

Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-95/11

PETSc 2.0 Users Manual

Revision 2.0.16

by

Satish Balay

William Gropp

Lois Curfman McInnes

Barry Smith

Mathematics and Computer Science Division

This manual is intended for use with PETSc 2.0.16.

February 1997

Contents

Abstract	vii
I Introduction to PETSc	1
1 Getting Started	3
1.1 Suggested Reading	4
1.2 Running PETSc Programs	5
1.3 Simple PETSc Examples	7
1.4 Directory Structure	17
II Programming with PETSc	19
2 Vectors	21
2.1 Creating and Assembling Vectors	21
2.2 Basic Vector Operations	22
2.3 Vector Internals	23
2.4 Index Sets	23
2.5 Scatters and Gathers	24
2.6 Application Orderings	25
2.7 Local to Global Mappings	26
2.8 Distributed Arrays	26
2.8.1 Creating Distributed Arrays	27
2.8.2 Local/Global Vectors and Scatters	28
2.8.3 Grid Information	28
2.9 Discrete Functions	38
3 Matrices	40
3.1 Creating and Assembling Matrices	40
3.1.1 Sparse Matrices	41
3.1.2 Dense Matrices	44
3.2 Basic Matrix Operations	45
3.3 Matrix-Free Matrices	46
3.4 Other Matrix Operations	46
4 SLES: Linear Equations Solvers	48
4.1 Using SLES	48
4.2 Solving Successive Linear Systems	49
4.3 KSP Component	49
4.3.1 Preconditioning within KSP	50
4.3.2 Convergence Tests	51
4.3.3 Convergence Monitoring	51
4.3.4 Understanding the Operators Spectrum	52

4.3.5	Other KSP Options	52
4.4	Preconditioners	53
4.4.1	ILU and ICC Preconditioners	53
4.4.2	SOR and SSOR Preconditioners	55
4.4.3	LU Factorization	55
4.4.4	Block Jacobi, Block Gauss-Seidel, and Overlapping Additive Schwarz Preconditioners	56
4.4.5	Shell Preconditioners	57
4.4.6	Multigrid Preconditioners	57
5	SNES: Nonlinear Solvers	59
5.1	Basic Usage	59
5.1.1	Solving Systems of Nonlinear Equations	64
5.1.2	Solving Unconstrained Minimization Problems	65
5.2	The Various Nonlinear Solvers	65
5.2.1	Line Search Techniques	65
5.2.2	Trust Region Methods	66
5.3	General Options	66
5.3.1	Convergence Tests	66
5.3.2	Convergence Monitoring	67
5.3.3	Checking Accuracy of Derivatives	67
5.4	Inexact Newton-like Methods	67
5.5	Matrix-Free Methods	68
5.6	Finite-Difference Jacobian Approximations	73
6	TS: Scalable ODE Solvers	76
6.1	Basic Usage	77
6.1.1	Solving Time-dependent Problems	77
6.1.2	Solving Steady-State Problems with Pseudo-Timestepping	78
7	Advanced Features of Matrices and Solvers	79
7.1	Matrix Factorization	79
7.2	Unimportant Details of KSP	81
7.3	Unimportant Details of PC	81
8	Graphics	83
8.1	Windows as Viewers	83
8.2	Simple Drawing	83
8.3	Line Graphs	84
8.4	Graphical Convergence Monitor	85
8.5	Other Graphical Output Types	86
9	PETSc Fortran Users	87
9.1	Differences between PETSc Interfaces for C and Fortran	87
9.1.1	Include Files	87
9.1.2	Error Checking	88
9.1.3	Array Arguments	88
9.1.4	Calling Fortran Routines from C (and C Routines from Fortran)	89
9.1.5	Passing Null Pointers	90
9.1.6	Duplicating Multiple Vectors	90
9.1.7	Matrix and Vector Indices	90
9.1.8	Setting Routines	90
9.1.9	Compiling and Linking Fortran Programs	90
9.1.10	Routines with Different Fortran Interfaces	91
9.2	Sample Fortran 77 Programs	91

III	Additional Information	103
10	Profiling	105
10.1	Basic Profiling Information	105
10.1.1	Interpreting -log_summary Output: The Basics	106
10.1.2	Interpreting -log_summary Output: Parallel Performance	106
10.1.3	Using -log and -log_all with PETScView	107
10.1.4	Using -log_mpe with Upshot/Nupshot	107
10.2	Profiling Application Codes	108
10.3	Profiling Multiple Sections of Code	108
10.4	Restricting Event Logging	109
10.5	Interpreting -log_info Output: Informative Messages	110
10.6	Time	110
10.7	Saving Output to a File	110
10.8	Accurate Profiling: Overcoming the Overhead of Paging	110
11	Hints for Performance Tuning	113
11.1	Compiler Options	113
11.2	Profiling	113
11.3	Aggregation	113
11.4	Efficient Memory Allocation	114
11.4.1	Sparse Matrix Assembly	114
11.4.2	Sparse Matrix Factorization	114
11.4.3	PetScMalloc() Calls	114
11.5	Data Structure Reuse	114
11.6	Numerical Experiments	115
11.7	Tips for Efficient Use of Linear Solvers	115
11.8	Finding Memory Leaks	115
11.9	Machine-specific Optimizations	116
11.10	System-related Problems	116
12	Other PETSc Features	117
12.1	Options	117
12.2	Viewers: Looking at PETSc Objects	118
12.3	Error Handling	119
12.4	Incremental Debugging	120
12.5	Complex Numbers	121
12.6	Emacs Users	121
12.7	VI Users	122
12.8	Parallel Communication	122
13	Makefiles	123
13.1	Our Makefile System	123
13.1.1	Makefile Commands	123
13.1.2	Customized Makefiles	124
13.2	PETSc Flags	124
13.3	Limitations	127
14	PETSc GUI Utilities	128
14.1	Getting Started	128
14.2	Using PETScView	128
14.2.1	Running PETScView	129
14.2.2	Loading a Configuration File	130
14.2.3	Printing a PETScView Object Tree	130
14.2.4	Exiting PETScView	130
14.2.5	The PETScView Simulation	130

