 3 -pc_type bjacobi -options_left

Number of iterations = 19
Residual norm 1.088292e-05
Number of iterations = 59
Residual norm 3.871022e-02
---------------------------------------------- PETSc Performance Summary: ------------
↪----------------------------------

./ex10 on a intel-bdw-opt named beboplogin4 with 4 processors, by jczhang Mon Apr 23␣
↪13:36:54 2018
Using PETSc Development Git Revision: v3.9-163-gbe3efd42 Git Date: 2018-04-16␣
↪10:45:40 -0500

                         Max       Max/Min        Avg      Total
Time (sec):           1.849e-01      1.00002   1.849e-01
Objects:              1.060e+02      1.00000   1.060e+02
Flops:                2.361e+08      1.00684   2.353e+08  9.413e+08
Flops/sec:            1.277e+09      1.00685   1.273e+09  5.091e+09
MPI Msg Count:        2.360e+02      1.34857   2.061e+02  8.245e+02
MPI Msg Len (bytes):  1.256e+07      2.24620   4.071e+04  3.357e+07
MPI Reductions:       2.160e+02      1.00000

Summary of Stages:   ----- Time ------  ----- Flop -----  --- Messages ---  --␣
↪Message Lengths --  -- Reductions --
                        Avg     %Total     Avg     %Total   counts   %Total     Avg  ␣
↪      %Total   counts   %Total
 0:      Main Stage: 5.9897e-04   0.3%  0.0000e+00   0.0%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 1:   Load System 0: 2.9113e-03   1.6%  0.0000e+00   0.0%  3.550e+01   4.3%  5.
↪984e+02        0.1%  2.200e+01  10.2%
 2:      KSPSetUp 0: 7.7349e-04   0.4%  9.9360e+03   0.0%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 3:      KSPSolve 0: 1.7690e-03   1.0%  2.9673e+05   0.0%  1.520e+02  18.4%  1.
↪800e+02        0.1%  3.900e+01  18.1%
 4:   Load System 1: 1.0056e-01  54.4%  0.0000e+00   0.0%  3.700e+01   4.5%  5.
↪657e+05       62.4%  2.200e+01  10.2%
 5:      KSPSetUp 1: 5.6883e-03   3.1%  2.1205e+07   2.3%  0.000e+00   0.0%  0.
↪000e+00        0.0%  2.000e+00   0.9%
 6:      KSPSolve 1: 7.2578e-02  39.3%  9.1979e+08  97.7%  6.000e+02  72.8%  2.
↪098e+04       37.5%  1.200e+02  55.6%

------------------------------------------------------------------------------------------
↪----------------------------------

.... [Summary of various phases, see part II below] ...

------------------------------------------------------------------------------------------
↪----------------------------------

Object Type          Creations   Destructions    (Reports information only for␣
↪process 0.)
...
--- Event Stage 3: KSPSolve 0
```

(continues on next page)

(continued from previous page)

```
              Matrix     0              4
              Vector    20             30
           Index Set     0              3
         Vec Scatter     0              1
       Krylov Solver     0              2
     Preconditioner     0              2
```

We next focus on the summaries for the various phases of the computation, as given in the table within the following *listing*. The summary for each phase presents the maximum times and flop rates over all processes, as well as the ratio of maximum to minimum times and flop rates for all processes. A ratio of approximately 1 indicates that computations within a given phase are well balanced among the processes; as the ratio increases, the balance becomes increasingly poor. Also, the total computational rate (in units of MFlop/sec) is given for each phase in the final column of the phase summary table.

$$\text{Total Mflop/sec} \;=\; 10^{-6} * (\text{sum of flop over all processors})/(\text{max time over all processors})$$

Note: Total computational rates < 1 MFlop are listed as 0 in this column of the phase summary table. Additional statistics for each phase include the total number of messages sent, the average message length, and the number of global reductions.

```
mpiexec -n 4 ./ex10 -f0 medium -f1 arco6 -ksp_gmres_classicalgramschmidt -log_view -
↪mat_type baij \
            -matload_block_size 3 -pc_type bjacobi -options_left

---------------------------------------------- PETSc Performance Summary: ------------
↪----------------------------------
.... [Overall summary, see part I] ...

Phase summary info:
   Count: number of times phase was executed
   Time and Flop/sec: Max - maximum over all processors
                      Ratio - ratio of maximum to minimum over all processors
   Mess: number of messages sent
   AvgLen: average message length
   Reduct: number of global reductions
   Global: entire computation
   Stage: optional user-defined stages of a computation. Set stages with␣
↪PetscLogStagePush() and PetscLogStagePop().
      %T - percent time in this phase         %F - percent flop in this phase
      %M - percent messages in this phase     %L - percent message lengths in this␣
↪phase
      %R - percent reductions in this phase
   Total Mflop/s: 10^6 * (sum of flop over all processors)/(max time over all␣
↪processors)
------------------------------------------------------------------------------------------
↪----------------------------------
Phase              Count      Time (sec)       Flop/sec                         ---␣
↪Global ---  --- Stage ----  Total
                          Max    Ratio      Max    Ratio  Mess AvgLen  Reduct  %T
↪%F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------
↪----------------------------------
...

--- Event Stage 5: KSPSetUp 1
```

(continues on next page)

(continued from previous page)

```
MatLUFactorNum         1 1.0 3.6440e-03 1.1 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  2 ␣
↪2  0  0  0  62100  0  0  0  5819
MatILUFactorSym        1 1.0 1.7111e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  1 ␣
↪0  0  0  0  26  0  0  0  0     0
MatGetRowIJ            1 1.0 1.1921e-06 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
MatGetOrdering         1 1.0 3.0041e-05 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   1  0  0  0  0     0
KSPSetUp               2 1.0 6.6495e-04 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 2.0e+00  0 ␣
↪0  0  0  1   9  0  0  0100     0
PCSetUp                2 1.0 5.4271e-03 1.2 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪2  0  0  0  90100  0  0  0  3907
PCSetUpOnBlocks        1 1.0 5.3999e-03 1.2 5.30e+06 1.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪2  0  0  0  90100  0  0  0  3927

--- Event Stage 6: KSPSolve 1

MatMult               60 1.0 2.4068e-02 1.1 6.54e+07 1.0 6.0e+02 2.1e+04 0.0e+00 12␣
↪27 73 37  0  32 28100100  0 10731
MatSolve              61 1.0 1.9177e-02 1.0 5.99e+07 1.0 0.0e+00 0.0e+00 0.0e+00 10␣
↪25  0  0  0  26 26  0  0  0 12491
VecMDot               59 1.0 1.4741e-02 1.3 4.86e+07 1.0 0.0e+00 0.0e+00 5.9e+01  7␣
↪21  0  0 27  18 21  0  0 49 13189
VecNorm               61 1.0 3.0417e-03 1.4 3.29e+06 1.0 0.0e+00 0.0e+00 6.1e+01  1 ␣
↪1  0  0 28   4  1  0  0 51  4332
VecScale              61 1.0 9.9802e-04 1.0 1.65e+06 1.0 0.0e+00 0.0e+00 0.0e+00  1 ␣
↪1  0  0  0   1  1  0  0  0  6602
VecCopy                2 1.0 5.9128e-05 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
VecSet                64 1.0 8.0323e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   1  0  0  0  0     0
VecAXPY                3 1.0 7.4387e-05 1.1 1.62e+05 1.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0  8712
VecMAXPY              61 1.0 8.8558e-03 1.1 5.18e+07 1.0 0.0e+00 0.0e+00 0.0e+00  5␣
↪22  0  0  0  12 23  0  0  0 23393
VecScatterBegin       60 1.0 9.6416e-04 1.8 0.00e+00 0.0 6.0e+02 2.1e+04 0.0e+00  0 ␣
↪0 73 37  0   1  0100100  0     0
VecScatterEnd         60 1.0 6.1543e-03 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3 ␣
↪0  0  0  0   8  0  0  0  0     0
VecNormalize          61 1.0 4.2675e-03 1.3 4.94e+06 1.0 0.0e+00 0.0e+00 6.1e+01  2 ␣
↪2  0  0 28   5  2  0  0 51  4632
KSPGMRESOrthog        59 1.0 2.2627e-02 1.1 9.72e+07 1.0 0.0e+00 0.0e+00 5.9e+01 11␣
↪41  0  0 27  29 42  0  0 49 17185
KSPSolve               1 1.0 7.2577e-02 1.0 2.31e+08 1.0 6.0e+02 2.1e+04 1.2e+02 39␣
↪98 73 37 56  99100100100100 12673
PCSetUpOnBlocks        1 1.0 9.5367e-07 0.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0 ␣
↪0  0  0  0   0  0  0  0  0     0
PCApply               61 1.0 2.0427e-02 1.0 5.99e+07 1.0 0.0e+00 0.0e+00 0.0e+00 11␣
↪25  0  0  0  28 26  0  0  0 11726
------------------------------------------------------------------------------------
↪----------------------------------
.... [Conclusion of overall summary, see part I] ...
```

As discussed in the preceding section, the performance summaries for higher-level PETSc routines include the statistics for the lower levels of which they are made up. For example, the communication within matrix-vector products `MatMult()` consists of vector scatter operations, as given by the routines `VecScatterBegin()` and `VecScatterEnd()`.

The final data presented are the percentages of the various statistics (time (`%T`), flop/sec (`%F`), messages(`%M`), average message length (`%L`), and reductions (`%R`)) for each event relative to the total computation and to any user-defined stages (discussed in *Profiling Multiple Sections of Code*). These statistics can aid in optimizing performance, since they indicate the sections of code that could benefit from various kinds of tuning. *Hints for Performance Tuning* gives suggestions about achieving good performance with PETSc codes.

The additional option `-log_view_memory` causes the display of additional columns of information about how much memory was allocated and freed during each logged event. This is useful to understand what phases of a computation require the most memory.

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`.

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 :[logfilename]:ascii_xml`. It can be viewed 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. A flamegraph can be visualized directly from stdout using, for example, `ImageMagick's display utility <https://imagemagick.org/script/display.php>`:

```
cd $PETSC_DIR/src/sys/tests
make ex30
mpiexec -n 2 ./ex30 -log_view ::ascii_flamegraph | flamegraph | display
```

Note that user-defined stages (see *Profiling Multiple Sections of Code*) will be ignored when using this nested format.

4.4.2 Profiling Application Codes

PETSc automatically logs object creation, times, and floating-point counts for the library routines. Users can easily supplement this information by profiling their application codes as well. The basic steps involved in logging a user-defined portion of code, called an *event*, are shown in the code fragment below:

```
PetscLogEvent  USER_EVENT;
PetscClassId   classid;
PetscLogDouble user_event_flops;

PetscClassIdRegister("class name",&classid);
PetscLogEventRegister("User event name",classid,&USER_EVENT);
PetscLogEventBegin(USER_EVENT,0,0,0,0);
/* code segment to monitor */
PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

One must register the event by calling `PetscLogEventRegister()`, which assigns a unique integer to identify the event for profiling purposes:

```
PetscLogEventRegister(const char string[],PetscClassId classid,PetscLogEvent *e);
```

Here `string` is a user-defined event name, and `color` is an optional user-defined event color (for use with *Jumpshot* logging; see *Using -log_mpe with Jumpshot*); one should see the manual page for details. The argument returned in `e` should then be passed to the `PetscLogEventBegin()` and `PetscLogEventEnd()` routines.

Events are logged by using the pair

```
PetscLogEventBegin(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject␣
↪o4);
PetscLogEventEnd(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject␣
↪o4);
```

The four objects are the PETSc objects that are most closely associated with the event. For instance, in a matrix-vector product they would be the matrix and the two vectors. These objects can be omitted by specifying 0 for `o1` - `o4`. The code between these two routine calls will be automatically timed and logged as part of the specified event.

Events are collective by default on the communicator of `o1` (if present). They can be made not collective by using `PetscLogEventSetCollective()`. No synchronization is performed on collective events in optimized builds unless the command line option `-log_sync` is used; however, we do check for collective semantics in debug mode.

The user can log the number of floating-point operations for this segment of code by calling

```
PetscLogFlops(number of flop for this code segment);
```

between the calls to `PetscLogEventBegin()` and `PetscLogEventEnd()`. This value will automatically be added to the global flop counter for the entire program.

4.4.3 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 several "stages" of code by switching among the various stages with the commands

```
PetscLogStagePush(PetscLogStage stage);
PetscLogStagePop();
```

see the manual pages for details. The command

```
PetscLogStageRegister(const char *name,PetscLogStage *stage)
```

allows one to associate a name with a stage; these names are printed whenever summaries are generated with `-log_view`. The following code fragment uses three profiling stages within an program.

```
PetscInitialize(int *argc,char ***args,0,0);
/* stage 0 of code here */
PetscLogStageRegister("Stage 0 of Code", &stagenum0);
for (i=0; i<ntimes; i++) {
    PetscLogStageRegister("Stage 1 of Code", &stagenum1);
    PetscLogStagePush(stagenum1);
    /* stage 1 of code here */
    PetscLogStagePop();
    PetscLogStageRegister("Stage 2 of Code", &stagenum2);
    PetscLogStagePush(stagenum2);
    /* stage 2 of code here */
    PetscLogStagePop();
}
PetscFinalize();
```

The listings above show output generated by `-log_view` for a program that employs several profiling stages. In particular, this program is subdivided into six stages: 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*.

4.4.4 Restricting Event Logging

By default, all PETSc operations are logged. To enable or disable the PETSc logging of individual events, one uses the commands

```
PetscLogEventActivate(PetscLogEvent event);
PetscLogEventDeactivate(PetscLogEvent event);
```

The `event` may be either a predefined PETSc event (as listed in the file `$PETSC_DIR/include/petsclog.h`) or one obtained with `PetscLogEventRegister()` (as described in *Profiling Application Codes*).

PETSc also provides routines that deactivate (or activate) logging for entire components of the library. Currently, the components that support such logging (de)activation are `Mat` (matrices), `Vec` (vectors), `KSP` (linear solvers, including `KSP` and `PC`), and `SNES` (nonlinear solvers):

```
PetscLogEventDeactivateClass(MAT_CLASSID);
PetscLogEventDeactivateClass(KSP_CLASSID); /* includes PC and KSP */
PetscLogEventDeactivateClass(VEC_CLASSID);
PetscLogEventDeactivateClass(SNES_CLASSID);
```

and

```
PetscLogEventActivateClass(MAT_CLASSID);
PetscLogEventActivateClass(KSP_CLASSID);   /* includes PC and KSP */
PetscLogEventActivateClass(VEC_CLASSID);
PetscLogEventActivateClass(SNES_CLASSID);
```

4.4.5 Interpreting `-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.

One can selectively turn off informative messages about any of the basic PETSc objects (e.g., `Mat`, `SNES`) with the command

```
PetscInfoDeactivateClass(int object_classid)
```

where `object_classid` is one of `MAT_CLASSID`, `SNES_CLASSID`, etc. Messages can be reactivated with the command

```
PetscInfoActivateClass(int object_classid)
```

Such deactivation can be useful when one wishes to view information about higher-level PETSc libraries (e.g., `TS` and `SNES`) without seeing all lower level data as well (e.g., `Mat`).

One can turn on or off logging for particular classes at runtime

```
-info [filename][:[~]<list,of,classnames>[:[~]self]]
```

The `list,of,classnames` is a list, separated by commas with no spaces, of classes one wishes to view the information on. For example `vec,ksp`. Information on all other classes will not be displayed. The ~ indicates to not display the list of classes but rather to display all other classes.

`self` indicates to display information on objects that are associated with `PETSC_COMM_SELF` while `~self` indicates to display information only for parallel objects.

See `PetscInfo()` for links to all the info operations that are available.

Application programmers can log their own messages, as well, by using the routine

```
PetscInfo(void* obj,char *message,...)
```

where `obj` is the PETSc object associated most closely with the logging statement, `message`. For example, in the line search Newton methods, we use a statement such as

```
PetscInfo(snes,"Cubic step, lambda %g\n",lambda);
```

4.4.6 Time

PETSc application programmers can access the wall clock time directly with the command

```
PetscLogDouble time;
PetscCall(PetscTime(&time));
```

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.

4.4.7 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.

4.4.8 Accurate Profiling and Paging Overheads

One factor that often plays a significant role in profiling a code is paging by the operating system. Generally, when running a program, only a few pages required to start it are loaded into memory rather than the entire executable. When the execution proceeds to code segments that are not in memory, a pagefault occurs, prompting the required pages to be loaded from the disk (a very slow process). This activity distorts the results significantly. (The paging effects are noticeable in the log files generated by `-log_mpe`, which is described in *Using -log_mpe with Jumpshot*.)

To eliminate the effects of paging when profiling the performance of a program, we have found an effective procedure is to run the *exact same code* on a small dummy problem before running it on the actual problem of interest. We thus ensure that all code required by a solver is loaded into memory during solution of the small problem. When the code proceeds to the actual (larger) problem of interest, all required pages have already been loaded into main memory, so that the performance numbers are not distorted.

When this procedure is used in conjunction with the user-defined stages of profiling described in *Profiling Multiple Sections of Code*, we can focus easily on the problem of interest. For example, we used this technique in the program KSP Tutorial ex10 to generate the timings within *listing* and *listing*. In this case, the profiled code of interest (solving the linear system for the larger problem) occurs within event stages 5 and 6. *Interpreting -log_view Output: Parallel Performance* provides details about interpreting such profiling data.

In particular, the macros

```
PetscPreLoadBegin(PetscBool flag,char* stagename)
PetscPreLoadStage(char *stagename)
```

and

```
PetscPreLoadEnd()
```

can be used to easily convert a regular PETSc program to one that uses preloading. The command line options `-preload true` and `-preload false` may be used to turn on and off preloading at run time for PETSc programs that use these macros.

4.4.9 NVIDIA Nsight Systems profiling

Nsight Systems will generate profiling data with a CUDA executable with the command `nsys`. For example, in serial

```
nsys profile -t nvtx,cuda -o file --stats=true --force-overwrite true ./a.out
```

will generate a file `file.qdstrm` with performance data that is annotated with PETSc events (methods) and Kokkos device kernel names. The Nsight Systems GUI, `nsys-ui`, can be used to navigate this file (https://developer.nvidia.com/nsight-systems). The Nsight Systems GUI lets you see a timeline of code performance information like kernels, memory mallocs and frees, CPU-GPU communication, and high-level data like time, sizes of memory copies, and more, in a popup window when the mouse hovers over the section. To view the data, start `nsys-ui` without any arguments and then `Import` the `.qdstrm` file in the GUI. A side effect of this viewing process is the generation of a file `file.nsys-rep`, which can be viewed directly with `nsys-ui` in the future.

For an MPI parallel job, only one process can call `nsys`, say have rank zero output `nsys` data and have all other ranks call the executable directly. For example with MPICH or Open MPI - we can run a parallel job on 4 MPI tasks as:

```
mpiexec -n 1 nsys profile -t nvtx,cuda -o file_name --stats=true --force-overwrite␣
↪true ./a.out : -n 3 ./a.out
```

Note: The Nsight GUI can open profiling reports from elsewhere. For example, a report from a compute node can be analyzed on your local machine, but care should be taken to use the exact same versions of Nsight Systems that generated the report. To check the version of Nsight on the compute node run `nsys-ui` and note the version number at the top of the window.