14.2.6	Advanced Features	131
14.3	Using PETScOpts	133
14.3.1	Running PETScOpts	133
14.3.2	Getting Help	133
14.3.3	Exiting PETScOpts	133
15	Design and Implementations of the Abstract Classes	134
15.1	Names	135
15.2	Coding Conventions and Style Guide	135
15.3	Option Names	136
15.4	Implementation of Profiling	136
15.5	The Various Matrix Classes	137
15.5.1	Sequential AIJ Sparse Matrices	137
15.5.2	Parallel AIJ Sparse Matrices	137
15.5.3	Sequential Block AIJ Sparse Matrices	137
15.5.4	Parallel Block AIJ Sparse Matrices	137
15.5.5	Sequential Dense Matrices	138
15.5.6	Parallel Dense Matrices	138
15.5.7	Parallel Cyclic Block Dense Matrices	138
15.5.8	Parallel BlockSolve Sparse Matrices	138
15.5.9	Block Diagonal Sparse Matrices	138
15.5.10	Parallel Block Diagonal Sparse Matrices	139
15.6	Other Libraries and Packages	139
A	PETSc Function Reference List	140
A.1	Vector Routines	140
A.2	Matrix Routines	143
A.3	Simplified Linear Solvers	150
A.4	Preconditioners	151
A.5	Krylov Subspace Methods	156
A.6	Nonlinear Solvers	160
A.7	Timestepping, ODE Solvers	165
A.8	Index Sets, Distributed Arrays, and Application Orderings	167
A.9	Utility and System Routines	170
A.10	Viewers	175
A.11	Profiling	176
A.12	Graphics Routines	177
	Acknowledgments	181
	Bibliography	182
	Subject Index	184
	Function Index	186

Abstract

This manual describes the use of PETSc 2.0 for the numerical solution of partial differential equations and related problems on high-performance computers. The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on parallel (and serial) computers. PETSc 2.0 uses the MPI standard for all message-passing communication.

PETSc includes an expanding suite of parallel linear and nonlinear equation solvers that may be used in application codes written in Fortran, C, and C++. PETSc provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. In addition, PETSc includes growing support for distributed arrays. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem. By using techniques of object-oriented programming, PETSc provides enormous flexibility for users.

PETSc is a sophisticated set of software tools; as such, for some users it initially has a much steeper learning curve than a simple subroutine library. In particular, for individuals without some computer science background or experience programming in C, Pascal, or C++, it may require a large 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 make the efficient implementation of many application codes much simpler than “rolling them” yourself. For many simple (or even relatively complicated) 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.

Since PETSc is still under development, small changes in usage and calling sequences of PETSc routines will continue to occur. Although keeping one’s code up to date can be somewhat annoying, all PETSc users will be rewarded in the long run with a cleaner, better designed, and easier-to-use interface.

Part I

Introduction to PETSc

Chapter 1

Getting Started

The Portable, Extensible Toolkit for Scientific Computation (PETSc) has successfully demonstrated that the use of modern programming paradigms can ease the development of large-scale scientific application codes in Fortran, C, and C++. Begun several years ago, the software has evolved into a powerful set of tools for the numerical solution of partial differential equations and related problems on high-performance computers.

PETSc consists of a variety of components (similar to classes in C++), which are discussed in detail in Parts II and III of the users manual. Each component manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on the objects. The objects and operations in PETSc are derived from our long experiences with scientific computation. Some of the PETSc modules deal with

- index sets, including permutations;
- vectors;
- matrices (both sparse and dense);
- distributed arrays (useful for parallelizing regular grid-based problems);
- Krylov subspace methods;
- preconditioners;
- nonlinear solvers;
- timesteppers for solving time dependent (nonlinear) PDEs; and
- graphics devices.

Each of these components consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. Thus, PETSc provides clean and effective codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping.

The components enable easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility, and separates the issues of parallelism from the choice of algorithms. In addition, the PETSc infrastructure creates a foundation for building large-scale applications and extended suites of numerical routines.

It is useful to consider the interrelationships among different pieces of PETSc 2.0. Figure 1 is a diagram of some of the components of PETSc; Figure 2 presents several of the individual components in more detail. These figures illustrate the library's hierarchical organization, which enables users to employ the level of abstraction that is most appropriate for a particular problem.

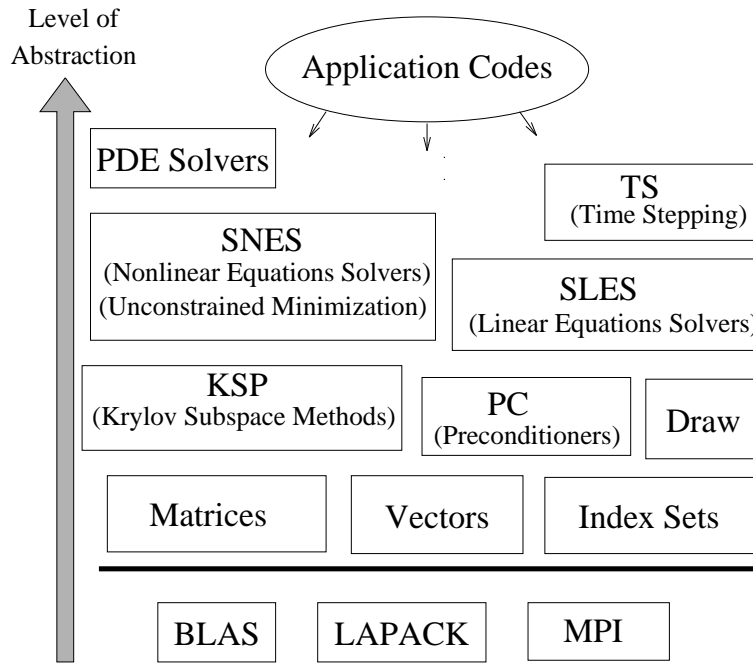


Figure 1: Organization of the PETSc Library

1.1 Suggested Reading

The *PETSc 2.0 Users Manual* replaces all of the previous PETSc users guides, including the SLES, KSP, and SNES manuals. The manual is divided into three parts:

- Part I - Introduction to PETSc
- Part II - Programming with PETSc
- Part III - Additional Information

Part I describes the basic procedure for using the PETSc library and presents two simple examples of solving linear systems with PETSc. This section conveys the typical style used throughout the library and enables the application programmer to begin using the software immediately. Part I is also distributed separately for individuals interested in an overview of the PETSc software, excluding the details of library usage. Readers of this separate distribution of Part I should note that all references within the text to particular chapters and sections indicate locations in the complete users manual.