4.4.10 ROCProfiler profiling

For AMD GPUs, log events registered to PETSc can be displayed as ranges in trace files generated by `rocprof` by running with the `-log_roctx` flag. See the `rocprof` documentation for details of how to run the profiler. At the least, you will need:

```
mpiexec -n 1 rocprofv3 --marker-trace -o file_name -- ./path/to/application -log_roctx
```

4.4.11 Using TAU

TAU profiles can be generated without the need for instrumentation through the use of the perfstubs package. PETSc by default is configured with `--with-tau-perfstubs`. To generate profiles with TAU, first setup TAU:

```
wget http://tau.uoregon.edu/tau.tgz
./configure -cc=mpicc -c++=mpicxx -mpi -bfd=download -unwind=download && make install
export PATH=<tau dir>/x86_64/bin:$PATH
```

For more information on configuring TAU, see http://tau.uoregon.edu. Next, run your program with TAU. For instance, to profile `ex56`,

```
cd $PETSC_DIR/src/snes/tutorials
make ex56
mpiexec -n 4 tau_exec -T mpi ./ex56 -log_perfstubs <args>
```

This should produce four `profile.*` files with profile data that can be viewed with `paraprof/pprof`:

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
↪-
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
↪-
100.0           26        1,838           1       41322    1838424 .TAU application
 73.2            1        1,345           2         168     672950 SNESSolve
 62.2            3        1,142           2        1282     571442 SNESJacobianEval
 62.0        1,136        1,138           2          76     569494 DMPlexJacobianFE
 60.1        0.046        1,105           1          32    1105001 Solve 1
 15.2           87          279           5       11102      55943 Mesh Setup
 13.2        0.315          241           1          32     241765 Solve 0
  7.8           80          144       38785       38785          4 MPI_Allreduce()
  7.0           69          128           6       43386      21491 DualSpaceSetUp
  6.2            1          114           4          54      28536 PCSetUp
  6.0           12          110           2         892      55407 PCSetUp_GAMG+
  3.9           70           70           1           0      70888 MPI_Init_thread()
  3.7           68           68       41747           0          2 MPI Collective Sync
  3.6            8           66           4        3536      16548 SNESFunctionEval
  2.6           45           48         171         171        281 MPI_Bcast()
  1.9           34           34        7836           0          4 MPI_Barrier()
  1.8        0.567           33           2          68      16912  GAMG Coarsen
```

4.5 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.

4.5.1 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.

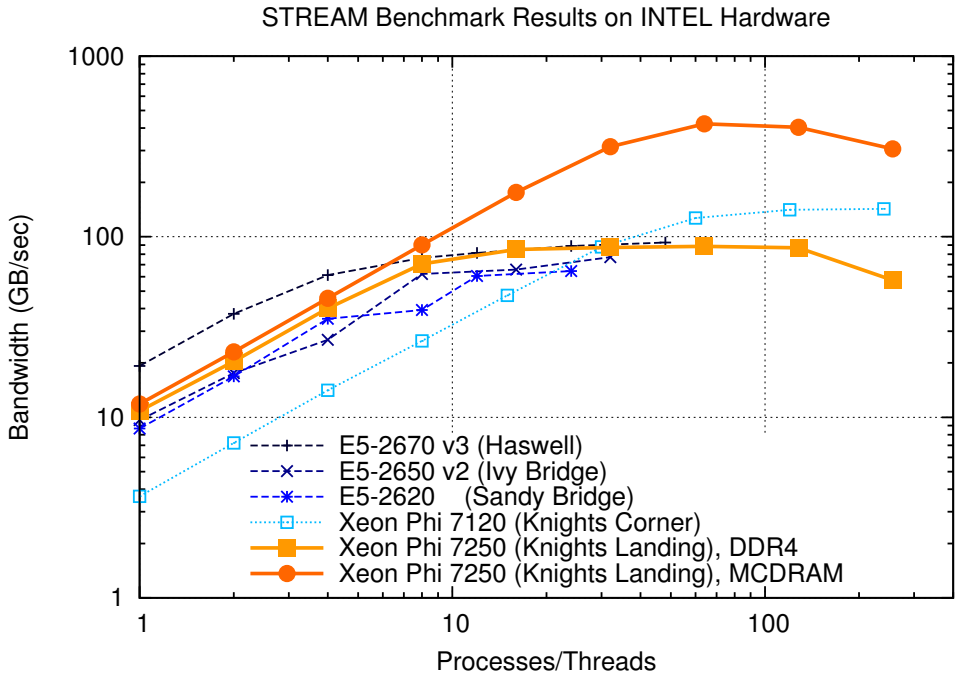


Fig. 4.1: Memory bandwidth obtained on Intel hardware (dual socket except KNL) over the number of processes used. One can get close to peak memory bandwidth with only a few processes.

As Fig. 4.1 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. 4.1. Consequently, running with more than 8 MPI ranks on such a system will not increase performance substantially. For the same reason, PETSc-based applications usually do not benefit from hyper-threading.

PETSc provides a simple way to measure memory bandwidth for different numbers of processes via the target `make streams` executed from `$PETSC_DIR`. The output provides an overview of the possible speedup one can obtain on the given machine (not necessarily a shared memory system). For example, the following is the most relevant output obtained on a dual-socket system equipped with two six-core-CPUs with hyperthreading:

```
np  speedup
1 1.0
2 1.58
3 2.19
4 2.42
5 2.63
6 2.69
...
21 3.82
22 3.49
```

(continues on next page)

(continued from previous page)

```
23 3.79
24 3.71
Estimation of possible speedup of MPI programs based on Streams benchmark.
It appears you have 1 node(s)
```

On this machine, one should expect a speed-up of typical memory bandwidth-bound PETSc applications of at most 4x when running multiple MPI ranks on the node. Most of the gains are already obtained when running with only 4-6 ranks. Because a smaller number of MPI ranks usually implies better preconditioners and better performance for smaller problems, the best performance for PETSc applications may be obtained with fewer ranks than there are physical CPU cores available.

Following the results from the above run of `make streams`, we recommend to use additional nodes instead of placing additional MPI ranks on the nodes. In particular, weak scaling (i.e. constant load per process, increasing the number of processes) and strong scaling (i.e. constant total work, increasing the number of processes) studies should keep the number of processes per node constant.

Non-Uniform Memory Access (NUMA) and Process Placement

CPUs in nodes with more than one CPU socket are internally connected via a high-speed fabric, cf. Fig. 4.2, 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.

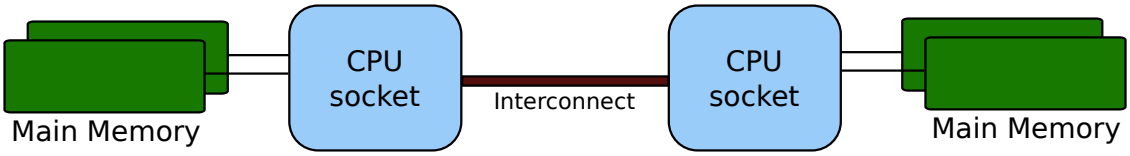


Fig. 4.2: Schematic of a two-socket NUMA system. Processes should be spread across both CPUs to obtain full bandwidth.

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 `mpiexec`. 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)
```

(continues on next page)

(continued from previous page)

```
        PU L#1 (P#12)
      L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#13)
      L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
        PU L#4 (P#2)
        PU L#5 (P#14)
      L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
        PU L#6 (P#3)
        PU L#7 (P#15)
      L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
        PU L#8 (P#4)
        PU L#9 (P#16)
      L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
        PU L#10 (P#5)
        PU L#11 (P#17)
  NUMANode L#1 (P#1 63GB)
    Package L#1 + L3 L#1 (15MB)
      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 Open MPI, where one needs to pass `--bind-to core --map-by socket` to `mpiexec`:

```
$ mpiexec -n 6 --bind-to core --map-by socket ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 000000100000000000100000
process 2 binding: 010000000000010000000000
process 3 binding: 000000010000000000010000
process 4 binding: 001000000000001000000000
process 5 binding: 000000001000000000001000
Triad:        45403.1949   Rate (MB/s)
```

In this configuration, process 0 is bound to the first physical core on the first socket (with IDs 0 and 12),

process 1 is bound to the first core on the second socket (IDs 6 and 18), and similarly for the remaining processes. The achieved bandwidth of 45 GB/sec is close to the practical peak of about 50 GB/sec available on the machine. If, however, all MPI processes are located on the same socket, memory bandwidth drops significantly:

```
$ mpiexec -n 6 --bind-to core --map-by core ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 010000000000010000000000
process 2 binding: 001000000000001000000000
process 3 binding: 000100000000000100000000
process 4 binding: 000010000000000010000000
process 5 binding: 000001000000000001000000
Triad:        25510.7507   Rate (MB/s)
```

All processes are now mapped to cores on the same socket. As a result, only the first memory channel is fully saturated at 25.5 GB/sec.

One must not assume that `mpiexec` uses good defaults. To demonstrate, compare the full output of `make streams` from *Memory Bandwidth vs. Processes* first, followed by the results 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
```

(continues on next page)

(continued from previous page)

```
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
24 4.22
```

For the non-optimized version on the left, the speedup obtained when using any number of processes between 3 and 13 is essentially constant up to fluctuations, indicating that all processes were by default executed on the same socket. Only with 14 or more processes, the speedup number increases again. In contrast, the results of

```
make streams
```

with proper processor placement shown second 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!

4.5.2 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, and 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=native`); otherwise, the executables built will not utilize the newer instructions sets and will not take advantage of the vector processing units.
- Beyond choosing the optimization levels, some value-unsafe optimizations (such as using reciprocals of values instead of dividing by those values, or allowing re-association of operands in a series of calculations) for floating point calculations may yield significant performance gains. Compilers often provide flags (e.g., `-ffast-math` in GCC) to enable a set of these optimizations, and they may be turned on when using options for very aggressive optimization (`-fast` or `-Ofast` in many compilers). These are worth exploring to maximize performance, but, if employed, it important to verify that these do not cause erroneous results with your code, since calculations may violate the IEEE standard for floating-point arithmetic.

Profiling

Users should not spend time optimizing a code until after having determined where it spends the bulk of its time on realistically sized problems. As discussed in detail in *Profiling*, the PETSc routines automatically log performance data if certain runtime options are specified.

To obtain a summary of where and how much time is spent in different sections of the code, use one of the following options:

- Run the code with the option `-log_view` to print a performance summary for various phases of the code.
- Run the code with the option `-log_mpe [logfilename]`, which creates a logfile of events suitable for viewing with Jumpshot (part of MPICH).

Then, focus on the sections where most of the time is spent. If you provided your own callback routines, e.g. for residual evaluations, search the profiling output for routines such as `SNESFunctionEval` or `SNESJacobianEval`. If their relative time is significant (say, more than 30 percent), consider optimizing these routines first. Generic instructions on how to optimize your callback functions are difficult; you may start by reading performance optimization guides for your system's hardware.

Aggregation

Performing operations on chunks of data rather than a single element at a time can significantly enhance performance because of cache reuse or lower data motion. Typical examples are:

- Insert several (many) elements of a matrix or vector at once, rather than looping and inserting a single value at a time. In order to access elements in of vector repeatedly, employ `VecGetArray()` to allow direct manipulation of the vector elements.
- When possible, use `VecMDot()` rather than a series of calls to `VecDot()`.
- If you require a sequence of matrix-vector products with the same matrix, consider packing your vectors into a single matrix and use matrix-matrix multiplications.
- Users should employ a reasonable number of `PetscMalloc()` calls in their codes. Hundreds or thousands of memory allocations may be appropriate; however, if tens of thousands are being used, then reducing the number of `PetscMalloc()` calls may be warranted. For example, reusing space or allocating large chunks and dividing it into pieces can produce a significant savings in allocation overhead. *Data Structure Reuse* gives details.

Aggressive aggregation of data may result in inflexible datastructures and code that is hard to maintain. We advise users to keep these competing goals in mind and not blindly optimize for performance only.

Memory Allocation for Sparse Matrix 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 `MatFactorInfo` structure when calling `MatLUFactorSymbolic()` or `MatILUFactorSymbolic()` can reduce greatly the number of mallocs and copies required, and thus greatly improve the performance of the factorization. One way to determine a good value for the fill parameter is to run a program with the option `-info`. The symbolic factorization phase will then print information such as

```
Info:MatILUFactorSymbolic_SeqAIJ:Reallocs 12 Fill ratio:given 1 needed 2.16423
```

This indicates that the user should have used a fill estimate factor of about 2.17 (instead of 1) to prevent the 12 required mallocs and copies. The command line option

```
-pc_factor_fill 2.17
```

will cause PETSc to preallocate the correct amount of space for the factorization.

Detecting Memory Allocation Problems and Memory Usage

PETSc provides tools to aid in understanding PETSc memory usage and detecting problems with memory allocation, including leaks and use of uninitialized space. Internally, PETSc uses the routines `PetscMalloc()` and `PetscFree()` for memory allocation; instead of directly calling `malloc()` and `free()`. This allows PETSc to track its memory usage and perform error checking. Users are urged to use these routines as well when appropriate.

- The option `-malloc_debug` turns on PETSc's extensive runtime error checking of memory for corruption. This checking can be expensive, so should not be used for production runs. The option `-malloc_test` is equivalent to `-malloc_debug` but only works when PETSc is configured with `--with-debugging` (the default configuration). We suggest setting the environmental variable `PETSC_OPTIONS=-malloc_test` in your shell startup file to automatically enable runtime check memory for developing code but not running optimized code. Using `-malloc_debug` or `-malloc_test` for large runs can slow them significantly, thus we recommend turning them off if you code is painfully slow and you don't need the testing. In addition, you can use

`-check_pointer_intensity 0` for long run debug runs that do not need extensive memory corruption testing. This option is occasionally added to the `PETSC_OPTIONS` environmental variable by some users.

- The option `-malloc_dump` will print a list of memory locations that have not been freed at the conclusion of a program. If all memory has been freed no message is printed. Note that the option `-malloc_dump` activates a call to `PetscMallocDump()` during `PetscFinalize()`. The user can also call `PetscMallocDump()` elsewhere in a program.
- Another useful option is `-malloc_view`, which reports memory usage in all routines at the conclusion of the program. Note that this option 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 can 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).
- The option `-memory_view` provides a high-level view of all memory usage, not just the memory used by `PetscMalloc()`, at the conclusion of the program.
- When running with `-log_view`, the additional option `-log_view_memory` causes the display of additional columns of information about how much memory was allocated and freed during each logged event. This is useful to understand what phases of a computation require the most memory.

One can also use Valgrind to track memory usage and find bugs, see `FAQ: Valgrind usage`.

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 ex5f90 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 `getrusage` system routine. The PETSc option `-malloc_view` reports all memory usage, 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.

4.6 STREAMS: Example Study

Most algorithms in PETSc are memory bandwidth limited. The speed of a simulation depends more on the total achievable[1] memory bandwidth of the computer than the speed (or number) of floating point units. The STREAMS benchmark, a key tool in our field, is invaluable for gaining insights into parallel performance (scaling) by measuring achievable memory bandwidth. PETSc contains multiple implementations of the `triad` STREAMS benchmark: including an OpenMP version and an MPI version.

```
for (int j = 0; j < n; ++j) a[j] = b[j]+scalar*c[j]
```

STREAMS measures the total memory bandwidth achievable when running `n` independent threads or processes on non-overlapping memory regions of an array of total length `N` on a shared memory node. The bandwidth is then computed as `3*n*sizeof(double)/min(time[])`. The timing is done with `MPI_Wtime()`. A call to the timer takes less than 3e-08 seconds, significantly smaller than the benchmark time. The STREAMS benchmark is intentionally embarrassingly parallel, that is, each thread or process works on its own data, completely independently of other threads or processes data. Though real simulations have more complex memory access patterns, most computations for PDEs have large sections of private data and share only data along ghost (halo) regions. Thus the completely independent non-overlapping memory STREAMS model still provides useful information.

As more threads or processes are added, the bandwidth achieved begins to saturate at some `n`, generally less than the number of cores on the node. How quickly the bandwidth saturates, and the speed up (or parallel efficiency) obtained on a given system indicates the likely performance of memory bandwidth-limited computations.

Fig. *STREAMS benchmark gcc* plots the total memory bandwidth achieved and the speedup for runs on an Intel system whose details are provided below. The achieved bandwidth increases rapidly with more cores initially but then less so as more cores are utilized. Also, note that the improvement may, unintuitively, be non-monotone when adding more cores. This is due to the complex interconnect between the cores and their various levels of caches and how the threads or processes are assigned to cores.

There are three important concepts needed to understand memory bandwidth-limited computing.

- Thread or process **binding** to hardware subsets of the shared memory node. The Unix operating system allows threads and processes to migrate among the cores of a node during a computation. This migration is managed by the operating system (OS).[2] A thread or process that is "near" some data may suddenly be far from the data when the thread or process gets migrated. Binding the thread or process to a hardware unit prevents or limits the migration.
- Thread or process **mapping** (assignment) to hardware subsets when more threads or processes are used. Physical memory is divided into multiple distinct units, each of which can independently provide a certain memory bandwidth. Different cores may be more closely connected to different memory units. This results in non-uniform memory access (**NUMA**), meaning the memory latency or bandwidth for any particular core depends on the physical address of the requested memory. When increasing from one thread or process to two, one obviously would like the second thread or process to use a different memory unit and not share the same unit with the first thread or process. Mapping each new thread

[1] Achievable memory bandwidth is the actual bandwidth one can obtain as opposed to the theoretical peak that is calculated using the hardware specification.

[2] Data can also be migrated among different memory sockets during a computation by the OS, but we ignore this possibility in the discussion.

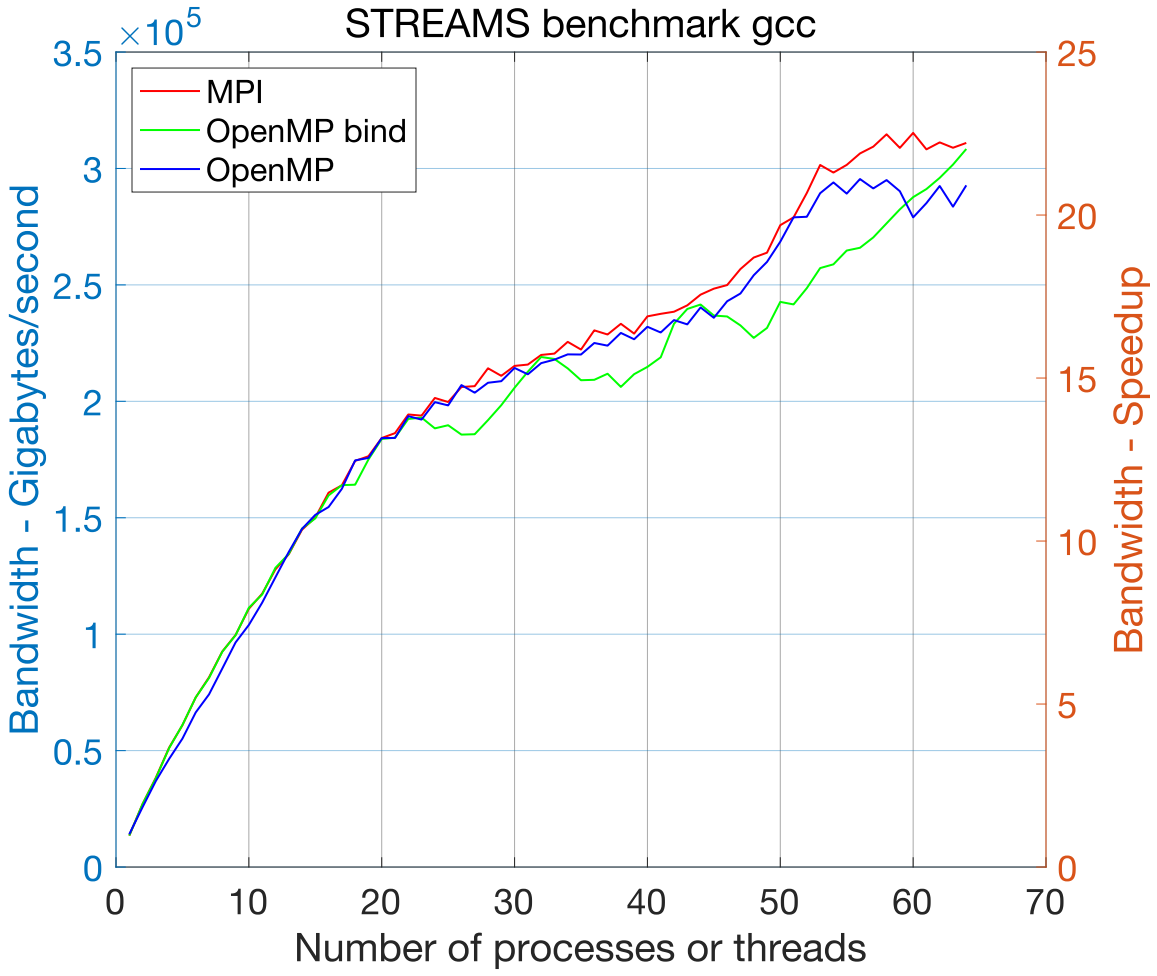


Fig. 4.3: STREAMS benchmark gcc

or process to cores that do not share the previously assigned core's memory unit ensures a higher total achievable bandwidth.

- In addition to mapping, one must ensure that each thread or process **uses data on the closest memory unit**. The OS selects the memory unit to place new pages of virtual memory based on **first touch**: the core of the first thread or process to touch (read or write to) a memory address determines to which memory unit the page of the data is assigned. This is automatic for multiple processes since only one process (on a particular core) will ever touch its data. For threads, care must be taken that the data a thread is to compute on is first touched by that thread. For example, the performance will suffer if the first thread initializes an entire array that multiple threads will later access. For small data arrays that remain in the cache, first touch may produce no performance difference.

MPI and OpenMP provide ways to bind and map processes and cores. They also provide ways to display the current mapping.

- MPI, options to `mpiexec`
  - –bind-to hwthread | core | l1cache | l2cache | l3cache | socket | numa | board
  - –map-by hwthread | core | socket | numa | board | node
  - –report-bindings
  - –cpu-list list of cores
  - –cpu-set list of sets of cores
- OpenMP, environmental variables
  - OMP_NUM_THREADS=n
  - OMP_PROC_BIND=close | spread
  - OMP_PLACES="list of sets of cores" for example {0:2},{2:2},{32:2},{34:2}
  - OMP_DISPLAY_ENV=false | true
  - OMP_DISPLAY_AFFINITY=false | true

Providing appropriate values may be crucial to high performance; the defaults may produce poor results. The best bindings for the STREAMS benchmark are often the best bindings for large PETSc applications. The Linux commands `lscpu` and `numactl -H` provide useful information about the hardware configuration.

It is possible that the MPI initialization (including the use of `mpiexec`) can change the default OpenMP binding/mapping behavior and thus seriously affect the application runtime. The C and Fortran) examples demonstrate this.

We run `ex69f` with four OpenMP threads without `mpiexec` and see almost perfect scaling. The CPU time of the process, which is summed over the four threads in process, is the same as the wall clock time indicating that each thread is run on a different core as desired.

```
$ OMP_NUM_THREADS=4  ./ex69f
  CPU time reported by cpu_time()               6.1660000000000006E-002
  Wall clock time reported by system_clock()    1.8335562000000000E-002
  Wall clock time reported by omp_get_wtime()   1.8330062011955306E-002
```

Running under `mpiexec` gives a very different wall clock time, indicating that all four threads ran on the same core.

```
$ OMP_NUM_THREADS=4 mpiexec -n 1  ./ex69f
  CPU time reported by cpu_time()               7.2290999999999994E-002
  Wall clock time reported by system_clock()    7.2356641999999999E-002
  Wall clock time reported by omp_get_wtime()   7.2353694995399565E-002
```

If we add some binding/mapping options to `mpiexec` we obtain

```
$ OMP_NUM_THREADS=4 mpiexec --bind-to numa -n 1 --map-by core ./ex69f
  CPU time reported by cpu_time()                7.0021000000000000E-002
  Wall clock time reported by system_clock()     1.8489282999999999E-002
  Wall clock time reported by omp_get_wtime()    1.8486462999135256E-002
```

Thus we conclude that this `mpiexec` implementation is, by default, binding the process (including all of its threads) to a single core. Consider also the `mpiexec` option `--map-by socket:pe=$OMP_NUM_THREADS` to ensure each thread gets is own core for computation.

Note that setting `OMP_PROC_BIND=spread` alone does not resolve the problem, as the output below indicates.

```
$ OMP_PROC_BIND=spread OMP_NUM_THREADS=4 mpiexec -n 1  ./ex69f
  CPU time reported by cpu_time()                7.2841999999999990E-002
  Wall clock time reported by system_clock()     7.2946015000000003E-002
  Wall clock time reported by omp_get_wtime()    7.2942997998325154E-002
```

The Fortran routine `cpu_time()` can sometimes produce misleading results when run with multiple threads. Consider again the Fortran example. For an OpenMP parallel loop with enough available cores and the proper binding of threads to cores, one expects the CPU time for the process to be roughly the number of threads times the wall clock time. However, for a loop that is not parallelized (like the second loop in the Fortran example), the CPU time one would expect would match the wall clock time. However, this may not be the case; for example, we have run the Fortran example on an Intel system with the Intel ifort compiler and observed the recorded CPU for the second loop to be roughly the number of threads times the wall clock time even though only a single thread is computing the loop. Thus, comparing the CPU time to the wall clock time of a computation with OpenMP does not give you a good measure of the speedup produced by OpenMP.

4.6.1 Detailed STREAMS study for large arrays

We now present a detailed study of a particular Intel Icelake system, the Intel(R) Xeon(R) Platinum 8362 CPU @ 2.80GH. It has 32 cores on each of two sockets (each with a single NUMA region, so a total of two NUMA regions), a 48 Megabyte L3 cache and 32 1.25 Megabyte L2 caches, each shared by 2 cores. It is running the Rocky Linux 8.8 (Green Obsidian) distribution. The compilers used are GNU 12.2, Intel(R) oneAPI Compiler 2023.0.0 with both icc and icx, and NVIDIA nvhpc/23.1. The MPI implementation is OpenMPI 4.0.7, except for nvhpc, which uses 3.15. The compiler options were

- gcc -O3 -march=native
- icc -O3 -march=native
- icx -O3 -ffinite-math-only (the -xHost option, that replaces -march=native, crashed the compiler so was not used)
- nvc -O3 -march=native

We first run the STREAMS benchmark with large double precision arrays of length 1.6×10^8; the size was selected to be large enough to eliminate cache effects. Fig. *Comprehensive STREAMS performance on Intel system* shows the achieved bandwidth for gcc, icc, icx, and nvc using MPI and OpenMP with their default bindings and with the MPI binding of `--bind-to core --map-by numa` and the OpenMP binding of `spread`.

Note the two dips in the performance with OpenMP and gcc using binding in Fig. *STREAMS benchmark gcc*. Requesting the `spread` binding produces better results for small core counts but poorer ones for larger ones. These are a result of a bug in the gcc `spread` option, placing more threads in one NUMA domain than the other. For example, with gcc, the `OMP_DISPLAY_AFFINITY` shows that for 28 threads, 12 are

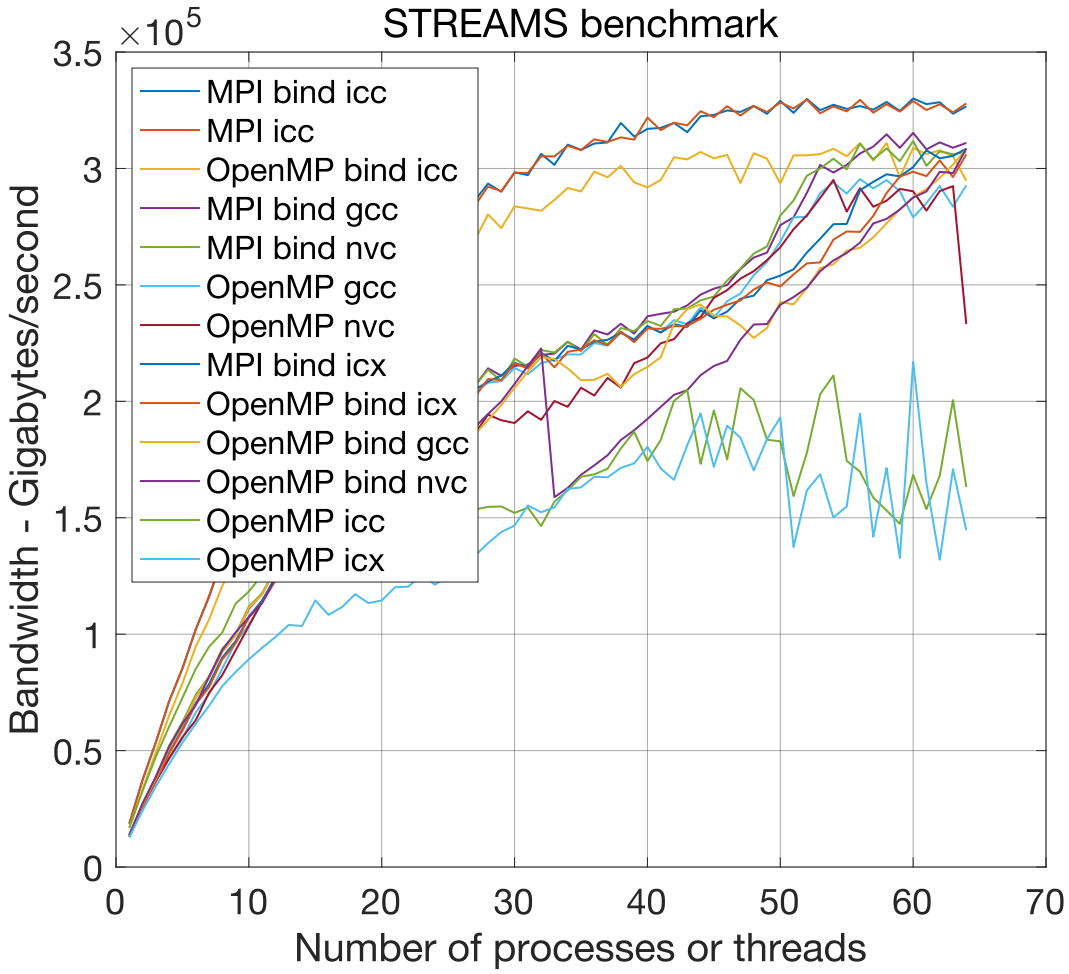


Fig. 4.4: Comprehensive STREAMS performance on Intel system

placed on NUMA region 1, and 16 are placed on the other NUMA region. The other compilers spread the cores evenly.

Fig. *STREAMS benchmark icc* shows the performance with the icc compiler. Note that the icc compiler produces significantly faster code for the benchmark than the other compilers so its STREAMS speedups are smaller, though it provides better performance. No significant dips occur with the OpenMP binding using icc, icx, and nvc; using `OMP_DISPLAY_AFFINITY` confirms, for example, that 14 threads (out of 28) are assigned to each NUMA domain, unlike with gcc. Using the exact thread placement that icc uses with gcc using the OpenMP `OMP_PLACES` option removes most of the dip in the gcc OpenMP binding result. Thus, we conclude that on this system, the `spread` option does not always give the best thread placement with gcc due to its bug.

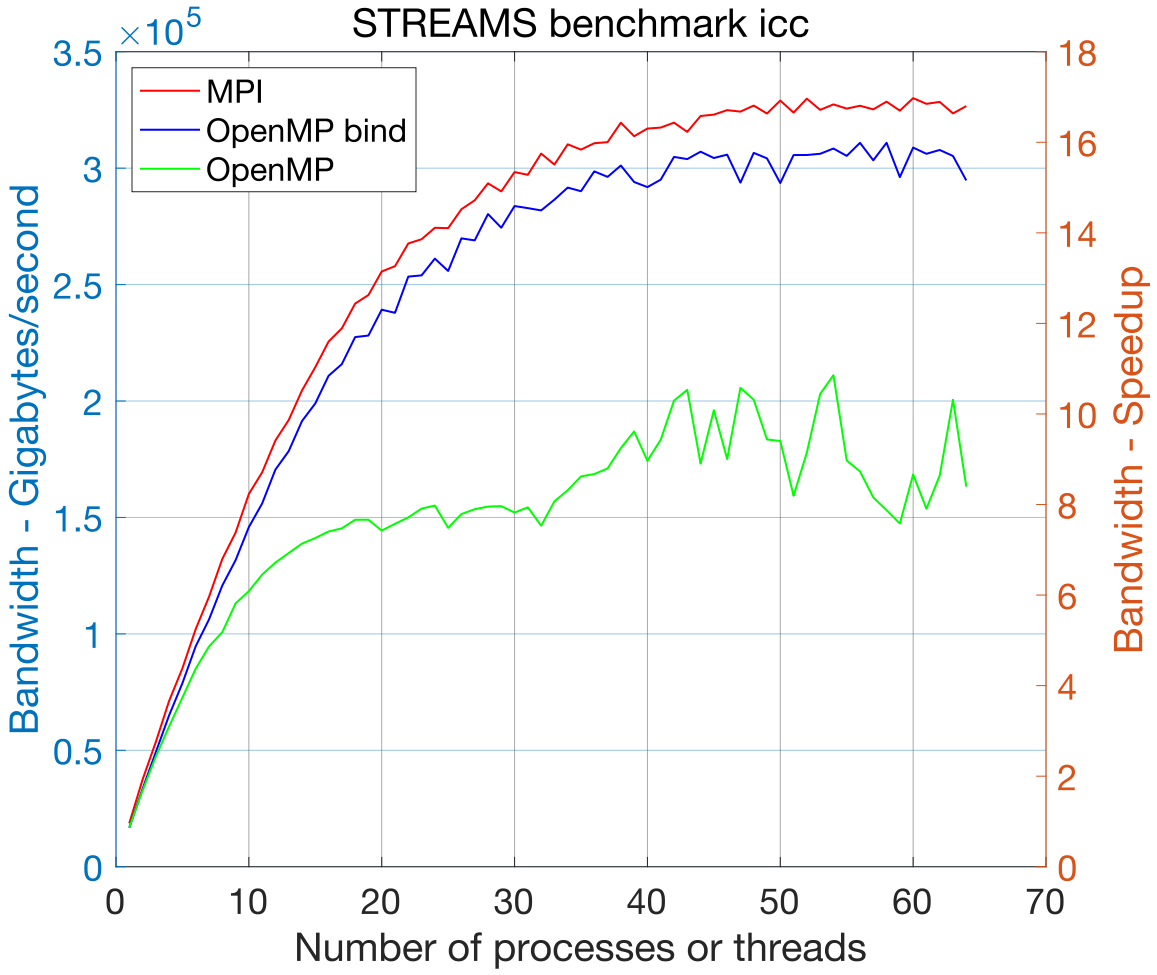


Fig. 4.5: STREAMS benchmark icc

Fig. *STREAMS benchmark icx* shows the performance with the icx compiler.

To understand the disparity in the STREAMS performance with icc we reran it with the highest optimization level that produced the same results as gcc and icx: `-O1` without `-march=native`. The results are displayed in Fig. *STREAMS benchmark icc -O1*; sure enough, the results now match that of gcc and icx.

Next we display the STREAMS results using gcc with parallel efficiency instead of speedup in *STREAMS parallel efficiency gcc*

Observations:

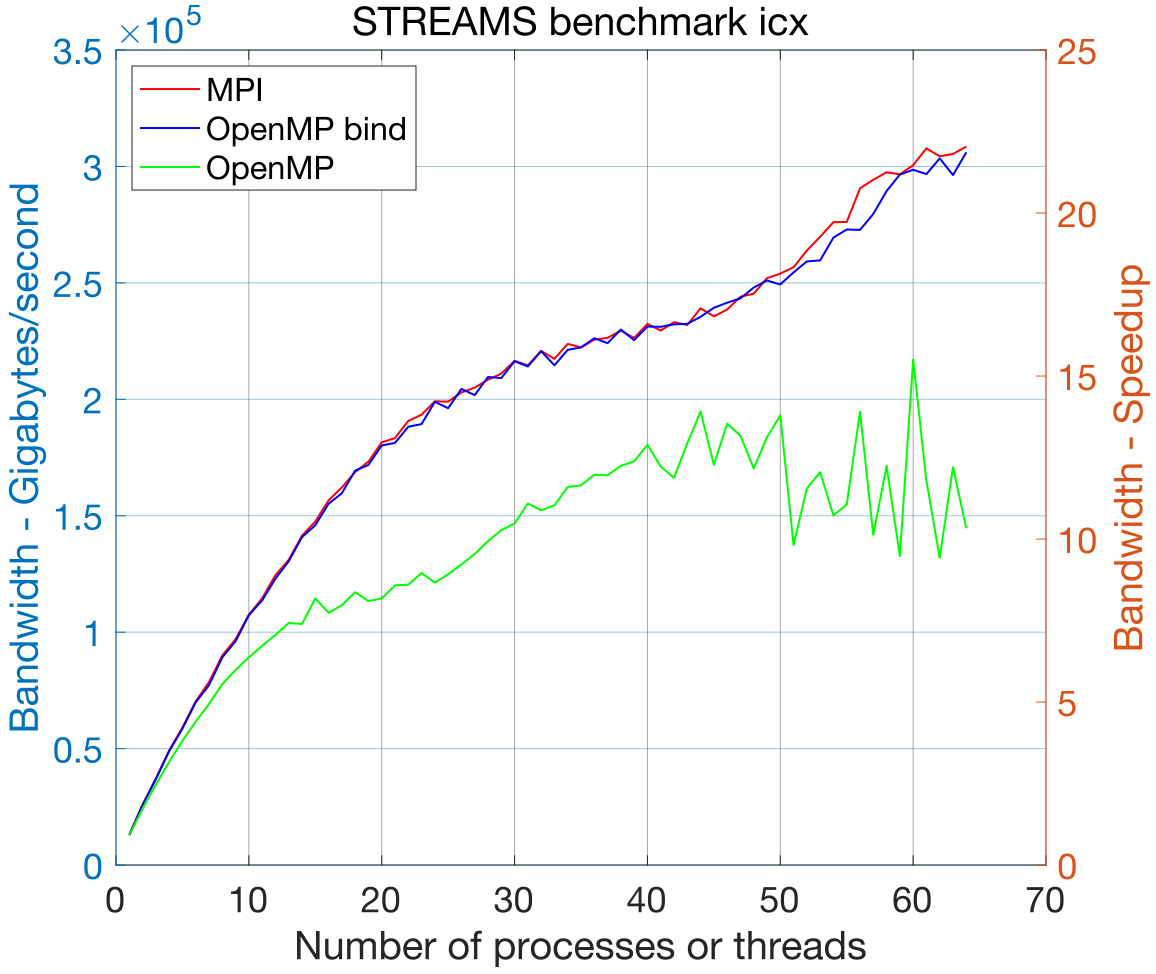


Fig. 4.6: STREAMS benchmark icx

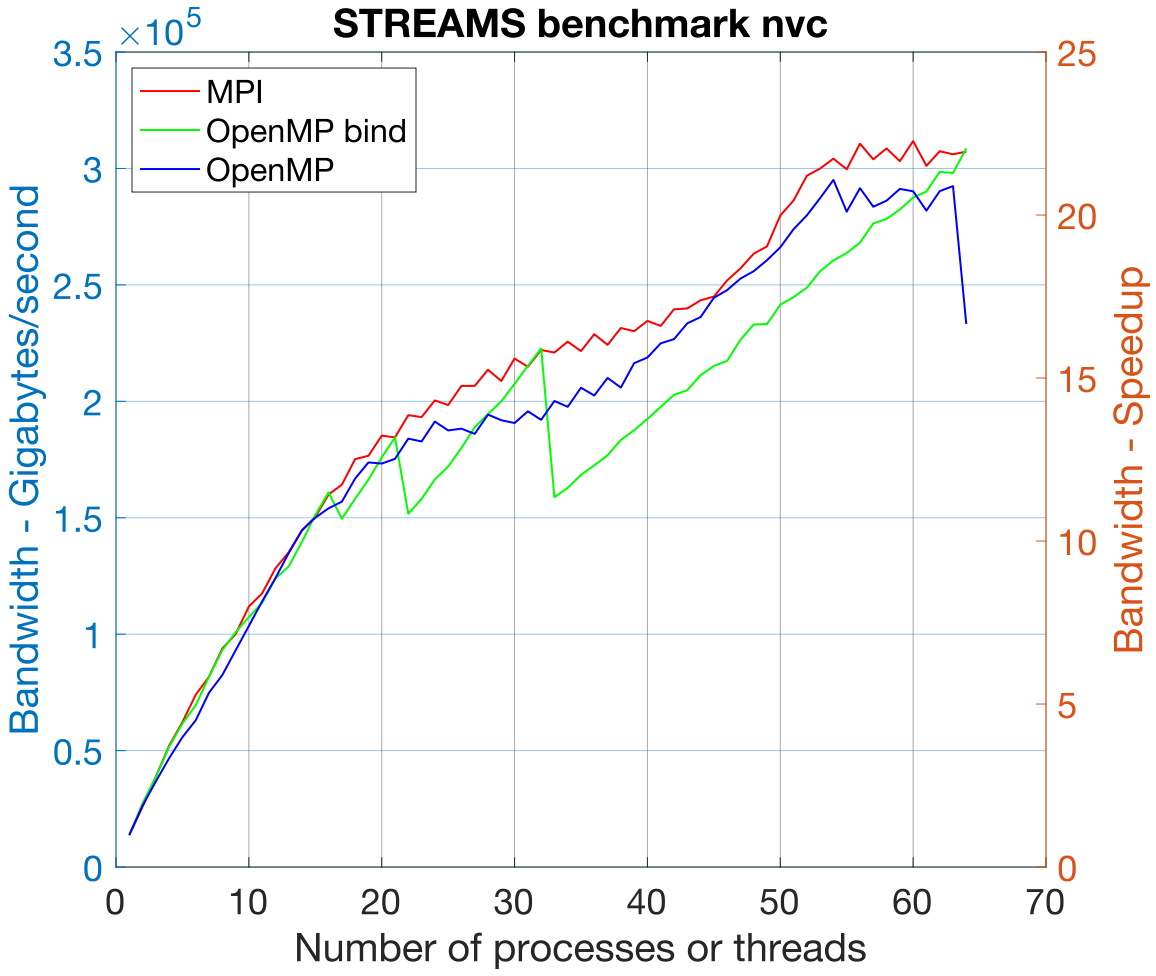


Fig. 4.7: STREAMS benchmark nvc

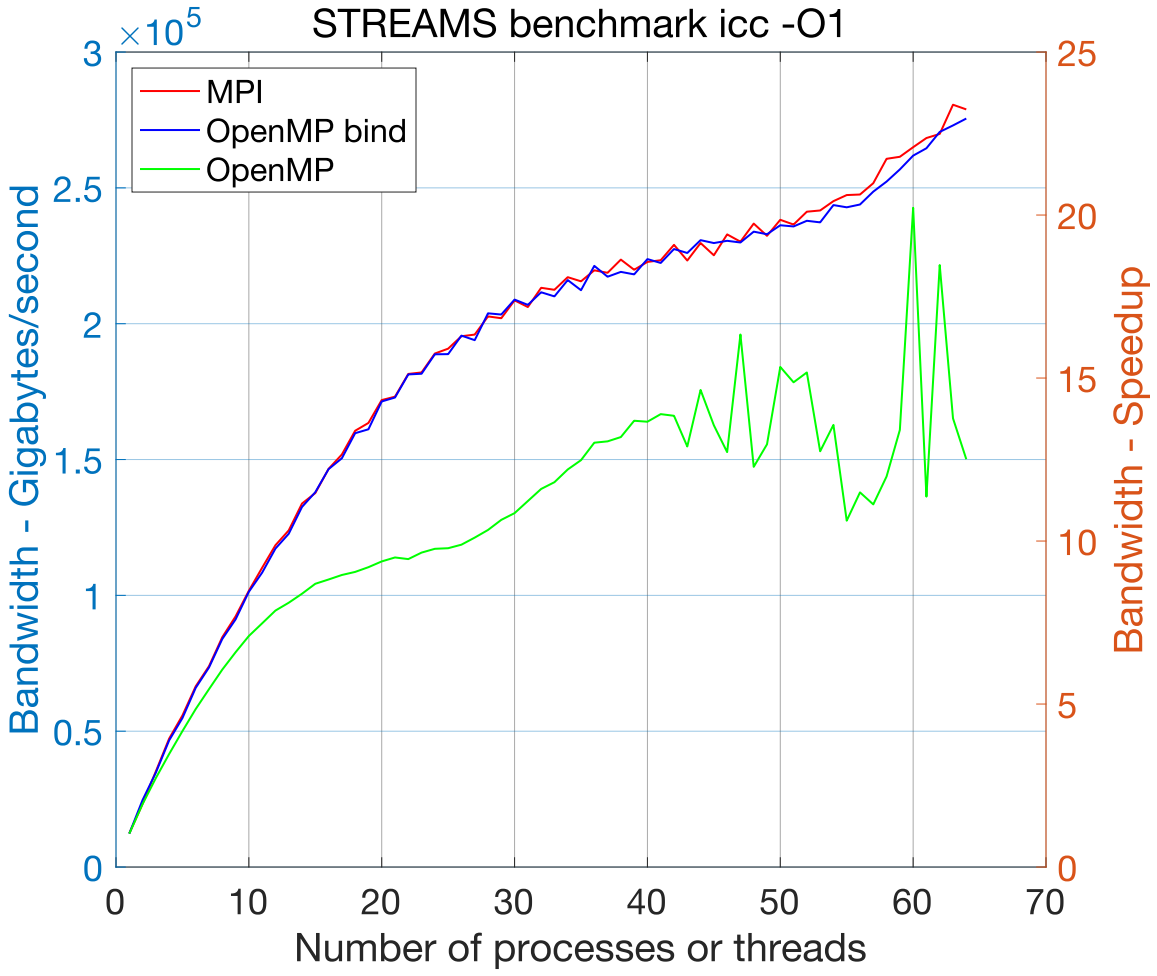


Fig. 4.8: STREAMS benchmark icc -O1

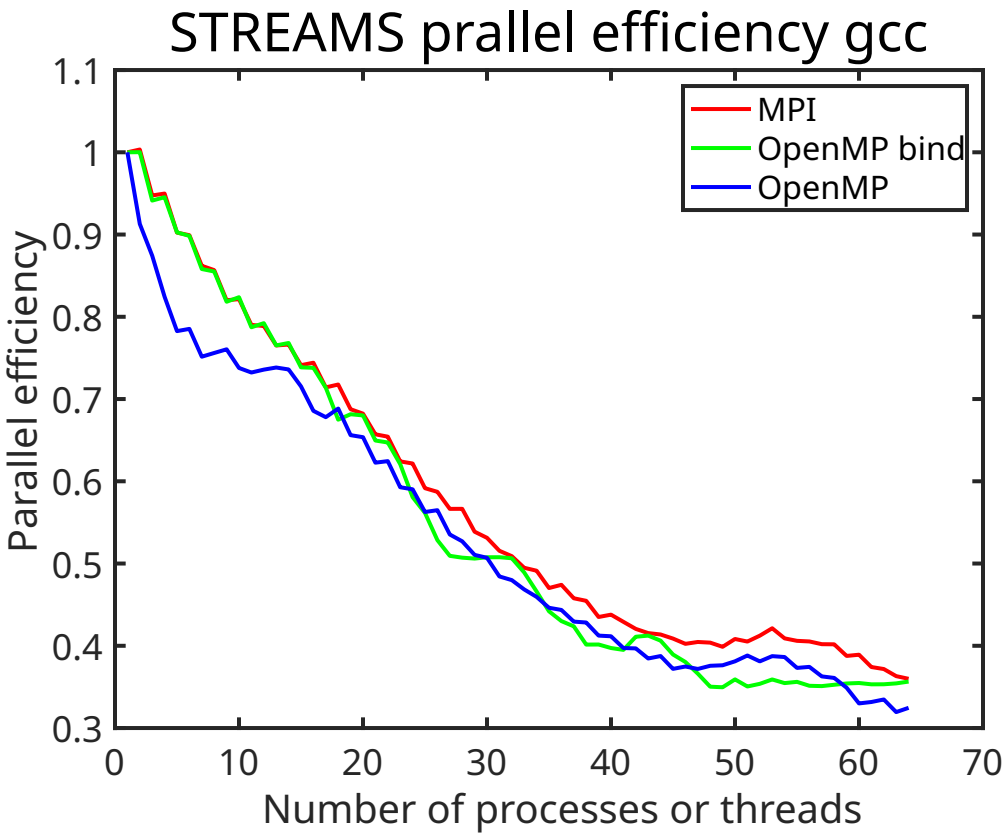


Fig. 4.9: STREAMS parallel efficiency gcc

- For MPI, the default binding and mapping on this system produces results that are as good as providing a specific binding and mapping. This is not true on many systems!
- For OpenMP gcc, the default binding is better than using `spread`, because `spread` has a bug. For the other compilers using `spread` is crucial for good performance on more than 32 cores.
- We do not have any explanation why the improvement in speedup for gcc, icx, and nvc slows down between 32 and 48 cores and then improves rapidly since we believe appropriate bindings are being used.

We now present a limited version of the analysis above on an Apple MacBook Pro M2 Max using MPICH, version 4.1, gcc version 13.2 (installed via Homebrew), XCode 15.0.1 and -O3 optimization flags with a smaller N of 80,000,000. macOS contains no public API for setting or controlling affinities so it is not possible to set bindings for either MPI or OpenMP. In addition, the M2 has a combination of performance and efficiency cores which we have no control over the use of.

Fig. *STREAMS benchmark on Apple M2* provides the results. Based on the plateau in the middle of the plot, we assume that the core numbering that is used by MPICH does not produce the best binding.

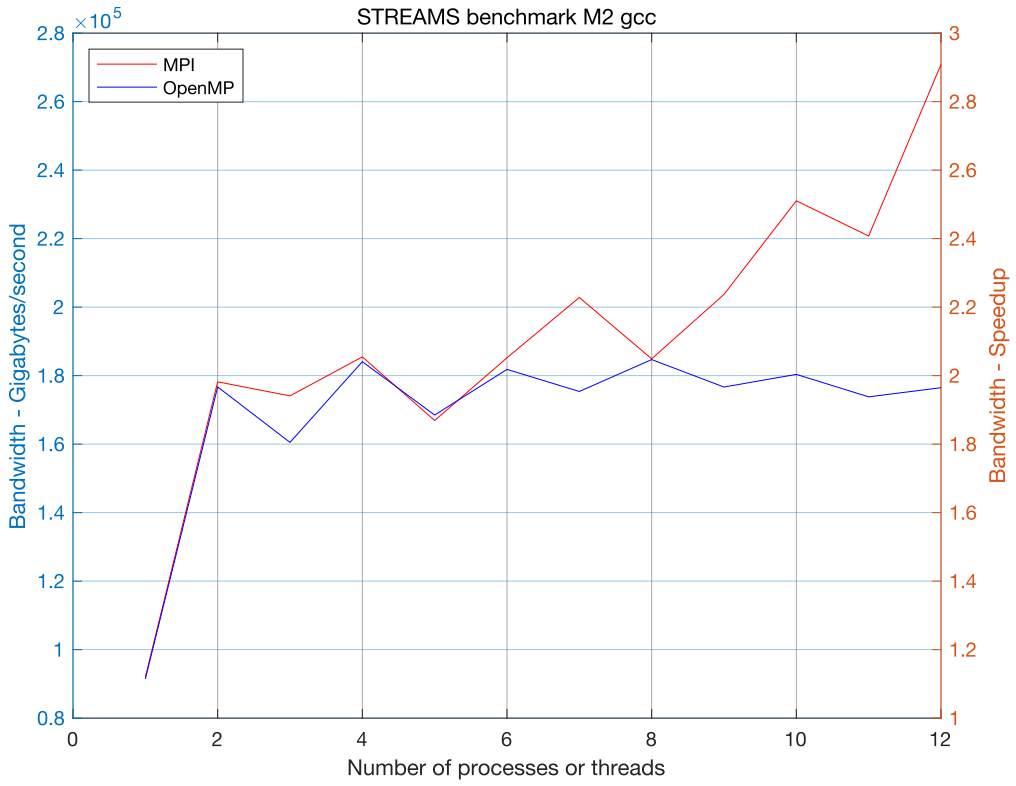


Fig. 4.10: STREAMS benchmark on Apple M2
OpenMPI (installed via Homebrew) produced similar results.

4.6.2 Detailed study with application

We now move on to a PETSc application which solves a three-dimensional Poisson problem on a unit cube discretized with finite differences whose linear system is solved with the PETSc algebraic multigrid preconditioner, `PCGAMG` and Krylov accelerator GMRES. Strong scaling is used to compare with the STREAMS benchmark: measuring the time to construct the preconditioner, the time to solve the linear system with the preconditioner, and the time for the matrix-vector products. These are displayed in Fig. *GAMG speedup*. The runtime options were `-da_refine 6 -pc_type gamg -log_view`. This study did not attempt to tune the default `PCGAMG` parameters. There were very similar speedups for all the compilers so we only display results for gcc.

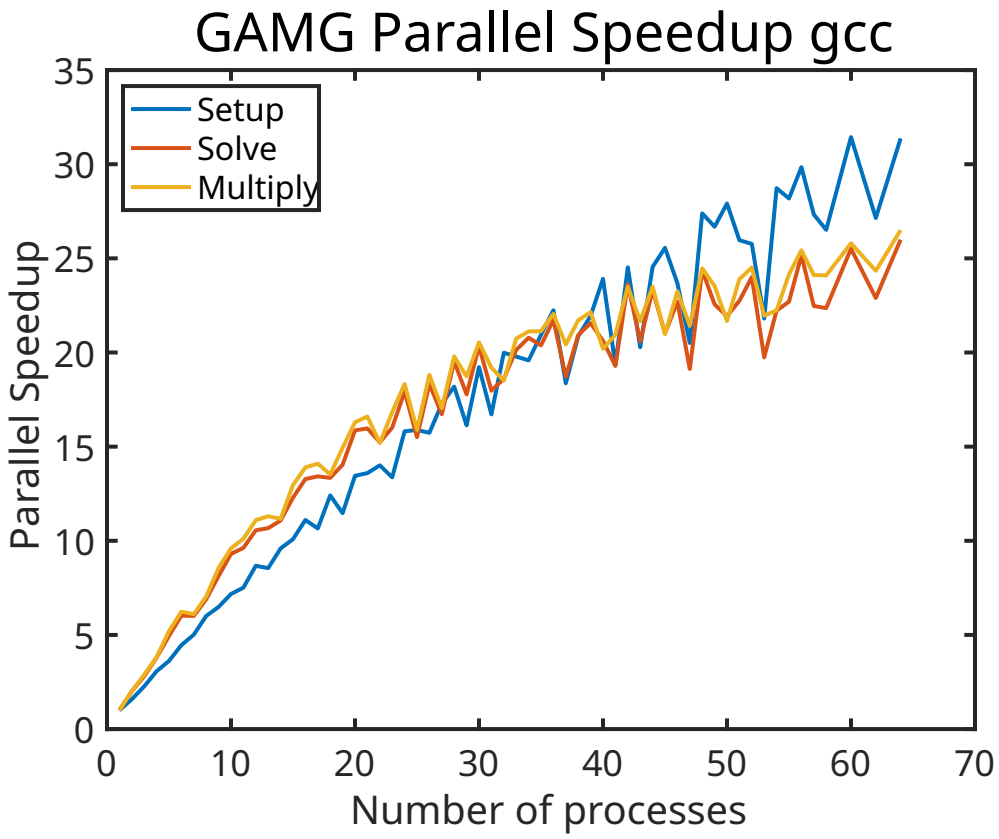


Fig. 4.11: GAMG speedup

The dips in the performance at certain core counts are consistent between compilers and results from the amount of MPI communication required from the communication pattern which results from the different three-dimensional parallel grid layout.

We now present GAMG on the Apple MacBook Pro M2 Max. Fig. *GAMG speedup Apple M2* provides the results. The performance is better than predicted by the STREAMS benchmark for all portions of the solver.