Part II explains in detail the use of the various PETSc components, such as vectors, matrices, index sets, linear and nonlinear solvers, and graphics. Part III describes a variety of useful information, including profiling, the options database, viewers, error handling, makefiles, and some details of PETSc design.

The *PETSc 2.0 Users Manual* documents *all* of PETSc 2.0; thus, it can be rather intimidating for new users. We recommend that one initially read the entire document before proceeding with serious use of PETSc, but bear in mind that PETSc can be used efficiently before one understands all of the material presented here.

Note to Fortran Programmers: In most of the manual, the examples and calling sequences are given for the C/C++ family of programming languages. We follow this convention because we highly recommend that PETSc applications be coded in C or C++. However, pure Fortran 77 programmers can use most of the functionality of PETSc from Fortran, with only minor differences in the user interface. Chapter 9 provides a discussion of the differences between using PETSc from Fortran and C, as well as several complete Fortran 77 examples.

Man pages for all PETSc functions can be accessed in HTML format with the command `$(PETSC_DIR)-/bin/petscman [-xmosaic]`. The option `-xmosaic` indicates viewing man pages via Mosaic. The HTML

Parallel Numerical Components of PETSc

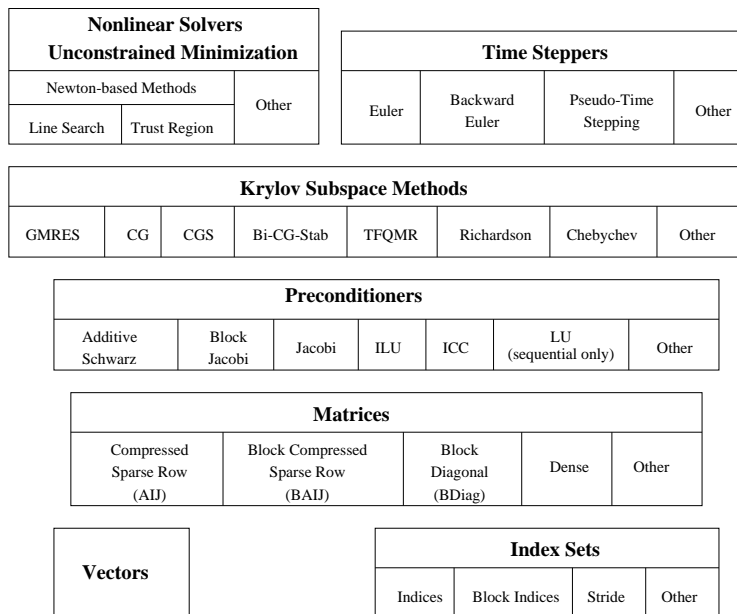


Figure 2: Numerical Components of PETSc

man pages provide hyperlinked indices (organized by both concepts and routine names) to the tutorial examples and enables easy movement among related topics. Within the PETSc distribution, the directory `$(PETSC_DIR)/docs` contains all documentation, including this manual and the man pages in PostScript and HTML formats. Note that one can also view the man pages in HTML format by loading the file `$(PETSC_DIR)/docs/www/www.html` into an HTML browser session that has been independently initiated. Emacs users may find the *etags* option to be extremely useful for exploring the PETSc source code. Details of this feature are provided in Section 12.6. Similarly, VI users may find the *ctags* option to be extremely useful. Details of this feature are provided in Section 12.7.

The PETSc source code is available by anonymous ftp from `ftp://info.mcs.anl.gov/pub/petsc` in the compressed tar files `petsc.tar.Z` and `petsc.tar.gz`. The file `manual.ps` contains the PostScript form of the *PETSc 2.0 Users Manual* in its entirety, while `intro.ps` includes only the introductory segment, Part I. The file `Installation` contains detailed instructions for installing PETSc. The complete PETSc distribution, users manual, man pages, and additional information are also available via the PETSc home page on the World Wide Web at `http://www.mcs.anl.gov/petsc/petsc.html`. The PETSc home page also contains details regarding installation, new features and changes in recent versions of PETSc, machines that we currently support, a troubleshooting guide, and a FAQ list for frequently asked questions.

1.2 Running PETSc Programs

Before using PETSc, the user must first set the environmental variable `PETSC_DIR`, indicating the full path of the PETSc home directory. For example, under the UNIX C shell a command of the form `setenv PETSC_DIR $HOME/petsc` can be placed in the user's `.cshrc` file. In addition, the user must set the environmental variable `PETSC_ARCH` to specify the architecture (e.g., `rs6000`, `sun4`, `solaris`) on which PETSc is being used. The utility `$(PETSC_DIR)/bin/petscarch` can be used for this purpose. For example,

```
setenv PETSC_ARCH '$PETSC_DIR/bin/petscarch'
```

can be placed in a `.cshrc` file. Thus, even if several machines of different types share the same filesystem, `PETSC_ARCH` will be set correctly when logging into any of them.

All PETSc programs use the MPI (Message Passing Interface) standard for message-passing communication [14]. 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 [8], the following command initiates a program that uses eight processors:

```
mpirun -np 8 petsc_program_name petsc_options
```

All PETSc 2.0-compliant programs support the use of the `-h` or `-help` option as well as the `-v` or `-version` option. For C and C++ programs these options can be placed on the command line. For most machines, options for Fortran programs can also be specified on the command line; otherwise, they can be set in the environmental variable `PETSC_OPTIONS` or placed in a file called `.petsrc` in the user's home directory. Under the UNIX C shell the environmental variable can be set with a command such as `setenv PETSC_OPTIONS "-help -version"`. See Section 12.1 for details.