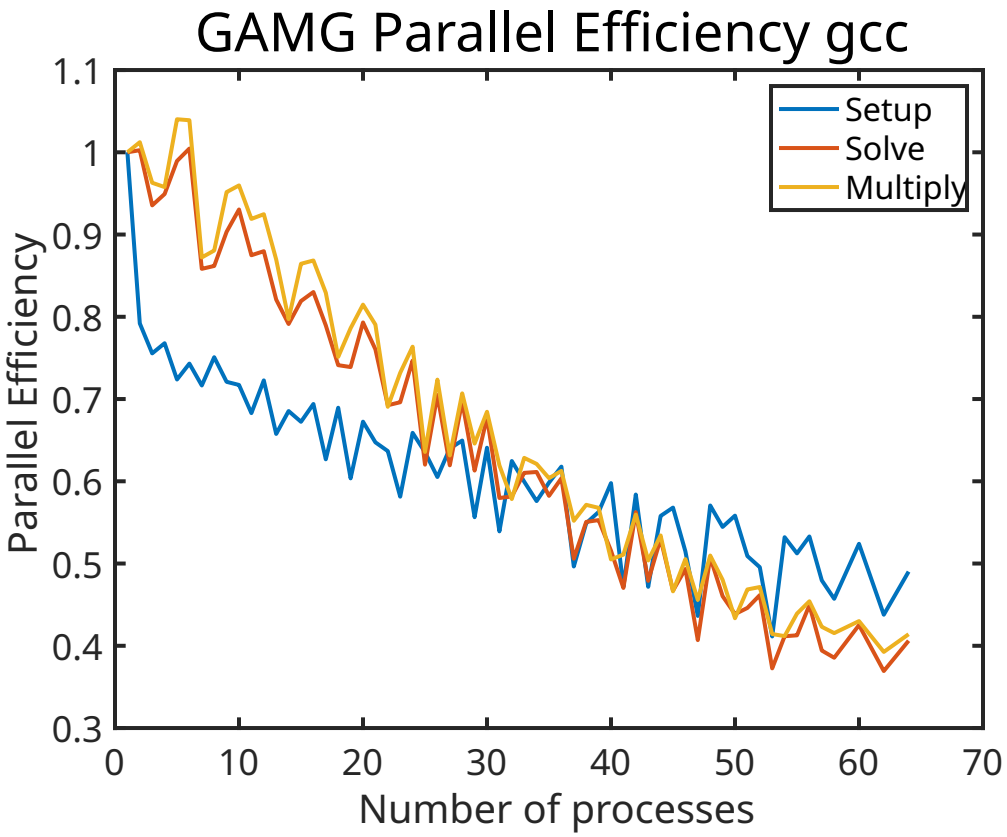


Fig. 4.12: GAMG parallel efficiency

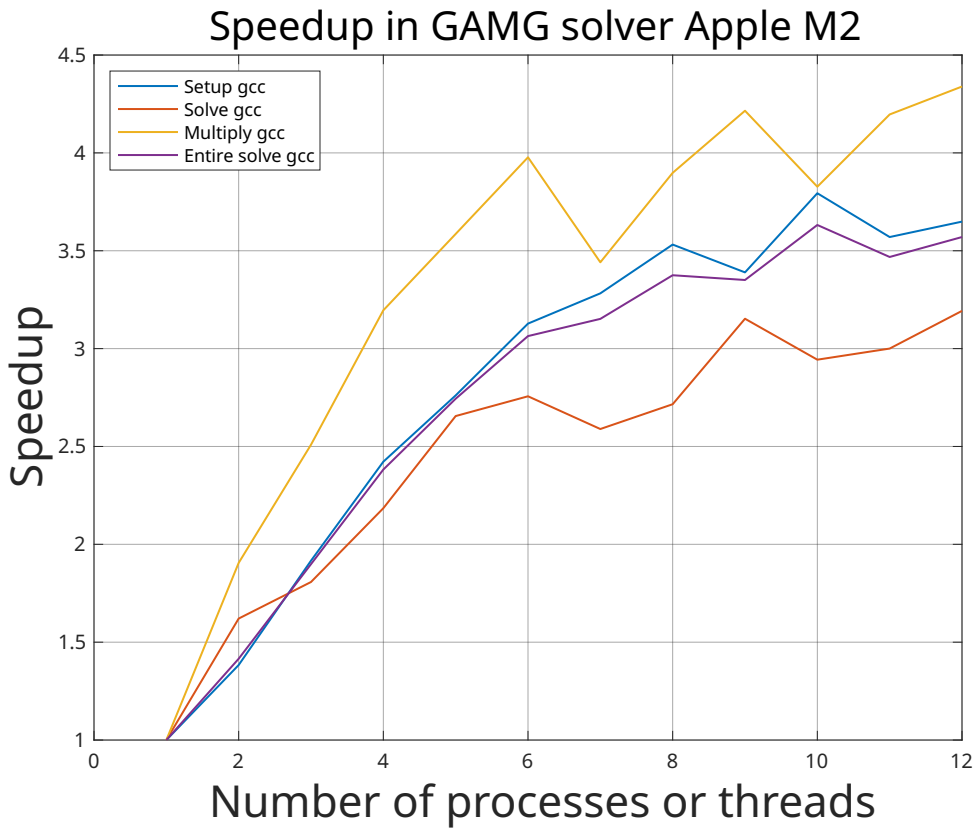


Fig. 4.13: GAMG speedup Apple M2

4.6.3 Application with the MPI linear solver server

We now run the same PETSc application using the MPI linear solver server mode, set using `-mpi_linear_solver_server`. All compilers deliver largely the same performance so we only present results with gcc. We plot the speedup in Fig. *GAMG server speedup* and parallel efficiency in *GAMG server parallel efficiency* Note that it is far below the parallel solve without the server. However, the distribution time for these runs was always less than three percent of the complete solution time. The reason for the poorer performance is because in the pure MPI version, the vectors are partitioned directly from the three-dimensional grid; the cube is divided into (approximate) sub-cubes, this minimizes the inter-process communication, especially in the matrix-vector product. In server mode, the vector is laid out using the cube's natural ordering, and then each MPI process is assigned a contiguous subset of the vector. As a result, the flop rate for the matrix-vector product is significantly higher than that of the pure MPI version. This indicates that a naive use of the MPI linear solver server will not produce as much performance as a usage that considers the matrix/vector layouts by performing an initial grid partitioning. For example, if OpenMP is used to generate the matrix, it would be appropriate to have each OpenMP thread assigned a contiguous vector mapping to a sub-cube of the domain. This would require, of course, a far more complicated OpenMP code that is written using MPI-like parallelism and decomposition of the data.

`PCMPI` has two approaches for distributing the linear system. The first uses `MPI_Scatterv()` to communicate the matrix and vector entries from the initial compute process to all of the server processes. Unfortunately, `MPI_Scatterv()` does not scale with more MPI processes; hence, the solution time is limited by the `MPI_Scatterv()`. To remove this limitation, the second communication mechanism is Unix shared memory `shmget()`. Here, `PCMPI` allocates shared memory from which all the MPI processes in the server can access their portion of the matrices and vectors that they need. There is still a (now much smaller) server processing overhead since the initial data storage of the sequential matrix (in `MATSEQAIJ` storage) still must be converted to `MATMPIAIJ` storage. `VecPlaceArray()` is used to convert the sequential vector to an MPI vector, so there is no overhead, not even a copy, for this operation.

In *GAMG server parallel efficiency vs STREAMS*, we plot the parallel efficiency of the linear solve and the STREAMS benchmark, which track each other well. This example demonstrates the **utility of the STREAMS benchmark to predict the speedup (parallel efficiency) of a memory bandwidth limited application** on a shared memory Linux system.

For the Apple M2, we present the results using Unix shared-memory communication of the matrix and vectors to the server processes in *GAMG server solver speedup on Apple M2*. To run this one must first set up the machine to use shared memory as described in `PetscShmgetAllocateArray()`

This example demonstrates that the **MPI linear solver server feature of PETSc can generate a reasonable speedup in the linear solver** on machines that have significant memory bandwidth. However, one should not expect the speedup to be near the total number of cores on the compute node.

4.7 The Use of BLAS and LAPACK in PETSc and external libraries

1. BLAS 1 operations (and GPU equivalents) - vector operations such as `VecNorm()`, `VecAXPY()`, and `VecScale()` are used extensively in PETSc. 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.

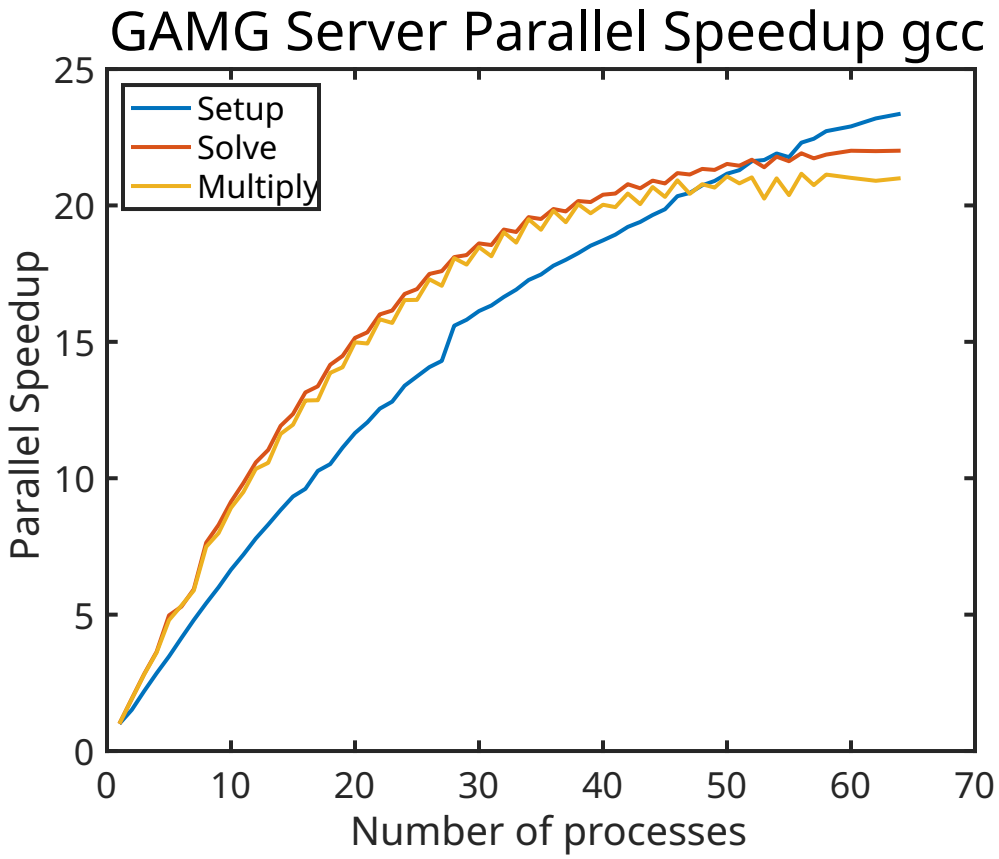


Fig. 4.14: GAMG server speedup

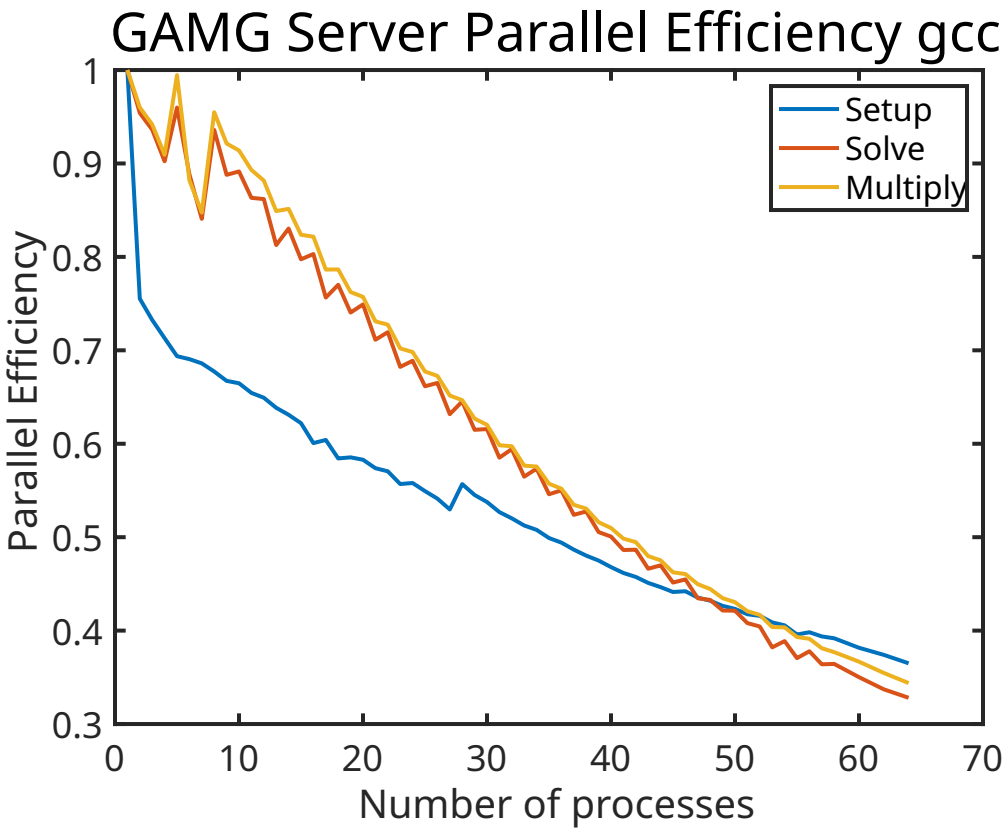


Fig. 4.15: GAMG server parallel efficiency

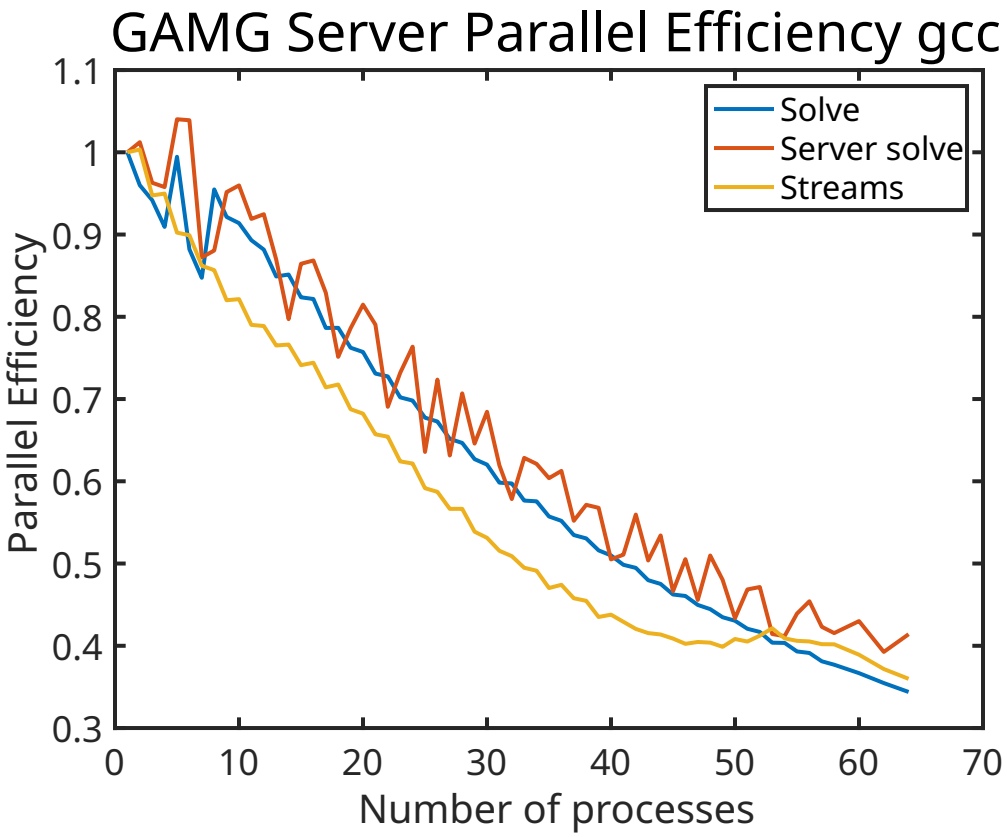


Fig. 4.16: GAMG server parallel efficiency vs STREAMS

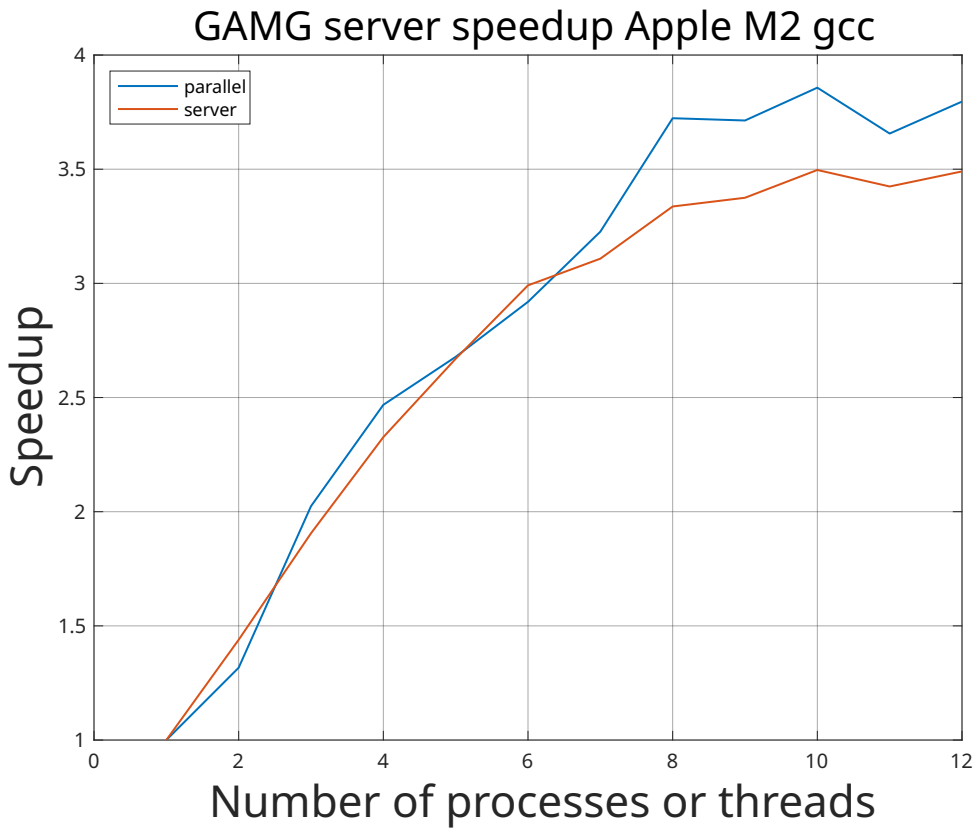


Fig. 4.17: GAMG server solver speedup on Apple M2

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.

4.7.1 32 or 64-bit BLAS/LAPACK integers

BLAS/LAPACK libraries may use 32 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 when 64-bit PETSc integers are used.

The configure option `--with-64-bit-blas-indices` attempts to locate and use a 64-bit integer version of BLAS/LAPACK library. Except for MKL Cluster PARDISO, most external packages do not support using 64-bit BLAS/LAPACK integers so if you are using such packages you cannot use 64-bit BLAS/LAPACK integers.

The configure options `--with-64-bit-indices` and `--with-64-bit-blas-indices` are independent. `--with-64-bit-indices` does not imply that the BLAS/LAPACK libraries use 64 bit indices.

4.7.2 Shared memory BLAS/LAPACK parallelism

Some BLAS/LAPACK libraries can make use of shared memory parallelism within the function calls, generally using OpenMP, or possibly PThreads. If this feature is turned 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 in 64 threads competing to use 16 cores which will perform poorly.

If one elects to use both MPI parallelism and shared memory BLAS/LAPACK parallelism one should ensure they do not over subscribe the hardware resources. Since PETSc does not natively use 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 each MPI process for its shared memory parallel BLAS/LAPACK. The additional environmental variables `OMP_PROC_BIND` and `OMP_PLACES` may also need to be set appropriately 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.

4.7.3 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 less optimized 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`. This is recommended as a portable high-performance option. It is possible if you use `--download-blis` without `--download-f2cblaslapack` the BLIS library installed will **not** be used! Instead, PETSc will link in some LAPACK implementation and the BLAS that comes with that implementation!

4.8 Other PETSc Features

4.8.1 PETSc on a process subset

Users who wish to employ PETSc on only a subset of MPI 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 use an existing MPI communicator `comm`,

```
PETSC_COMM_WORLD = comm; /* To use a previously-defined MPI_Comm */
```

before calling `PetscInitialize()`, but, obviously, after calling `MPI_Init()`.

4.8.2 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 solver BiCGStab by calling

```
KSPSetType(ksp, KSPBCGS);
```

One could then override this choice at runtime with the option

```
-ksp_type tfqmr
```

to select the Transpose-Free QMR algorithm. (See *KSP: Linear System Solvers* for details.)

The remainder of this section discusses details of runtime options.

The Options Database

Each PETSc process maintains a database of option names and values (stored as text strings). This database is generated with the command `PetscInitialize()`, which is listed below in its C/C++ and Fortran variants, respectively:

```
PetscInitialize(int *argc, char ***args, const char *file, const char *help); // C
```

```
call PetscInitialize(integer ierr) ! Fortran
```

The arguments `argc` and `args` (in the C/C++ version only) are the addresses of the usual command line arguments, while the `file` is a name of an optional 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. For uniformity throughout PETSc, we employ the format `-[prefix_]package_option` (for instance, `-ksp_type`, `-mat_view ::info`, or `-mg_levels_ksp_type`).

Users can specify an alias for any option name (to avoid typing the sometimes lengthy default name) by adding an alias to the `.petscrc` file in the format

```
alias -newname -oldname
```

For example,

```
alias -kspt -ksp_type
alias -sd -start_in_debugger
```

Comments can be placed in the `.petscrc` file by using `#` in the first column of a line.

Options Prefixes

Options prefixes allow specific objects to be controlled from the options database. For instance, `PCMG` gives prefixes to its nested `KSP` objects; one may control the coarse grid solver by adding the `mg_coarse` prefix, for example `-mg_coarse_ksp_type preonly`. One may also use `KSPSetOptionsPrefix()`,`DMSetOptionsPrefix()` , `SNESSetOptionsPrefix()`, `TSSetOptionsPrefix()`, and similar functions to assign custom prefixes, useful for applications with multiple or nested solvers.

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 with `PetscOptionsInsertFileYAML()`. For example, the following file:

```
$$: ignored
$$tail: ignored

$$ans: &ans 42
$$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

$$1: &seq-bool [true, false]
$$2: &seq-int  [123, 456, 789]
$$3: &seq-real [*pi, *eu]
$$4: &seq-str  [abc, ijk, fgh]

seq1: {
  bool: *seq-bool,
  int:  *seq-int,
  real: *seq-real,
  str:  *seq-str,
}

seq2:
  bool:
    - true
    - false
  int:
    - 123
    - 456
    - 789
  real:
    - *pi
    - *eu
  str:
    - rst
    - uvw
    - xyz
```

(continues on next page)

(continued from previous page)

```
map:
  - key0: 0
  - key1: 1
  - key2: 2
  - $$: ignored
  - $$tail: ignored
```

corresponds to the following PETSc options:

```
-map key0,key1,key2 # (source: file)
-map_key0 0 # (source: file)
-map_key1 1 # (source: file)
-map_key2 2 # (source: file)
-opt_bool true # (source: file)
-opt_cmplx -3.14+2.72i # (source: file)
-opt_imag 2.72i # (source: file)
-opt_int 42 # (source: file)
-opt_real 3.14 # (source: file)
-opt_str petsc # (source: file)
-seq1_bool true,false # (source: file)
-seq1_int 123,456,789 # (source: file)
-seq1_real 3.14,2.72 # (source: file)
-seq1_str abc,ijk,fgh # (source: file)
-seq2_bool true,false # (source: file)
-seq2_int 123,456,789 # (source: file)
-seq2_real 3.14,2.72 # (source: file)
-seq2_str rst,uvw,xyz # (source: file)
```

With `-options_file`, PETSc will parse the file as YAML if it ends in a standard YAML or JSON[1] 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 1.2
---
name: value
```

and

```
---
name: value
```

both correspond to options

[1] JSON is a subset of YAML

```
-name value # (source: file)
```

User-Defined PetscOptions

Any subroutine in a PETSc program can add entries to the database with the command

```
PetscOptionsSetValue(PetscOptions options, char *name, char *value);
```

though this is rarely done. To locate options in the database, one should use the commands

```
PetscOptionsHasName(PetscOptions options, char *pre, char *name, PetscBool *flg);
PetscOptionsGetInt(PetscOptions options, char *pre, char *name, PetscInt *value,␣
↪PetscBool *flg);
PetscOptionsGetReal(PetscOptions options, char *pre, char *name, PetscReal *value,␣
↪PetscBool *flg);
PetscOptionsGetString(PetscOptions options, char *pre, char *name, char *value, size_
↪t maxlen, PetscBool  *flg);
PetscOptionsGetStringArray(PetscOptions options, char *pre, char *name, char **values,
↪ PetscInt *nmax, PetscBool *flg);
PetscOptionsGetIntArray(PetscOptions options, char *pre, char *name, PetscInt *value,␣
↪PetscInt *nmax, PetscBool *flg);
PetscOptionsGetRealArray(PetscOptions options, char *pre, char *name, PetscReal␣
↪*value, PetscInt *nmax, PetscBool *flg);
```

All of these routines set `flg=PETSC_TRUE` if the corresponding option was found, `flg=PETSC_FALSE` if it was not found. The optional argument `pre` indicates that the true name of the option is the given name (with the dash "-" removed) prepended by the prefix `pre`. Usually `pre` should be set to `NULL` (or `PETSC_NULL_CHARACTER` for Fortran); its purpose is to allow someone to rename all the options in a package without knowing the names of the individual options. For example, when using block Jacobi preconditioning, the `KSP` and `PC` methods used on the individual blocks can be controlled via the options `-sub_ksp_type` and `-sub_pc_type`.

Keeping Track of Options

One useful means of keeping track of user-specified runtime options is use of `-options_view`, which prints to `stdout` during `PetscFinalize()` a table of all runtime options that the user has specified. A related option is `-options_left`, which prints the options table and indicates any options that have *not* been requested upon a call to `PetscFinalize()`. This feature is useful to check whether an option has been activated for a particular PETSc object (such as a solver or matrix format), or whether an option name may have been accidentally misspelled.

The option `-options_monitor <viewer>` turns on the default monitoring of options. `PetscOptionsMonitorSet()` can be used to provide custom monitors. The option `-options_monitor_cancel` prevents any monitoring by monitors set with `PetscOptionsMonitorSet()` (but not that set with `-options_monitor`).

4.8.3 Viewers: Looking at PETSc Objects

PETSc employs a consistent scheme for examining, printing, and saving objects through commands of the form

```
XXXView(XXX obj, PetscViewer viewer);
```

Here `obj` is a PETSc object of type `XXX`, where `XXX` is `Mat`, `Vec`, `SNES`, etc. There are several predefined viewers.

- Passing in a zero (`0`) for the viewer causes the object to be printed to the screen; this is useful when viewing an object in a debugger but should be avoided in source code.
- `PETSC_VIEWER_STDOUT_SELF` and `PETSC_VIEWER_STDOUT_WORLD` causes the object to be printed to the screen.
- `PETSC_VIEWER_DRAW_SELF PETSC_VIEWER_DRAW_WORLD` causes the object to be drawn in a default X window.
- Passing in a viewer obtained by `PetscViewerDrawOpen()` causes the object to be displayed graphically. See *Graphics* for more on PETSc's graphics support.
- To save an object to a file in ASCII format, the user creates the viewer object with the command `PetscViewerASCIIOpen(MPI_Comm comm, char* file, PetscViewer *viewer)`. This object is analogous to `PETSC_VIEWER_STDOUT_SELF` (for a communicator of `MPI_COMM_SELF`) and `PETSC_VIEWER_STDOUT_WORLD` (for a parallel communicator).
- To save an object to a file in binary format, the user creates the viewer object with the command `PetscViewerBinaryOpen(MPI_Comm comm, char* file, PetscViewerBinaryType type, PetscViewer *viewer)`. Details of binary I/O are discussed below.
- Vector and matrix objects can be passed to a running MATLAB process with a viewer created by `PetscViewerSocketOpen(MPI_Comm comm, char *machine, int port, PetscViewer *viewer)`. 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(Mat newmat, PetscViewer viewer);
VecLoad(Vec newvec, PetscViewer viewer);
```

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. The info file is automatically created when `VecView()` or `MatView()` is used with a binary viewer; hence if you save a matrix with a given block size with `MatView()`, then a `MatLoad()` on that file will automatically use the saved block size.

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 `PetscOptionsCreateViewer()` 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`.

4.8.4 Using SAWs with PETSc

The Scientific Application Web server, SAWs[2], allows one to monitor running PETSc applications from a browser. To use SAWs you must `configure` PETSc with the option `--download-saws`. Options to use SAWs include

- `-saws_options` - allows setting values in the PETSc options database via the browser (works only on one process).
- `-stack_view saws` - allows monitoring the current stack frame that PETSc is in; refresh to see the new location.
- `-snes_monitor_saws, -ksp_monitor_saws` - monitor the solvers' iterations from the web browser.

For each of these you need to point your browser to `http://hostname:8080`, for example `http://localhost:8080`. Options that control behavior of SAWs include

- `-saws_log filename` - log all SAWs actions in a file.
- `-saws_https certfile` - use HTTPS instead of HTTP with a certificate.
- `-saws_port_auto_select` - have SAWs pick a port number instead of using 8080.
- `-saws_port port` - use `port` instead of 8080.
- `-saws_root rootdirectory` - local directory to which the SAWs browser will have read access.
- `-saws_local` - use the local file system to obtain the SAWS javascript files (they much be in `rootdirectory/js`).

Also see the manual pages for `PetscSAWsBlock()`, `PetscObjectSAWsTakeAccess()`, `PetscObjectSAWsGrantAccess()`, `PetscObjectSAWsSetBlock()`, `PetscStackSAWsGrantAccess()` `PetscStackSAWsTakeAccess()`, `KSPMonitorSAWs()`, and `SNESMonitorSAWs()`.

4.8.5 Debugging

PETSc programs may be debugged using one of the two options below.

- `-start_in_debugger [(noxterm)],[(gdb|lldb|...)] [-display name]` - start all processes in debugger
- `-on_error_attach_debugger [(noxterm)],[(gdb|lldb|...)] [-display name]` - start debugger only on encountering an error

Note that, in general, debugging MPI programs cannot be done in the usual manner of starting the programming in the debugger (because then it cannot set up the MPI communication and remote processes).

By default on Linux systems the GNU debugger `gdb` is used, on macOS systems `lldb` is used

By default, the debugger will be started in a new xterm (Apple Terminal on macOS), to enable running separate debuggers on each process, unless the option `noxterm` is used. In order to handle the MPI startup phase, the debugger command `cont` should be used to continue execution of the program within the debugger. Rerunning the program through the debugger requires terminating the first job and restarting the processor(s); the usual `run` option in the debugger will not correctly handle the MPI startup and should not be used. Not all debuggers work on all machines, the user may have to experiment to find one that works correctly.

You can select a subset of the processes to be debugged (the rest just run without the debugger) with the option

[2] Saws wiki on Bitbucket

```
-debugger_ranks rank1,rank2,...
```

where you simply list the ranks you want the debugger to run with.

4.8.6 Error Handling

Errors are handled through the routine `PetscError()`. This routine checks a stack of error handlers and calls the one on the top. If the stack is empty, it selects `PetscTraceBackErrorHandler()`, which tries to print a traceback. A new error handler can be put on the stack with

```
PetscPushErrorHandler(PetscErrorCode (*HandlerFunction)(int line, char *dir, char␣
↪*file, char *message, int number, void*), void *HandlerContext)
```

The arguments to `HandlerFunction()` are the line number where the error occurred, the file in which the error was detected, the corresponding directory, the error message, the error integer, and the `HandlerContext`. The routine

```
PetscPopErrorHandler()
```

removes the last error handler and discards it.

PETSc provides two additional error handlers besides `PetscTraceBackErrorHandler()`:

```
PetscAbortErrorHandler()
PetscAttachDebuggerErrorHandler()
```

The function `PetscAbortErrorHandler()` calls abort on encountering an error, while `PetscAttachDebuggerErrorHandler()` 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,][(gdb|lldb)] [-display DISPLAY]`.

All PETSc calls can be traced (useful for determining where a program is hanging without running in the debugger) with the option

```
-log_trace [filename]
```

where `filename` is optional. By default the traces are printed to the screen. This can also be set with the command `PetscLogTraceBegin(FILE*)`.

It is also possible to trap signals by using the command

```
PetscPushSignalHandler(PetscErrorCode (*Handler)(int, PetscCtx), PetscCtx 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 turn on the default PETSc signal handler in `PetscInitialize()`, this can be prevented with the option `-no_signal_handler` that can be provided on the command line, in the ~./petscrc file, or with the call

```
PetscCall(PetscOptionsSetValue(NULL, "-no_signal_handler", "true"));
```

Once the first PETSc signal handler has been pushed it is impossible to go back to to a signal handler that was set directly by the user with the UNIX signal handler API or by the loader.

Some Fortran compilers/loaders cause, by default, a traceback of the Fortran call stack when a segmentation violation occurs to be printed. This is handled by them setting a special signal handler when the program is

started up. This feature is useful for debugging without needing to start up a debugger. If `PetscPushSignalHandler()` has been called this traceback will not occur, hence if the Fortran traceback is desired one should put

```
PetscCallA(PetscOptionsSetValue(PETSC_NULL_OPTIONS,"-no_signal_handler","true",ierr))
```

before the call to `PetscInitialize()`. This prevents PETSc from defaulting to using a 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

```
PetscFPTrapPush(PetscFPTrap flag);
```

can be used within a program. 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.

```
PetscFPTrapPop(void);
```

should be used to revert to the previous handling of floating point exceptions before the call to `PetscFPTrapPush()`.

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

```
PetscCall(PetscRoutine(...));
```

Likewise, all memory allocations should be checked with

```
PetscCall(PetscMalloc1(n, &ptr));
```

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()`, `PetscCall()`. A simple example of usage is given below.

```
PetscErrorCode MyRoutine1()
{
    /* Declarations Here */
    PetscFunctionBeginUser;
    /* code here */
    PetscFunctionReturn(PETSC_SUCCESS);
}
```

4.8.7 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[3].

Recall that each configuration of the PETSc libraries is stored in a different directory, given by `$PETSC_DIR/$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.

4.8.8 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.

4.8.9 Graphics

The PETSc graphics library is not intended to compete with high-quality graphics packages. Instead, it is intended to be easy to use interactively with PETSc programs. We urge users to generate their publication-quality graphics using a professional graphics package. If a user wants to hook certain packages into PETSc, he or she should send a message to petsc-maint@mcs.anl.gov; we will see whether it is reasonable to try to provide direct interfaces.

Windows as PetscViewers

For drawing predefined PETSc objects such as matrices and vectors, one may first create a viewer using the command

```
PetscViewerDrawOpen(MPI_Comm comm, char *display, char *title, int x, int y, int w,␣
↪int h, PetscViewer *viewer);
```

This viewer may be passed to any of the `XXXView()` routines. Alternately, one may use command-line options to quickly specify viewer formats, including `PetscDraw`-based ones; see *Viewing From Options*.

To draw directly into the viewer, one must obtain the `PetscDraw` object with the command

```
PetscViewerDrawGetDraw(PetscViewer viewer, PetscDraw *draw);
```

[3] Note that this option is not required to use PETSc with C++

Then one can call any of the `PetscDrawXXX()` commands on the `draw` object. If one obtains the `draw` object in this manner, one does not call the `PetscDrawOpenX()` command discussed below.

Predefined viewers, `PETSC_VIEWER_DRAW_WORLD` and `PETSC_VIEWER_DRAW_SELF`, may be used at any time. Their initial use will cause the appropriate window to be created.

Implementations using OpenGL, TikZ, and other formats may be selected with `PetscDrawSetType()`. PETSc can also produce movies; see `PetscDrawSetSaveMovie()`, and note that command-line options can also be convenient; see the `PetscDrawSetFromOptions()` man page.

By default, PETSc drawing tools employ a private colormap, which remedies the problem of poor color choices for contour plots due to an external program's mangling of the colormap. Unfortunately, this may cause flashing of colors as the mouse is moved between the PETSc windows and other windows. Alternatively, a shared colormap can be used via the option `-draw_x_shared_colormap`.

Simple PetscDrawing

With the default format, one can open a window that is not associated with a viewer directly under the X11 Window System with the command

```
PetscDrawCreate(MPI_Comm comm, char *display, char *title, int x, int y, int w, int h,
↪ PetscDraw *win);
PetscDrawSetFromOptions(win);
```

All drawing routines are performed relative to the window's coordinate system and viewport. By default, the drawing coordinates are from `(0,0)` to `(1,1)`, where `(0,0)` indicates the lower left corner of the window. The application program can change the window coordinates with the command

```
PetscDrawSetCoordinates(PetscDraw win, PetscReal xl, PetscReal yl, PetscReal xr,␣
↪PetscReal yr);
```

By default, graphics will be drawn in the entire window. To restrict the drawing to a portion of the window, one may use the command

```
PetscDrawSetViewPort(PetscDraw win, PetscReal xl, PetscReal yl, PetscReal xr,␣
↪PetscReal yr);
```

These arguments, which indicate the fraction of the window in which the drawing should be done, must satisfy $0 \leq$ `xl` $\leq$ `xr` ≤ 1 and $0 \leq$ `yl` $\leq$ `yr` ≤ 1.

To draw a line, one uses the command

```
PetscDrawLine(PetscDraw win, PetscReal xl, PetscReal yl, PetscReal xr, PetscReal yr,␣
↪int cl);
```

The argument `cl` indicates the color (which is an integer between 0 and 255) of the line. A list of predefined colors may be found in `include/petscdraw.h` and includes `PETSC_DRAW_BLACK`, `PETSC_DRAW_RED`, `PETSC_DRAW_BLUE` etc.

To ensure that all graphics actually have been displayed, one should use the command

```
PetscDrawFlush(PetscDraw win);
```

When displaying by using double buffering, which is set with the command

```
PetscDrawSetDoubleBuffer(PetscDraw win);
```

all processes must call

```
PetscDrawFlush(PetscDraw win);
```

in order to swap the buffers. From the options database one may use `-draw_pause n`, which causes the PETSc application to pause `n` seconds at each `PetscDrawPause()`. A time of `-1` indicates that the application should pause until receiving mouse input from the user.

Text can be drawn with commands

```
PetscDrawString(PetscDraw win, PetscReal x, PetscReal y, int color, char *text);
PetscDrawStringVertical(PetscDraw win, PetscReal x, PetscReal y, int color, const␣
↪char *text);
PetscDrawStringCentered(PetscDraw win, PetscReal x, PetscReal y, int color, const␣
↪char *text);
PetscDrawStringBoxed(PetscDraw draw, PetscReal sxl, PetscReal syl, int sc, int bc,␣
↪const char text[], PetscReal *w, PetscReal *h);
```

The user can set the text font size or determine it with the commands

```
PetscDrawStringSetSize(PetscDraw win, PetscReal width, PetscReal height);
PetscDrawStringGetSize(PetscDraw win, PetscReal *width, PetscReal *height);
```

Line Graphs

PETSc includes a set of routines for manipulating simple two-dimensional graphs. These routines, which begin with `PetscDrawAxisDraw()`, are usually not used directly by the application programmer. Instead, the programmer employs the line graph routines to draw simple line graphs. As shown in the *listing below*, line graphs are created with the command