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_summary` - summarize the program's performance
- `-fp_trap` - stop on floating-point exceptions
- `-trdump` - enable memory tracing; dump list of unfreed memory at conclusion of the run
- `-trmalloc` - enable memory tracing (by default this is activated for the debugging versions of PETSc)
- `-start_in_debugger` [`noxterm`,`dbx`,`xxgdb`] [`-display name`] - start all processes in debugger
- `-on_error_attach_debugger` [`noxterm`,`dbx`,`xxgdb`] [`-display name`] - start debugger only on encountering an error

By default the GNU debugger `gdb` is used when `-start_in_debugger` or `-on_error_attach_debugger` is specified. To employ either `xxgdb` or the common UNIX debugger `dbx`, one uses command line options as indicated above. On HP-UX machines the debugger `xdb` should be used instead of `dbx`; on RS/6000 machines the `xldb` debugger is supported as well. By default, the debugger will be started in a new xterm (to enable running separate debuggers on each process), unless the option `noxterm` is used. In order to handle the MPI startup phase, the debugger command "cont" should be used to continue execution of the program within the debugger. Rerunning the program through the debugger requires terminating the first job and restarting the processor(s); the usual "run" option in the debugger will not correctly handle the MPI startup and should not be used. Not all debuggers work on all machines, so the user may have to experiment to find one that works correctly.

Most PETSc programs begin with a call to

```
ierr = PetscInitialize(int *argc,char ***argv,char *file_name,char *help_message);
```

which initializes PETSc and MPI. The arguments `argc` and `argv` are the command line arguments delivered in all C and C++ programs. The argument `file_name` optionally indicates an alternative name for the PETSc options file, `.petsrc`, which resides by default in the user's home directory. Section 12.1 provides details regarding this file and the PETSc options database, which can be used for runtime customization. The final argument, `help_message`, 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_name,integer ierr)
```

`PetscInitialize()` automatically calls `MPI_Init()` if MPI has not been previously initialized. In certain circumstances in which MPI needs to be initialized directly (or is initialized by some other library), the user should first call `MPI_Init()` (or have the other library do it), and then call `PetscInitialize()`.

By default, `PetscInitialize()` sets the PETSc "world" communicator, given by `PETSC_COMM_WORLD`, to `MPI_COMM_WORLD`. This communicator specifies the processor group involved in certain operations (such as the default parallel viewers and performance summaries). Users who wish to employ PETSc routines on only a subset of processors within a larger parallel job, or who wish to use a "master" process to coordinate the work of "slave" PETSc processes, should specify an alternative communicator for `PETSC_COMM_WORLD` by calling

```
ierr = PetscSetCommWorld(MPI_Comm comm)
```

before calling `PetscInitialize()`, but, obviously, after calling `MPI_Init()`. `PetscSetCommWorld()` can only be called at most once per process. Most users will never need to use `PetscSetCommWorld()`.

As illustrated by the `PetscInitialize()` statements above, PETSc 2.0 routines return an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For the C/C++ interface, the error variable is the routine's return value,

while for the Fortran version, each PETSc routine has as its final argument an integer error variable. Error tracebacks are discussed in the following section.

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

```
ierr = PetscFinalize();
call PetscFinalize(ierr)
```

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

1.3 Simple PETSc Examples

To help the user start using PETSc immediately, we begin with a simple uniprocessor example in Figure 3 that solves the one-dimensional Laplacian problem with finite differences. This sequential code, which can be found in `$(PETSC_DIR)/src/sles/examples/tutorials/ex1.c`, illustrates the solution of a linear system with SLES, the simplified 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.

```
#ifndef lint
static char vcid[] = "$Id: ex1.c,v 1.59 1997/02/06 15:19:31 curfman Exp $";
#endif

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

/*T
  Concepts: SLES~Solving a system of linear equations (basic uniprocessor example);
  Routines: SLESCreate(); SLESSetOperators(); SLESSetFromOptions();
  Routines: SLESSolve(); SLESView(); SLESGetKSP(); SLESGetPC();
  Routines: KSPSetTolerances(); PCSetType();
  Processors: 1
T*/

/*
  Include "sles.h" so that we can use SLES solvers. Note that this file
  automatically includes:
    petsc.h - base PETSc routines    vec.h - vectors
    sys.h   - system routines        mat.h - matrices
    is.h     - index sets             ksp.h - Krylov subspace methods
    viewer.h - viewers                pc.h  - preconditioners
*/
#include "sles.h"
#include <stdio.h>

int main(int argc, char **args)
{
  Vec      x, b, u;      /* approx solution, RHS, exact solution */
  Mat      A;            /* linear system matrix */
  SLES      sles;         /* linear solver context */
  PC        pc;           /* preconditioner context */
  KSP       ksp;          /* Krylov subspace method context */
  double    norm;         /* norm of solution error */
  int       ierr, i, n = 10, col[3], its, flg, size;
  Scalar    none = -1.0, one = 1.0, value[3];

  PetscInitialize(&argc, &args, (char *)0, help);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (size != 1) SETERRA(1, 0, "This is a uniprocessor example only!");
  ierr = OptionsGetInt(PETSC_NULL, "-n", &n, &flg); CHKERRA(ierr);

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

```

/*
    Create matrix.  When using MatCreate(), the matrix format can
    be specified at runtime.
*/
ierr = MatCreate(MPI_COMM_WORLD,n,n,&A); CHKERRA(ierr);

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

/*
    Create vectors.  Note that we form 1 vector from scratch and
    then duplicate as needed.
*/
ierr = VecCreate(MPI_COMM_WORLD,n,&x); CHKERRA(ierr);
ierr = VecDuplicate(x,&b); CHKERRA(ierr);
ierr = VecDuplicate(x,&u); CHKERRA(ierr);

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

/* - - - - -
    Create the linear solver and set various options
    - - - - - */
/*
    Create linear solver context
*/
ierr = SLESCreate(MPI_COMM_WORLD,&sles); CHKERRA(ierr);

/*
    Set operators. Here the matrix that defines the linear system
    also serves as the preconditioning matrix.
*/
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN); CHKERRA(ierr);

/*
    Set linear solver defaults for this problem (optional).
    - By extracting the KSP and PC contexts from the SLES 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
      SLESSetFromOptions();
*/
ierr = SLESGetKSP(sles,&ksp); CHKERRA(ierr);
ierr = SLESGetPC(sles,&pc); CHKERRA(ierr);
ierr = PCSetType(pc,PCJACOBI); CHKERRA(ierr);
ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,
    PETSC_DEFAULT); CHKERRA(ierr);

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

```

```

    routines.
*/
ierr = SLESSetFromOptions(sles); CHKERRA(ierr);

/* -----
           Solve the linear system
----- */
/*
    Solve linear system
*/
ierr = SLESSolve(sles,b,x,&its); CHKERRA(ierr);

/*
    View solver info; we could instead use the option -sles_view
*/
ierr = SLESView(sles,VIEWER_STDOUT_WORLD); CHKERRA(ierr);

/* -----
           Check solution and clean up
----- */
/*
    Check the error
*/
ierr = VecAXPY(&none,u,x); CHKERRA(ierr);
ierr = VecNorm(x,NORM_2,&norm); CHKERRA(ierr);
if (norm > 1.e-12)
    PetscPrintf(MPI_COMM_WORLD,"Norm of error %g, Iterations %d\n",norm,its);
else
    PetscPrintf(MPI_COMM_WORLD,"Norm of error < 1.e-12, Iterations %d\n",its);

/*
    Free work space. All PETSc objects should be destroyed when they
    are no longer needed.
*/
ierr = VecDestroy(x); CHKERRA(ierr); ierr = VecDestroy(u); CHKERRA(ierr);
ierr = VecDestroy(b); CHKERRA(ierr); ierr = MatDestroy(A); CHKERRA(ierr);
ierr = SLESDestroy(sles); CHKERRA(ierr);
PetscFinalize();
return 0;
}

```

Figure 3: Example of Uniprocessor PETSc Code

Include Files

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

```
#include "sles.h"
```

where `sles.h` is the include file for the SLES component. Each PETSc program must specify an include file that corresponds to the highest-level PETSc objects needed within the program; all of the required lower-level include files are automatically included within the higher-level files. For example, `sles.h` includes `mat.h` (matrices), `vector.h` (vectors), and `petsc.h` (base PETSc file). The PETSc include files are located in the directory `$(PETSC_DIR)/include`. See Section 9.1.1 for a discussion of PETSc include files in Fortran programs.

The Options Database

As shown in Figure 3, the user can input control data at run time using the options database. In this example the command `OptionsGetInt(PETSC_NULL,"-n",&n,&flg);` checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged. A complete description of the options database may be found in Section 12.1.

Vectors

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

```
ierr = VecCreate(MPI_Comm comm,int M,Vec *x);
```

where **comm** denotes the MPI communicator. Additional vectors of the same type can be formed with

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

The commands

```
ierr = VecSet(Scalar *value,Vec x);  
ierr = VecSetValues(Vec x,int n,int *indices,Scalar *values,INSERT_VALUES);
```

respectively set all the components of a vector to a particular scalar value and assign a different value to each component. More detailed information about PETSc vectors, including their basic operations, scattering/gathering, index sets, and distributed arrays, is discussed in Chapter 2.

Note the use of the PETSc variable type **Scalar** in this example. The **Scalar** is simply defined to be **double** in C/C++ (or correspondingly **double precision** in Fortran) for versions of PETSc that have *not* been compiled for use with complex numbers. The **Scalar** data type enables identical code to be used when the PETSc libraries have been compiled for use with complex numbers. Section 12.5 discusses the use of complex numbers in PETSc programs.

Matrices

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

```
ierr = MatCreate(MPI_Comm comm,int M,int N,Mat *A);
```

where the matrix format can be specified at runtime. Values can then be set with the command

```
ierr = MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar *values,INSERT_VALUES);
```

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

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

Chapter 3 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, $\mathbf{Ax} = \mathbf{b}$, the user can then use SLES to solve the system with the following sequence of commands:

```
ierr = SLESCreate(MPI_Comm comm,SLES *sles);  
ierr = SLESSetOperators(SLES sles,Mat A,Mat PrecA,MatStructure flag);  
ierr = SLESSetFromOptions(SLES sles);  
ierr = SLESSolve(SLES sles,Vec b,Vec x,int *its);  
ierr = SLESDestroy(SLES sles);
```

The user first creates the SLES context and sets the operators associated with the system (linear system matrix and optionally different preconditioning matrix). The user then sets various options for customized solution, solves the linear system, and finally destroys the SLES context. We emphasize the command **SLESSetFromOptions()**, which enables the user to customize the linear solution method at runtime by using the options database, which is discussed in Section 12.1. Through this database, the user not only can select an iterative method and preconditioner, but also can prescribe the convergence tolerance, set various monitoring routines, and so forth (see, e.g., Figure 7).

Chapter 4 describes in detail the SLES package, including the PC and KSP components for preconditioners and Krylov subspace methods.

Error Checking

All PETSc 2.0 routines return an integer indicating whether an error has occurred during the call. The PETSc macro `CHKERRQ(ierr)` checks the value of `ierr` and calls the PETSc 2.0 error handler upon error detection. `CHKERRQ(ierr)` should be used in all subroutines to enable a complete error traceback. A variant of this macro, `CHKERRA(ierr)`, should be used in the main program to enable correct termination of all processes when an error is encountered. In Figure 4 we indicate a traceback generated by error detection within a sample PETSc program. The error occurred on line 858 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 49. See Section 9.1.2 for details regarding error checking when using the PETSc Fortran interface.

```
eagle>mpirun ex3 -m 1000000000000
mat/impls/aij/seq/aij.c line # 858 No memory
mat/interface/matrix.c line # 123
ex3.c line # 49
Aborting program!
```

Figure 4: Example of Error Traceback

Parallel Programming

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

```
ierr = MatCreate(MPI_Comm comm,int M,int N,Mat *A);
ierr = VecCreate(MPI_Comm comm,int M,Vec *x);
ierr = SLESCreate(MPI_Comm comm,SLES *sles);
```

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

The next example, given in Figure 5, illustrates the solution of a linear system in parallel. This code, corresponding to `$(PETSC_DIR)/src/sles/examples/tutorials/ex2.c`, handles the two-dimensional Laplacian discretized with finite differences, where the linear system is again solved with SLES. The code performs the same tasks as the sequential version within Figure 3. 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 Figures 3 and 5 is that each processor forms only its local part of the matrix and vectors in the parallel case.

```

#ifndef lint
static char vcid[] = "$Id: ex2.c,v 1.61 1997/02/06 15:19:35 curfman Exp $";
#endif

/* Usage: mpirun ex2 [-help] [all PETSc options] */

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

/*T
  Concepts: SLES`Solving a system of linear equations (basic parallel example);
  Concepts: SLES`Laplacian, 2d
  Concepts: Laplacian, 2d
  Routines: SLESCreate(); SLESSetOperators(); SLESSetFromOptions();
  Routines: SLESSolve(); SLESGetKSP(); SLESGetPC();
  Routines: KSPSetTolerances(); PCSetType();
  Processors: n
T*/

/*
  Include "sles.h" so that we can use SLES solvers. Note that this file
  automatically includes:
    petsc.h - base PETSc routines    vec.h - vectors
    sys.h   - system routines        mat.h - matrices
    is.h    - index sets             ksp.h - Krylov subspace methods
    viewer.h - viewers                pc.h  - preconditioners
*/
#include "sles.h"
#include <stdio.h>

int main(int argc, char **args)
{
  Vec      x, b, u; /* approx solution, RHS, exact solution */
  Mat      A;       /* linear system matrix */
  SLES     sles;     /* linear solver context */
  PetscRandom rctx; /* random number generator context */
  double    norm;    /* norm of solution error */
  int       i, j, I, J, Istart, Iend, ierr, m = 8, n = 7, its, flg;
  Scalar    v, one = 1.0, neg_one = -1.0;

  /* These variables are currently unused */
  /* PC      pc; */ /* preconditioner context */
  /* KSP     ksp; */ /* Krylov subspace method context */

  PetscInitialize(&argc, &args, (char *)0, help);
  ierr = OptionsGetInt(PETSC_NULL, "-m", &m, &flg); CHKERRA(ierr);
  ierr = OptionsGetInt(PETSC_NULL, "-n", &n, &flg); CHKERRA(ierr);

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

  /*
   Create parallel matrix, specifying only its global dimensions.
   When using MatCreate(), the matrix format can be specified at
   runtime. Also, the parallel partitioning of the matrix is
   determined by PETSc at runtime.
  */
  ierr = MatCreate(MPI_COMM_WORLD, m*n, m*n, &A); CHKERRA(ierr);

  /*
   Currently, all PETSc parallel matrix formats are partitioned by
   contiguous chunks of rows across the processors. Determine which
   rows of the matrix are locally owned.
  */

```

```

ierr = MatGetOwnershipRange(A,&Istart,&Iend); CHKERRA(ierr);

/*
  Set matrix elements for the 2-D, five-point stencil in parallel.
  - Each processor needs to insert only elements that it owns
    locally (but any non-local elements will be sent to the
    appropriate processor during matrix assembly).
  - Always specify global rows and columns of matrix entries.
*/
for ( I=Istart; I<Iend; I++ ) {
  v = -1.0; i = I/n; j = I - i*n;
  if ( i>0 ) {J = I - n; MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES); CHKERRA(ierr);}
  if ( i<m-1 ) {J = I + n; MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES); CHKERRA(ierr);}
  if ( j>0 ) {J = I - 1; MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES); CHKERRA(ierr);}
  if ( j<n-1 ) {J = I + 1; MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES); CHKERRA(ierr);}
  v = 4.0; MatSetValues(A,1,&I,1,&I,&v,INSERT_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.
*/
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRA(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRA(ierr);

/*
  Create parallel vectors.
  - When using VecCreate(), 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.
  - Note: We form 1 vector from scratch and then duplicate as needed.
*/
ierr = VecCreate(MPI_COMM_WORLD,m*n,&u); CHKERRA(ierr);
ierr = VecDuplicate(u,&b); CHKERRA(ierr);
ierr = VecDuplicate(b,&x); CHKERRA(ierr);

/*
  Set exact solution; then compute right-hand-side vector.
  By default we use an exact solution of a vector with all
  elements of 1.0; Alternatively, using the runtime option
  -random_sol forms a solution vector with random components.
*/
ierr = OptionsHasName(PETSC_NULL,"-random_exact_sol",&flg); CHKERRA(ierr);
if (flg) {
  ierr = PetscRandomCreate(MPI_COMM_WORLD,RANDOM_DEFAULT,&rctx); CHKERRA(ierr);
  ierr = VecSetRandom(rctx,u); CHKERRA(ierr);
  ierr = PetscRandomDestroy(rctx); CHKERRA(ierr);
} else {
  ierr = VecSet(&one,u); CHKERRA(ierr);
}
ierr = MatMult(A,u,b); CHKERRA(ierr);

/*
  View the exact solution vector if desired
*/
ierr = OptionsHasName(PETSC_NULL,"-view_exact_sol",&flg); CHKERRA(ierr);
if (flg) {ierr = VecView(u,VIEWER_STDOUT_WORLD); CHKERRA(ierr);}

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

/*
  Create linear solver context

```

```

*/
ierr = SLESCreate(MPI_COMM_WORLD,&sles); CHKERRA(ierr);

/*
   Set operators. Here the matrix that defines the linear system
   also serves as the preconditioning matrix.
*/
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN); CHKERRA(ierr);

/*
   Set linear solver defaults for this problem (optional).
   - By extracting the KSP and PC contexts from the SLES 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
     SLESSetFromOptions(). All of these defaults can be
     overridden at runtime, as indicated below.
*/
/*
   We comment out this section of code since the Jacobi
   preconditioner is not a good general default.

   ierr = SLESGetKSP(sles,&ksp); CHKERRA(ierr);
   ierr = SLESGetPC(sles,&pc); CHKERRA(ierr);
   ierr = PCSetType(pc,PCJACOBI); CHKERRA(ierr);
   ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,
       PETSC_DEFAULT); CHKERRA(ierr);
*/

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

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

ierr = SLESSolve(sles,b,x,&its); CHKERRA(ierr);

/* - - - - -
           Check solution and clean up
   - - - - - */

/*
   Check the error
*/
ierr = VecAXPY(&neg_one,u,x); CHKERRA(ierr);
ierr = VecNorm(x,NORM_2,&norm); CHKERRA(ierr);

/*
   Print convergence information. PetscPrintf() produces a single
   print statement from all processes that share a communicator.
*/
if (norm > 1.e-12)
    PetscPrintf(MPI_COMM_WORLD,"Norm of error %g iterations %d\n",norm,its);
else
    PetscPrintf(MPI_COMM_WORLD,"Norm of error < 1.e-12 Iterations %d\n",its);

/*
   Free work space. All PETSc objects should be destroyed when they
   are no longer needed.
*/
ierr = SLESDestroy(sles); CHKERRA(ierr);

```

```

ierr = VecDestroy(u); CHKERRA(ierr); ierr = VecDestroy(x); CHKERRA(ierr);
ierr = VecDestroy(b); CHKERRA(ierr); ierr = MatDestroy(A); CHKERRA(ierr);

/*
  Always call PetscFinalize() before exiting a program. This routine
  - finalizes the PETSc libraries as well as MPI
  - provides summary and diagnostic information if certain runtime
    options are chosen (e.g., -log_summary). See PetscFinalize()
    manpage for more information.
*/
PetscFinalize();
return 0;
}

```

Figure 5: Example of Multiprocessor PETSc Code

Compiling and Running Programs

Figure 6 illustrates compiling and running a PETSc program using MPICH. Note that different sites may have slightly different library and compiler names. See Chapter 13 for a discussion about compiling PETSc programs. Users who are experiencing difficulties linking PETSc programs should refer to the troubleshooting guide via the PETSc WWW home page or given in the file `$(PETSC_DIR)/Troubleshooting`.

```

eagle> make BOPT=g ex2
gcc -DPETSC_ARCH_sun4 -pipe -c -I../.../ -I../.../include
-I/usr/local/mpi/include -I../.../src -g
-DPETSC_DEBUG -DPETSC_MALLOC -DPETSC_LOG ex1.c
gcc -g -DPETSC_DEBUG -DPETSC_MALLOC -DPETSC_LOG -o ex1 ex1.o
/home/bsmith/petsc/lib/libg/sun4/libpetscsles.a
-L/home/bsmith/petsc/lib/libg/sun4 -lpetscstencil -lpetscgrid -lpetscsles
-lpetscksp -lpetscmat -lpetscvec -lpetscsys -lpetscdraw
/usr/local/lapack/lib/lapack.a /usr/local/lapack/lib/blas.a
/usr/lang/SC1.0.1/libF77.a -lm /usr/lang/SC1.0.1/libm.a -lX11
/usr/local/mpi/lib/sun4/ch_p4/libmpi.a
/usr/lib/debug/malloc.o /usr/lib/debug/mallocmap.o
/usr/lang/SC1.0.1/libF77.a -lm /usr/lang/SC1.0.1/libm.a -lm
rm -f ex1.o
eagle> mpirun ex2
Norm of error 3.6618e-05 iterations 7
eagle>
eagle> mpirun -np 2 ex2
Norm of error 5.34462e-05 iterations 9

```

Figure 6: Running a PETSc Program

As shown in Figure 7, the option `-log_summary` activates printing of a performance summary, including times, floating-point operation (flop) rates, and message-passing activity for the various PETSc events. Chapter 10 provides details about profiling, including interpretation of the output data within Figure 7 and more information about monitoring parallel programs. This particular example involves the solution of a linear system on one processor using GMRES and ILU. The low floating-point operation (flop) rates in this example are due to the fact that the code was run on a tiny matrix. We include this example merely to demonstrate the ease of extracting performance information from PETSc.

```
eagle> mpirun ex1 -n 1000 -pc_type ilu -ksp_type gmres -ksp_rtol 1.e-7 -log_summary
----- PETSc Performance Summary: -----

ex1 on a sun4 named merlin.mcs.anl.gov with 1 processor, by curfman Wed Aug 7 17:24:27 1996
```

	Max	Min	Avg	Total
Time (sec):	1.150e-01	1.0	1.150e-01	
Objects:	1.900e+01	1.0	1.900e+01	
Flops:	3.998e+04	1.0	3.998e+04	3.998e+04
Flops/sec:	3.475e+05	1.0		3.475e+05
MPI Messages:	0.000e+00	0.0	0.000e+00	0.000e+00
MPI Messages:	0.000e+00	0.0	0.000e+00	0.000e+00 (lengths)
MPI Reductions:	0.000e+00	0.0		

```
-----
```

Phase	Count	Time (sec)		Flops/sec		Mess	Avg len	Reduct	-- Global --					
		Max	Ratio	Max	Ratio				%T	%F	%M	%L	%R	
MatMult	2	2.553e-03	1.0	3.9e+06	1.0	0.0e+00	0.0e+00	0.0e+00	2	25	0	0	0	
MatAssemblyBegin	1	2.193e-05	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	0	0	0	0	0	
MatAssemblyEnd	1	5.004e-03	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	4	0	0	0	0	
MatGetReordering	1	3.004e-03	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	3	0	0	0	0	
MatILUFctrSymbol	1	5.719e-03	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	5	0	0	0	0	
MatLUFactorNumer	1	1.092e-02	1.0	2.7e+05	1.0	0.0e+00	0.0e+00	0.0e+00	9	7	0	0	0	
MatSolve	2	4.193e-03	1.0	2.4e+06	1.0	0.0e+00	0.0e+00	0.0e+00	4	25	0	0	0	
MatSetValues	1000	2.461e-02	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	21	0	0	0	0	
VecDot	1	2.060e-04	1.0	9.7e+06	1.0	0.0e+00	0.0e+00	0.0e+00	0	5	0	0	0	
VecNorm	3	5.870e-04	1.0	1.0e+07	1.0	0.0e+00	0.0e+00	0.0e+00	1	15	0	0	0	
VecScale	1	1.640e-04	1.0	6.1e+06	1.0	0.0e+00	0.0e+00	0.0e+00	0	3	0	0	0	
VecCopy	1	3.101e-04	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	0	0	0	0	0	
VecSet	3	5.029e-04	1.0	0.0e+00	0.0	0.0e+00	0.0e+00	0.0e+00	0	0	0	0	0	
VecAXPY	3	8.690e-04	1.0	6.9e+06	1.0	0.0e+00	0.0e+00	0.0e+00	1	15	0	0	0	
VecMAXPY	1	2.550e-04	1.0	7.8e+06	1.0	0.0e+00	0.0e+00	0.0e+00	0	5	0	0	0	
SLESSolve	1	1.288e-02	1.0	2.2e+06	1.0	0.0e+00	0.0e+00	0.0e+00	11	70	0	0	0	
SLESSetUp	1	2.669e-02	1.0	1.1e+05	1.0	0.0e+00	0.0e+00	0.0e+00	23	7	0	0	0	
KSPGMRESOrthog	1	1.151e-03	1.0	3.5e+06	1.0	0.0e+00	0.0e+00	0.0e+00	1	10	0	0	0	
PCSetUp	1	2.024e-02	1.0	1.5e+05	1.0	0.0e+00	0.0e+00	0.0e+00	18	7	0	0	0	
PCApply	2	4.474e-03	1.0	2.2e+06	1.0	0.0e+00	0.0e+00	0.0e+00	4	25	0	0	0	

```
-----
```

Memory usage is given in bytes:

Object Type	Creations	Destructions	Memory	Descendants' Mem.
Viewer	3	3	0	0
Index set	3	3	12420	0
Vector	8	8	65728	0
Matrix	2	2	184924	4140
Krylov Solver	1	1	16892	41080
Preconditioner	1	1	0	64872
SLES	1	1	0	122844

Figure 7: Running a PETSc Program with Profiling

Writing Application Codes with PETSc

The examples throughout the PETSc library demonstrate the details of using the software and can serve as templates for developing custom application programs. We suggest that new PETSc users examine some programs in the directories $\$(\text{PETSC_DIR})/\text{src}/\langle\text{component}\rangle/\text{examples}/\text{tutorials}$, where $\langle\text{component}\rangle$ denotes any of the PETSc component directories (listed in the following section), such as `sn`es or `sles`. Note that we are in the process of organizing the examples in tutorial and test categories. Currently some PETSc components have examples only in the directory $\$(\text{PETSC_DIR})/\text{src}/\langle\text{component}\rangle/\text{examples}/\text{tests}$; more tutorial examples will be forthcoming. The HTML version of the man pages provides indices (organized by both routine names and concepts) to the tutorial examples.

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

1. Install and test PETSc according to the instructions in the file $\$(\text{PETSC_DIR})/\text{Installation}$.

2. Copy one of the many PETSc examples in the component directory that corresponds to the class of problem of interest (e.g., for linear solvers, see `$(PETSC_DIR)/src/sles/examples/tutorials`).
3. Copy the corresponding makefile within the example directory; compile and run the example program.
4. Use the example program as a starting point for developing a custom code.

1.4 Directory Structure

We conclude this introduction with an overview of the organization of the PETSc software. As shown in Figure 8, the root directory of PETSc contains the following directories:

- **docs** - All documentation for PETSc. The files **manual.ps** and **manual.html** contain the users manual in PostScript and HTML formats, respectively. Includes the subdirectories
 - **www** (HTML man pages).
- **bin** - Utilities and short scripts for use with PETSc, including
 - **petscman** (man page viewer),
 - **petsarch** (utility for setting **PETSC_ARCH** environmental variable),
 - **petscview** (GUI utility for high-level visualization of program activity), and
 - **petscopts** (GUI utility for setting runtime options).
- **bmake** - Base PETSc makefile directory. Includes subdirectories for various architectures.
- **include** - All include files for PETSc that are visible to the user.
- **include/FINCLUDE** - PETSc include files for Fortran programmers using the .F suffix (recommended).
- **include/finclude** - PETSc include files for Fortran programmers using the .f suffix.
- **include/pinclude** - Private PETSc include files that should *not* be used by application programmers.
- **src** - The source code for all PETSc components, which currently includes
 - is** - index sets,
 - vec** - vectors,
 - da** - distributed arrays,
 - mat** - matrices,
 - ksp** - Krylov space accelerators,
 - pc** - preconditioners,
 - sles** - complete linear equations solvers,
 - snes** - nonlinear solvers,
 - ts** - ODE solvers and timestepping,
 - sys** - general system-related routines,
 - plog** - PETSc logging and profiling routines,
 - draw** - simple graphics,
 - ao** - application orderings,
 - fortran** - Fortran interface stubs,
 - mpiuni** - experimental, stripped-down uniprocessor MPI version, and
 - contrib** - contributed modules that use PETSc but are not part of the official PETSc package.

We encourage users who have developed such code that they wish to share with others to let us know by writing to `petsc-maint@mcs.anl.gov`.