```
PetscDrawLGCreate(PetscDraw win, PetscInt ncurves, PetscDrawLG *ctx);
```

The argument `ncurves` indicates how many curves are to be drawn. Points can be added to each of the curves with the command

```
PetscDrawLGAddPoint(PetscDrawLG ctx, PetscReal *x, PetscReal *y);
```

The arguments `x` and `y` are arrays containing the next point value for each curve. Several points for each curve may be added with

```
PetscDrawLGAddPoints(PetscDrawLG ctx, PetscInt n, PetscReal **x, PetscReal **y);
```

The line graph is drawn (or redrawn) with the command

```
PetscDrawLGDraw(PetscDrawLG ctx);
```

A line graph that is no longer needed can be destroyed with the command

```
PetscDrawLGDestroy(PetscDrawLG *ctx);
```

To plot new curves, one can reset a linegraph with the command

```
PetscDrawLGReset(PetscDrawLG ctx);
```

The line graph automatically determines the range of values to display on the two axes. The user can change these defaults with the command

```
PetscDrawLGSetLimits(PetscDrawLG ctx, PetscReal xmin, PetscReal xmax, PetscReal ymin,␣
↪PetscReal ymax);
```

It is also possible to change the display of the axes and to label them. This procedure is done by first obtaining the axes context with the command

```
PetscDrawLGGetAxis(PetscDrawLG ctx, PetscDrawAxis *axis);
```

One can set the axes' colors and labels, respectively, by using the commands

```
PetscDrawAxisSetColors(PetscDrawAxis axis, int axis_lines, int ticks, int text);
PetscDrawAxisSetLabels(PetscDrawAxis axis, char *top, char *x, char *y);
```

It is possible to turn off all graphics with the option `-nox`. This will prevent any windows from being opened or any drawing actions to be done. This is useful for running large jobs when the graphics overhead is too large, or for timing.

The full example, Draw Test ex3, follows.

Listing: `src/classes/draw/tests/ex3.c`

```c
static char help[] = "Plots a simple line graph.\n";

#if defined(PETSC_APPLE_FRAMEWORK)
  #import <PETSc/petscsys.h>
  #import <PETSc/petscdraw.h>
#else

  #include <petscsys.h>
  #include <petscdraw.h>
#endif

int main(int argc, char **argv)
{
  PetscDraw           draw;
  PetscDrawLG         lg;
  PetscDrawAxis       axis;
  PetscInt            n = 15, i, nports = 1;
  int                 x = 0, y = 0, width = 400, height = 300;
  PetscBool           useports, flg;
  const char         *xlabel, *ylabel, *toplabel, *legend;
  PetscReal           xd, yd;
  PetscDrawViewPorts *ports = NULL;

  toplabel = "Top Label";
  xlabel   = "X-axis Label";
  ylabel   = "Y-axis Label";
  legend   = "Legend";

  PetscFunctionBeginUser;
  PetscCall(PetscInitialize(&argc, &argv, NULL, help));
  PetscCall(PetscOptionsGetMPIInt(NULL, NULL, "-x", &x, NULL));
  PetscCall(PetscOptionsGetMPIInt(NULL, NULL, "-y", &y, NULL));
  PetscCall(PetscOptionsGetMPIInt(NULL, NULL, "-width", &width, NULL));
  PetscCall(PetscOptionsGetMPIInt(NULL, NULL, "-height", &height, NULL));
  PetscCall(PetscOptionsGetInt(NULL, NULL, "-n", &n, NULL));
  PetscCall(PetscOptionsGetInt(NULL, NULL, "-nports", &nports, &useports));
```

(continues on next page)

(continued from previous page)

```
  PetscCall(PetscOptionsHasName(NULL, NULL, "-nolegend", &flg));
  if (flg) legend = NULL;
  PetscCall(PetscOptionsHasName(NULL, NULL, "-notoplabel", &flg));
  if (flg) toplabel = NULL;
  PetscCall(PetscOptionsHasName(NULL, NULL, "-noxlabel", &flg));
  if (flg) xlabel = NULL;
  PetscCall(PetscOptionsHasName(NULL, NULL, "-noylabel", &flg));
  if (flg) ylabel = NULL;
  PetscCall(PetscOptionsHasName(NULL, NULL, "-nolabels", &flg));
  if (flg) {
    toplabel = NULL;
    xlabel   = NULL;
    ylabel   = NULL;
  }

  PetscCall(PetscDrawCreate(PETSC_COMM_WORLD, 0, "Title", x, y, width, height, &
↪draw));
  PetscCall(PetscDrawSetFromOptions(draw));
  if (useports) {
    PetscCall(PetscDrawViewPortsCreate(draw, nports, &ports));
    PetscCall(PetscDrawViewPortsSet(ports, 0));
  }
  PetscCall(PetscDrawLGCreate(draw, 1, &lg));
  PetscCall(PetscDrawLGSetUseMarkers(lg, PETSC_TRUE));
  PetscCall(PetscDrawLGGetAxis(lg, &axis));
  PetscCall(PetscDrawAxisSetColors(axis, PETSC_DRAW_BLACK, PETSC_DRAW_RED, PETSC_DRAW_
↪BLUE));
  PetscCall(PetscDrawAxisSetLabels(axis, toplabel, xlabel, ylabel));
  PetscCall(PetscDrawLGSetLegend(lg, &legend));
  PetscCall(PetscDrawLGSetFromOptions(lg));

  for (i = 0; i <= n; i++) {
    xd = (PetscReal)(i - 5);
    yd = xd * xd;
    PetscCall(PetscDrawLGAddPoint(lg, &xd, &yd));
  }
  PetscCall(PetscDrawLGDraw(lg));
  PetscCall(PetscDrawLGSave(lg));

  PetscCall(PetscDrawViewPortsDestroy(ports));
  PetscCall(PetscDrawLGDestroy(&lg));
  PetscCall(PetscDrawDestroy(&draw));
  PetscCall(PetscFinalize());
  return 0;
}
```

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`

4.9 Developer Environments

Coding styles for most editors or integrated development environments are defined with EditorConfig in `.editorconfig`.

4.9.1 Emacs Users

Many PETSc developers use Emacs, which can be used as a "simple" text editor or a comprehensive development environment. For a more integrated development environment, we recommend using lsp-mode (or eglot) with clangd. The most convenient way to teach clangd what compilation flags to use is to install Bear ("build ear") and run:

```
bear make -B
```

which will do a complete rebuild (`-B`) of PETSc and capture the compilation commands in a file named `compile_commands.json`, which will be automatically picked up by clangd. You can use the same procedure when building examples or your own project. It can also be used with any other editor that supports clangd, including VS Code and Vim. When lsp-mode is accompanied by flycheck, Emacs will provide real-time feedback and syntax checking, along with refactoring tools provided by clangd.

The easiest way to install packages in recent Emacs is to use the "Options" menu to select "Manage Emacs Packages".

Tags

It is sometimes useful to cross-reference tags across projects. Regardless of whether you use lsp-mode, it can be useful to use GNU Global (install `gtags`) to provide reverse lookups (e.g. find all call sites for a given function) across all projects you might work on/browse. Tags for PETSc can be generated by running `make allgtags` from `$PETSC_DIR`, or one can generate tags for all projects by running a command such as

```
find $PETSC_DIR/{include,src,tutorials,$PETSC_ARCH/include} any/other/paths \
   -regex '.*\.\(cc\|hh\|cpp\|cxx\|C\|hpp\|c\|h\|cu\)$' \
   | grep -v ftn-auto | gtags -f -
```

from your home directory or wherever you keep source code. If you are making large changes, it is useful to either set this up to run as a cron job or to make a convenient alias so that refreshing is easy. Then add the following to `~/.emacs` to enable gtags and specify key bindings.

```
(when (require 'gtags)
  (global-set-key (kbd "C-c f") 'gtags-find-file)
  (global-set-key (kbd "C-c .") 'gtags-find-tag)
  (global-set-key (kbd "C-c r") 'gtags-find-rtag)
  (global-set-key (kbd "C-c ,") 'gtags-pop-stack))
(add-hook 'c-mode-common-hook
          '(lambda () (gtags-mode t))) ; Or add to existing hook
```

A more basic alternative to the GNU Global (`gtags`) approach that does not require adding packages is to use the builtin `etags` feature. First, run `make etags` from the PETSc home directory to generate the file `$PETSC_DIR/TAGS`, and then from within Emacs, run

```
M-x visit-tags-table
```

where `M` denotes the Emacs Meta key, and enter the name of the `TAGS` file. Then the command `M-.` will cause Emacs to find the file and line number where a desired PETSc function is defined. Any string in any of the PETSc files can be found with the command `M-x tags-search`. To find repeated occurrences, one can simply use `M-,` to find the next occurrence.

4.9.2 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 *Developer Environments*). If you have no prior attachment to a specific code editor, we recommend trying VS Code.

4.9.3 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.

Vim 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 *Developer Environments* 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 etags` from the PETSc home directory. Once the file exists, from Vi/Vim the user should issue the command

```
:set tags=CTAGS
```

from the `$PETSC_DIR` directory and enter the name of the `CTAGS` file. 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`.

See online tutorials for additional Vi/Vim options.

4.9.4 Eclipse Users

If you are interested in developing code that uses PETSc from Eclipse or developing PETSc in Eclipse and have knowledge of how to do indexing and build libraries in Eclipse, please contact us at petsc-dev@mcs.anl.gov.

One way to index and build PETSc in Eclipse is as follows.

1. Open "File→Import→Git→Projects from Git". In the next two panels, you can either add your existing local repository or download PETSc from Bitbucket by providing the URL. Most Eclipse distributions come with Git support. If not, install the EGit plugin. When importing the project, select the wizard "Import as general project".
2. Right-click on the project (or the "File" menu on top) and select "New → Convert to a C/C++ Project (Adds C/C++ Nature)". In the setting window, choose "C Project" and specify the project type as "Shared Library".
3. Right-click on the C project and open the "Properties" panel. Under "C/C++ Build → Builder Settings", set the Build directory to `$PETSC_DIR` and make sure "Generate Makefiles automatically" is unselected. Under the section "C/C++ General→Paths and Symbols", add the PETSc paths to "Includes".

```
    $PETSC_DIR/include
    $PETSC_DIR/$PETSC_ARCH/include

Under the section "C/C++ General\ :math:`\rightarrow`\ index", choose
"Use active build configuration".
```

1. Configure PETSc normally outside Eclipse to generate a makefile and then build the project in Eclipse. The source code will be parsed by Eclipse.

If you launch Eclipse from the Dock on Mac OS X, `.bashrc` will not be loaded (a known OS X behavior, for security reasons). This will be a problem if you set the environment variables `$PETSC_DIR` and `$PETSC_ARCH` in `.bashrc`. A solution which involves replacing the executable can be found at `/questions/829749/launch-mac-eclipse-with-environment-variables-set </questions/829749/launch-mac-eclipse-with-environment-variables-set>‘__. Alternatively, you can add `$PETSC_DIR` and `$PETSC_ARCH` manually under "Properties → C/C++ Build → Environment".

To allow an Eclipse code to compile with the PETSc include files and link with the PETSc libraries, a PETSc user has suggested the following.

1. Right-click on your C project and select "Properties → C/C++ Build → Settings"
2. A new window on the righthand side appears with various settings options. Select "Includes" and add the required PETSc paths,

```
$PETSC_DIR/include
$PETSC_DIR/$PETSC_ARCH/include
```

1. Select "Libraries" under the header Linker and set the library search path:

```
   $PETSC_DIR/$PETSC_ARCH/lib

and the libraries, for example
```

```
m, petsc, stdc++, mpichxx, mpich, lapack, blas, gfortran, dl, rt,gcc_s, pthread, X11
```

Another PETSc user has provided the following steps to build an Eclipse index for PETSc that can be used with their own code, without compiling PETSc source into their project.

1. In the user project source directory, create a symlink to the PETSc `src/` directory.
2. Refresh the project explorer in Eclipse, so the new symlink is followed.
3. Right-click on the project in the project explorer, and choose "Index → Rebuild". The index should now be built.
4. Right-click on the PETSc symlink in the project explorer, and choose "Exclude from build…" to make sure Eclipse does not try to compile PETSc with the project.

For further examples of using Eclipse with a PETSc-based application, see the documentation for LaMEM[4].

4.9.5 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 Microsoft 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:

```
# 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

# 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
```

(continues on next page)

[4] See the `doc/` directory at https://bitbucket.org/bkaus/lamem

(continued from previous page)

```
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` environment variable. For more information see the Qt Creator documentation.
- `QMAKE_CC` and `QMAKE_CXX`: Define which C/C++ compilers use.
- `QMAKE_LINK`: Defines the proper linker to be used. Relevant if compiling C++ projects.
- `QMAKE_CFLAGS`, `QMAKE_CXXFLAGS` and `QMAKE_LFLAGS`: Set the corresponding compile and linking flags.
- `SOURCES`: Source files to be compiled.
- `HEADERS`: Header files required by the application.
- `OTHER_FILES`: Other files to include (source, header, or any other extension). Note that none of the source files placed here are compiled.

More options can be included in a `.pro` file; see https://doc.qt.io/qt-5/qmake-project-files.html. Once the `.pro` file is generated, the user can simply open it via Qt Creator. Upon opening, one has the option to create two different build options, debug and release, and switch between the two. For more information on using the Qt Creator interface and other more advanced aspects of the IDE, refer to https://www.qt.io/qt-features-libraries-apis-tools-and-ide/

4.9.6 Visual Studio Users

To use PETSc from Microsoft 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.

4.9.7 Xcode IDE Users

See `doc_macos_install` for the standard Unix command line tools approach to development on macOS. The information below is only if you plan to write code within the Xcode IDE.

Apple Xcode IDE for macOS Applications

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`.

Apple Xcode IDE for iPhone/iPad iOS Applications

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`.

A thorough discussion of the procedure is given in Comparison of Migration Techniques for High-Performance Code to Android and iOS.

For Android, you must have your standalone bin folder in the path, so that the compilers are visible.

The installation process has not been tested for iOS or Android since 2017.

4.10 Advanced Features of Matrices and Solvers

This chapter introduces additional features of the PETSc matrices and solvers.

4.10.1 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 `cols` 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. This function is used internally by `PCFIELDSPLIT`.

One can also extract one or more submatrices per MPI process with

```
MatCreateSubMatrices(Mat A,PetscInt n,IS rows[],IS cols[],MatReuse call,Mat *B[]);
```

This extracts n (zero or more) matrices with the `rows[k]` and `cols[k]` of the matrix `A` into an array of sequential matrices `B[k]` on this process. If call is `MAT_INITIAL_MATRIX` it will create the array of matrices `B`. If call is `MAT_REUSE_MATRIX` it will reuse the `B` created with a previous call. The `IS` arguments are sequential. The array of matrices should be destroyed with `MatDestroySubMatrices()`. This function is used by `PCBJACOBI` and `PCASM`.

Each submatrix may be parallel, existing on a `MPI_Comm` associated with each pair of `IS rows[k]` and `cols[k]`, using

```
MatCreateSubMatricesMPI(Mat A,PetscInt n,IS rows[],IS cols[],MatReuse call,Mat *B[]);
```

Finally this version has a specialization

```
MatGetMultiProcBlock(Mat A, MPI_Comm subComm, MatReuse scall,Mat *subMat);
```

where collections of non-overlapping MPI processes share a single parallel matrix on their sub-communicator. This function is used by `PCBJACOBI` and `PCASM`.

The routine

```
MatCreateRedundantMatrix(Mat A,PetscInt nsubcomm,MPI_Comm subcomm,MatReuse reuse,Mat␣
↪*matredundant);
```

where `nsubcomm` copies of the entire matrix are stored, one on each `subcomm`. The routine `PetscSubcommCreate()` and its `PetscSubcomm` object may, but need not be, used to construct the `subcomm`.

The routine

```
MatMPIAdjToSeq(Mat A,Mat *B);
```

is a specialization that duplicates an entire `MATMPIADJ` matrix on each MPI process.

4.10.2 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 ILU, LU, ICC, Cholesky, and QR matrix factorizations are split into two or three stages depending on the user's needs. The first stage is to calculate an ordering for the matrix. The ordering generally is done to reduce fill in a sparse factorization; it does not make much sense for a dense matrix.

```
MatGetOrdering(Mat matrix,MatOrderingType type,IS* rowperm,IS* colperm);
```

The currently available alternatives for the ordering `type` are

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

These orderings can also be set through the options database.

Certain matrix formats may support only a subset of these. All of these orderings are symmetric at the moment; ordering routines that are not symmetric may be added. Currently we support orderings only for sequential matrices.

Users can add their own ordering routines by providing a function with the calling sequence

```
int reorder(Mat A,MatOrderingType type,IS* rowperm,IS* colperm);
```

Here `A` is the matrix for which we wish to generate a new ordering, `type` may be ignored and `rowperm` and `colperm` are the row and column permutations generated by the ordering routine. The user registers the ordering routine with the command

```
MatOrderingRegister(MatOrderingType ordname,char *path,char *sname,PetscErrorCode␣
↪(*reorder)(Mat,MatOrderingType,IS*,IS*)));
```

The input argument `ordname` is a string of the user's choice, either an ordering defined in `petscmat.h` or the name of a new ordering introduced by the user. See the code in `src/mat/impls/order/sorder.c` and other files in that directory for examples on how the reordering routines may be written.

Once the reordering routine has been registered, it can be selected for use at runtime with the command line option `-pc_factor_mat_ordering_type ordname`. If reordering from the API, the user should provide the `ordname` as the second input argument of `MatGetOrdering()`.

PETSc matrices interface to a variety of external factorization/solver packages via the `MatSolverType` which can be `MATSOLVERSUPERLU_DIST`, `MATSOLVERMUMPS`, `MATSOLVERPASTIX`, `MATSOLVERMKL_PARDISO`, `MATSOLVERMKL_CPARDISO`, `MATSOLVERUMFPACK`, `MATSOLVERCHOLMOD`, `MATSOLVERKLU`, `MATSOLVERCUSPARSE`, and `MATSOLVERCUDA`. The last three of which can run on GPUs, while `MATSOLVERSUPERLU_DIST` can partially run on GPUs. See `doc_linsolve` for a table of the factorization based solvers in PETSc.

Most of these packages compute their own orderings and cannot use ones provided so calls to the following routines with those packages can pass NULL as the `IS` permutations.

The following routines perform incomplete and complete, in-place, symbolic, and numerical factorizations for symmetric and nonsymmetric matrices:

```
MatICCFactor(Mat matrix,IS permutation,const MatFactorInfo *info);
MatCholeskyFactor(Mat matrix,IS permutation,const MatFactorInfo *info);
MatILUFactor(Mat matrix,IS rowpermutation,IS columnpermutation,const MatFactorInfo␣
↪*info);
MatLUFactor(Mat matrix,IS rowpermutation,IS columnpermutation,const MatFactorInfo␣
↪*info);
MatQRFactor(Mat matrix, 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 preferred out-of-place mode, first one obtains that matrix object that will hold the factor using

```
MatGetFactor(Mat matrix,MatSolverType package,MatFactorType ftype,Mat *factor);
```

and then performs the factorization

```
MatICCFactorSymbolic(Mat factor,Mat matrix,IS perm,const MatFactorInfo *info);
MatCholeskyFactorSymbolic(Mat factor,Mat matrix,IS perm,const MatFactorInfo *info);
MatCholeskyFactorNumeric(Mat factor,Mat matrix,const MatFactorInfo);

MatILUFactorSymbolic(Mat factor,Mat matrix,IS rowperm,IS colperm,const MatFactorInfo␣
↪*info);
MatLUFactorSymbolic(Mat factor,Mat matrix,IS rowperm,IS colperm,const MatFactorInfo␣
↪*info);
MatLUFactorNumeric(Mat factor,Mat matrix,const MatFactorInfo *info);

MatQRFactorSymbolic(Mat factor,Mat matrix,IS perm,const MatFactorInfo *info);
MatQRFactorNumeric(Mat factor,Mat matrix,const MatFactorInfo *info);
```

In this case, the contents of the matrix `factor` 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 `factor` 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 `factor` 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 provide our own sparse matrix factorization with pivoting for numerical stability. This is because trying to both reduce fill and do pivoting can become quite complicated. Instead, we provide a poor stepchild substitute. After one has obtained a reordering, with `MatGetOrdering(Mat A,MatOrdering type,IS *row,IS *col)` one may call

```
MatReorderForNonzeroDiagonal(Mat A,PetscReal tol,IS row, IS col);
```

which will try to reorder the columns to ensure that no values along the diagonal are smaller than `tol` in a absolute value. If small values are detected and corrected for, a nonsymmetric permutation of the rows and columns will result. This is not guaranteed to work, but may help if one was simply unlucky in the original ordering. When using the `KSP` solver interface the option `-pc_factor_nonzeros_along_diagonal <tol>` may be used. Here, `tol` is an optional tolerance to decide if a value is nonzero; by default it is `1.e-10`.

The external MatSolverType's MATSOLVERSUPERLU_DIST and MATSOLVERMUMPS do manage numerical pivoting internal to their API.

The external factorization packages each provide a wide number of options to chose from, details on these may be found by consulting the manual page for the solver package, such as, MATSOLVERSUPERLU_DIST. Most of the options can be easily set via the options database even when the factorization solvers are accessed via KSP.

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.

Some of the factorizations also support solves with multiple right-hand sides stored in a Mat using

```
MatMatSolve(Mat A,Mat B,Mat X);
```

and

```
MatMatSolveTranspose(Mat A,Mat B,Mat X);
```

Finally, MATSOLVERMUMPS, provides access to Schur complements obtained after partial factorizations as well as the inertia of a matrix via MatGetInertia().

4.10.3 Matrix-Matrix Products

PETSc matrices provide code for computing various matrix-matrix products. This section will introduce the two sets of routines available. For now consult MatCreateProduct() and MatMatMult().

4.10.4 Creating PC's Directly

Users obtain their preconditioner contexts from the KSP context with the command KSPGetPC(). It is possible to create, manipulate, and destroy PC contexts directly, although this capability should rarely be needed. To create a PC context, one uses the command

```
PCCreate(MPI_Comm comm,PC *pc);
```

The routine

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

sets the preconditioner method to be used. The routine

```
PCSetOperators(PC pc,Mat Amat,Mat Pmat);
```

set the matrices that are to be used with the preconditioner. The routine

```
PCGetOperators(PC pc,Mat *Amat,Mat *Pmat);
```

returns the values set with PCSetOperators().

The preconditioners in PETSc can be used in several ways. The two most basic routines simply apply the preconditioner or its transpose and are given, respectively, by

```
PCApply(PC pc,Vec x,Vec y);
PCApplyTranspose(PC pc,Vec x,Vec y);
```

In particular, for a matrix, B, that has been set via PCSetOperators(pc,Amat,Pmat), the routine PCApply(pc,x,y) computes $y = B^{-1}x$ by solving the linear system $By = x$ with the specified preconditioner method.

Additional preconditioner routines are

```
PCApplyBAorAB(PC pc,PCSide right,Vec x,Vec y,Vec work);
PCApplyBAorABTranspose(PC pc,PCSide right,Vec x,Vec y,Vec work);
PCApplyRichardson(PC pc,Vec x,Vec y,Vec work,PetscReal rtol,PetscReal atol, PetscReal␣
↪dtol,PetscInt maxits,PetscBool zeroguess,PetscInt *outits,
↪PCRichardsonConvergedReason*);
```

The first two routines apply the action of the matrix followed by the preconditioner or the preconditioner followed by the matrix depending on whether the right is PC_LEFT or PC_RIGHT. The final routine applies its iterations of Richardson's method. The last three routines are provided to improve efficiency for certain Krylov subspace methods.

A PC context that is no longer needed can be destroyed with the command

```
PCDestroy(PC *pc);
```

4.11 Running PETSc Tests

4.11.1 Quick start with the 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> check
```

For comprehensive testing of builds, the general invocation from the $PETSC_DIR is:

```
$ make PETSC_DIR=<PETSC_DIR> $PETSC_ARCH=<PETSC_ARCH> alltests
```

or

```
$ make [-j <n>] 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 test
```

which will create/use the directories tests/* in the current directory for generated test files. You may pass an additional argument TESTDIR=mytests to place these generated files elsewhere.

For a full list of options, use

```
$ make help-test
```

4.11.2 Understanding test output and more information

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)[1]. 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 test search='vec_vec_tutorials-ex6'
```

(see the discussion of `search` below), denotes the shell script:

```
$ $PETSC_DIR/$PETSC_ARCH/tests/vec/vec/tutorials/runex6.sh
```

These shell scripts can be run independently in those directories, and take arguments to show the commands run, change arguments, etc. Use the `-h` option to the shell script to see these options.

Often, you want to run only a subset of tests. Our makefiles use `gmake`'s wildcard syntax. In this syntax, `%` is a wild card character and is passed in using the `search` argument. Two wildcard characters cannot be used in a search, so the `searchin` argument is used to provide the equivalent of `%pattern%` search. The default examples have default arguments, and we often wish to test examples with various arguments; we use the `argsearch` argument for these searches. Like `searchin`, it does not use wildcards, but rather whether the string is within the arguments.

Some examples are:

```
$ make test search='ts*'                      # Run all TS examples
$ make test searchin='tutorials'              # Run all tutorials
$ make test search='ts*' searchin='tutorials' # Run all TS tutorials
$ make test argsearch='cuda'                  # Run examples with cuda in arguments
$ make test test-fail='1'
$ make 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:

[1] See https://testanything.org/tap-specification.html

```
$ make print-test argsearch='cuda'
```

To see all of the test targets which would be run, this command can be used:

```
$ make print-test
```

To learn more about the test system details, one can look at the the PETSc developers documentation.

4.11.3 Using the test harness for your own code

Select a package name, for example, `mypkg` and create a sub-directory with that name, say `/home/mine/mypackage/src/mypkg`. In any sub-directory of that directory named `tests` or `tutorials` put a PETSc makefile, for example, `src/ts/tutorials/makefile` and standalone test applications that the makefile can compile with, for example

```
$ make mytest
```

Include at the bottom of the test code a formatted comment indicating what tests should be run, see `test_harness`. Also select a directory where you wish the tests to be compiled and run, say `/home/mine/mytests`.

You can build and run the tests with

```
$ make -f ${PETSC_DIR}/gmakefile.test TESTSRCDIR=/home/mine/mypackage/src TESTDIR=/
↪home/mine/mytests pkgs=mypkg
```

There is not yet a mechanism to have your test code also link against your library, contact us for ideas.

BIBLIOGRAPHY

[For94] MPI Forum. MPI: a message-passing interface standard. *International J. Supercomputing Applications*, 1994.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[ref-EM17a] Jennifer B Erway and Roummel F Marcia. On solving large-scale limited-memory quasi-newton equations. *Linear Algebra and its Applications*, 515:196–225, 2017.

[ref-EM17b] Jennifer B Erway and Roummel F Marcia. On solving large-scale limited-memory quasi-Newton equations. *Linear Algebra and its Applications*, 515:196–225, 2017.

[ref-GL89] J. C. Gilbert and C. Lemarechal. Some numerical experiments with variable-storage quasi-newton algorithms. *Mathematical Programming*, 45:407–434, 1989.

[ref-Gri12] Andreas Griewank. Broyden updating, the good and the bad! *Optimization Stories, Documenta Mathematica. Extra Volume: Optimization Stories*, pages 301–315, 2012.

[ref-NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[BBKL11] Achi Brandt, James Brannick, Karsten Kahl, and Irene Livshits. Bootstrap AMG. *SIAM Journal on Scientific Computing*, 33(2):612–632, 2011.

[CS99] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Scientific Computing*, 21:792–797, 1999.

[CGS+94] Tony F Chan, Efstratios Gallopoulos, Valeria Simoncini, Tedd Szeto, and Charles H Tong. A quasi-minimal residual variant of the Bi-CGSTAB algorithm for nonsymmetric systems. *SIAM Journal on Scientific Computing*, 15(2):338–347, 1994.

[Eis81] S. Eisenstat. Efficient implementation of a class of CG methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[EES83] S.C. Eisenstat, H.C. Elman, and M.H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357, 1983.

[EHS+06] H.C. Elman, V.E. Howle, J. Shadid, R. Shuttleworth, and R. Tuminaro. Block preconditioners based on approximate commutators. *SIAM J. Sci. Comput.*, 27(5):1651–1668, 2006.

[EHS+08] H.C. Elman, V.E. Howle, J. Shadid, R. Shuttleworth, and R. Tuminaro. A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 227(1):1790–1808, 2008. URL: https://www.osti.gov/biblio/920807/.

[FGN92] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica. Cambridge University Press, 1992.

[Fre93] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Stat. Comput.*, 14:470–482, 1993.

[GAMV13] P. Ghysels, T.J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013.

[GV14] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications. doi:10.1016/j.parco.2013.06.001.

[HS52] Magnus R. Hestenes and Eduard Steifel. Methods of conjugate gradients for solving linear systems. *J. Research of the National Bureau of Standards*, 49:409–436, 1952.

[ISG15] Tobin Isaac, Georg Stadler, and Omar Ghattas. Solution of nonlinear Stokes equations discretized by high-order finite elements on nonconforming and anisotropic meshes, with application to ice sheet dynamics. *SIAM Journal on Scientific Computing*, 37(6):804–833, 2015. doi:10.1137/140974407.

[Not00] Yvan Notay. Flexible conjugate gradients. *SIAM Journal on Scientific Computing*, 22(4):1444–1460, 2000.

[PS75] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[SS86] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 44:856–869, 1986.

[Saa93] Youcef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993. doi:10.1137/0914028.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003. doi:10.1016/S1570-579X(01)80025-2.

[SSM16] P. Sanan, S. M. Schnepp, and D. A. May. Pipelined, flexible Krylov subspace methods. *SIAM Journal on Scientific Computing*, 38(5):C441–C470, 2016. doi:10.1137/15M1049130.

[SEKW01] D. Silvester, H. Elman, D. Kay, and A. Wathen. Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow. *Journal of Computational and Applied Mathematics*, 128(1-2):261–279, 2001.

[SBjorstadG96] Barry F. Smith, Petter Bjørstad, and William D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996. URL: http://www.mcs.anl.gov/~bsmith/ddbook.html.

[Son89] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[vdV03] H. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, 2003. ISBN 9780521818285.

[VanveekBM01] P. Vaněek, M. Brezina, and J. Mandel. Convergence of algebraic multigrid based on smoothed aggregation. *Numerische Mathematik*, 88(3):559–579, 2001.

[VanvekMB96] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.

[vandVorst92] H. A. van der Vorst. BiCGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[BS90] Peter N. Brown and Youcef Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[BKST15] Peter R. Brune, Matthew G. Knepley, Barry F. Smith, and Xuemin Tu. Composing scalable nonlinear algebraic solvers. *SIAM Review*, 57(4):535–565, 2015. http://www.mcs.anl.gov/papers/P2010-0112.pdf. URL: http://www.mcs.anl.gov/papers/P2010-0112.pdf, doi:10.1137/130936725.

[EW96] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact Newton method. *SIAM J. Scientific Computing*, 17:16–32, 1996.

[JP93] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[LPP+11] Sofie E. Leon, Glaucio H. Paulino, Anderson Pereira, Ivan F. M. Menezes, and Eduardo N. Lages. A unified library of nonlinear solution schemes. *Applied Mechanics Reviews*, 64(4):040803, July 2011. doi:10.1115/1.4006992.

[MoreSGH84] Jorge J. Moré, Danny C. Sorenson, Burton S. Garbow, and Kenneth E. Hillstrom. The MINPACK project. In Wayne R. Cowell, editor, *Sources and Development of Mathematical Software*, 88–111. 1984.

[PW98] M. Pernice and H. F. Walker. NITSOL: a Newton iterative solver for nonlinear systems. *SIAM J. Sci. Stat. Comput.*, 19:302–318, 1998.

[RCorreaC08] Manuel Ritto-Corrêa and Dinar Camotim. On the arc-length and other quadratic control methods: established, less known and new implementation procedures. *Computers & Structures*, 86(11):1353–1368, June 2008. doi:10.1016/j.compstruc.2007.08.003.

[DennisJrS83] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[ARS97] U.M. Ascher, S.J. Ruuth, and R.J. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25:151–167, 1997.

[AP98] Uri M Ascher and Linda R Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Volume 61. SIAM, 1998.

[BPR11] S. Boscarino, L. Pareschi, and G. Russo. Implicit-explicit Runge-Kutta schemes for hyperbolic systems and kinetic equations in the diffusion limit. Arxiv preprint arXiv:1110.4375, 2011.

[ColomesB16] Oriol Colomés and Santiago Badia. Segregated Runge–Kutta methods for the incompressible Navier–Stokes equations. *International Journal for Numerical Methods in Engineering*, 105(5):372–400, 2016.

[Con16] E.M. Constantinescu. Estimating global errors in time stepping. *ArXiv e-prints*, March 2016. arXiv:1503.05166.

[GKC13] F.X. Giraldo, J.F. Kelly, and E.M. Constantinescu. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA). *SIAM Journal on Scientific Computing*, 35(5):B1162–B1194, 2013. doi:10.1137/120876034.

[KC03] C.A. Kennedy and M.H. Carpenter. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.*, 44(1-2):139–181, 2003. doi:10.1016/S0168-9274(02)00138-1.

[PR05] L. Pareschi and G. Russo. Implicit-explicit Runge-Kutta schemes and applications to hyperbolic systems with relaxation. *Journal of Scientific Computing*, 25(1):129–155, 2005.

[RA05] J. Rang and L. Angermann. New Rosenbrock W-methods of order 3 for partial differential algebraic equations of index 1. *BIT Numerical Mathematics*, 45(4):761–787, 2005. doi:10.1007/s10543-005-0035-y.

[SVB+97] A. Sandu, J.G. Verwer, J.G. Blom, E.J. Spee, G.R. Carmichael, and F.A. Potra. Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock solvers. *Atmospheric Environment*, 31(20):3459–3472, 1997.

[AG07] Galen Andrew and Jianfeng Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the 24th international conference on Machine learning (ICML)*, 33–40. 2007.

[Arm66] L. Armijo. Minimization of functions having Lipschitz-continuous first partial derivatives. *Pacific Journal of Mathematics*, 16:1–3, 1966.

[Ber82] Dimitri P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM Journal on Control and Optimization*, 20:221–246, 1982.

[BPC+11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, and others. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[CGT00] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust-Region Methods*. SIAM, Philadelphia, Pennsylvania, 2000.

[CSV09] Andrew R. Conn, Katya Scheinberg, and Luís N. Vicente. *Introduction to Derivative-Free Optimization*. MPS/SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. ISBN 0-89871-460-5.

[Cot64] R. W. Cottle. *Nonlinear programs with positively bounded Jacobians*. PhD thesis, Department of Mathematics, University of California, Berkeley, California, 1964.

[FFK97] Francisco Facchinei, Andreas Fischer, and Christian Kanzow. A semismooth Newton method for variational inequalities: The case of box constraints. *Complementarity and Variational Problems: State of the Art*, 92:76, 1997.

[FP97] M. C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. *SIAM Review*, 39:669–713, 1997. URL: http://www.siam.org/journals/sirev/39-4/28596.html.

[Fis92] A. Fischer. A special Newton–type optimization method. *Optimization*, 24:269–284, 1992.

[GLL86] L. Grippo, F. Lampariello, and S. Lucidi. A nonmonotone line search technique for Newton's method. *SIAM Journal on Numerical Analysis*, 23:707–716, 1986.

[Hes69] Magnus R Hestenes. Multiplier and gradient methods. *Journal of optimization theory and applications*, 4(5):303–320, 1969.

[HP98] J. Huang and J. S. Pang. Option pricing and linear complementarity. *Journal of Computational Finance*, 2:31–60, 1998.

[Kar39] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[KT51] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, Berkeley and Los Angeles, 1951.

[LMore99] C.-J. Lin and J. J. Moré. Newton's method for large bound-constrained optimization problems. *SIOPT*, 9(4):1100–1127, 1999. URL: http://www.mcs.anl.gov/home/more/papers/nb.ps.gz.

[Mif77] R. Mifflin. Semismooth and semiconvex functions in constrained optimization. *SIAM Journal on Control and Optimization*, 15:957–972, 1977.

[MoreT91] Jorge J. Moré and G. Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIOPT*, 1:93–113, 1991.

[MoreT92] Jorge J. Moré and David Thuente. Line search algorithms with guaranteed sufficient decrease. Technical Report MCS-P330-1092, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[MFF+01] T. S. Munson, F. Facchinei, M. C. Ferris, A. Fischer, and C. Kanzow. The semismooth algorithm for large scale complementarity problems. *INFORMS Journal on Computing*, 2001.

[Nas50] J. F. Nash. Equilibrium points in N–person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[Pow69] Michael JD Powell. A method for nonlinear constraints in minimization problems. *Optimization*, pages 283–298, 1969.

[Qi93] L. Qi. Convergence analysis of some algorithms for solving nonsmooth equations. *Mathematics of Operations Research*, 18:227–244, 1993.

[QS93] L. Qi and J. Sun. A nonsmooth version of Newton's method. *Mathematical Programming*, 58:353–368, 1993.

[Roc74] R Tyrrell Rockafellar. Augmented lagrange multiplier functions and duality in nonconvex programming. *SIAM Journal on Control*, 12(2):268–285, 1974.

[DeLucaFK96] T. De Luca, F. Facchinei, and C. Kanzow. A semismooth equation approach to the solution of nonlinear complementarity problems. *Mathematical Programming*, 75:407–439, 1996.

[KortelainenLesinskiMore+10] M. Kortelainen, T. Lesinski, J. Moré, W. Nazarewicz, J. Sarich, N. Schunck, M. V. Stoitsov, and S. M. Wild. Nuclear energy density optimization. *Physical Review C*, 82(2):024313, 2010. doi:10.1103/PhysRevC.82.024313.

[Ala10] Frédéric Alauzet. Metric-based anisotropic mesh adaptation. https://pages.saclay.inria.fr/frederic.alauzet/cours/cea2010_V3.pdf, 2010.

[Getal] William Gropp and et. al. MPICH Web page. http://www.mpich.org. URL: http://www.mpich.org.

[HL91] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, August 1991.



Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov



Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC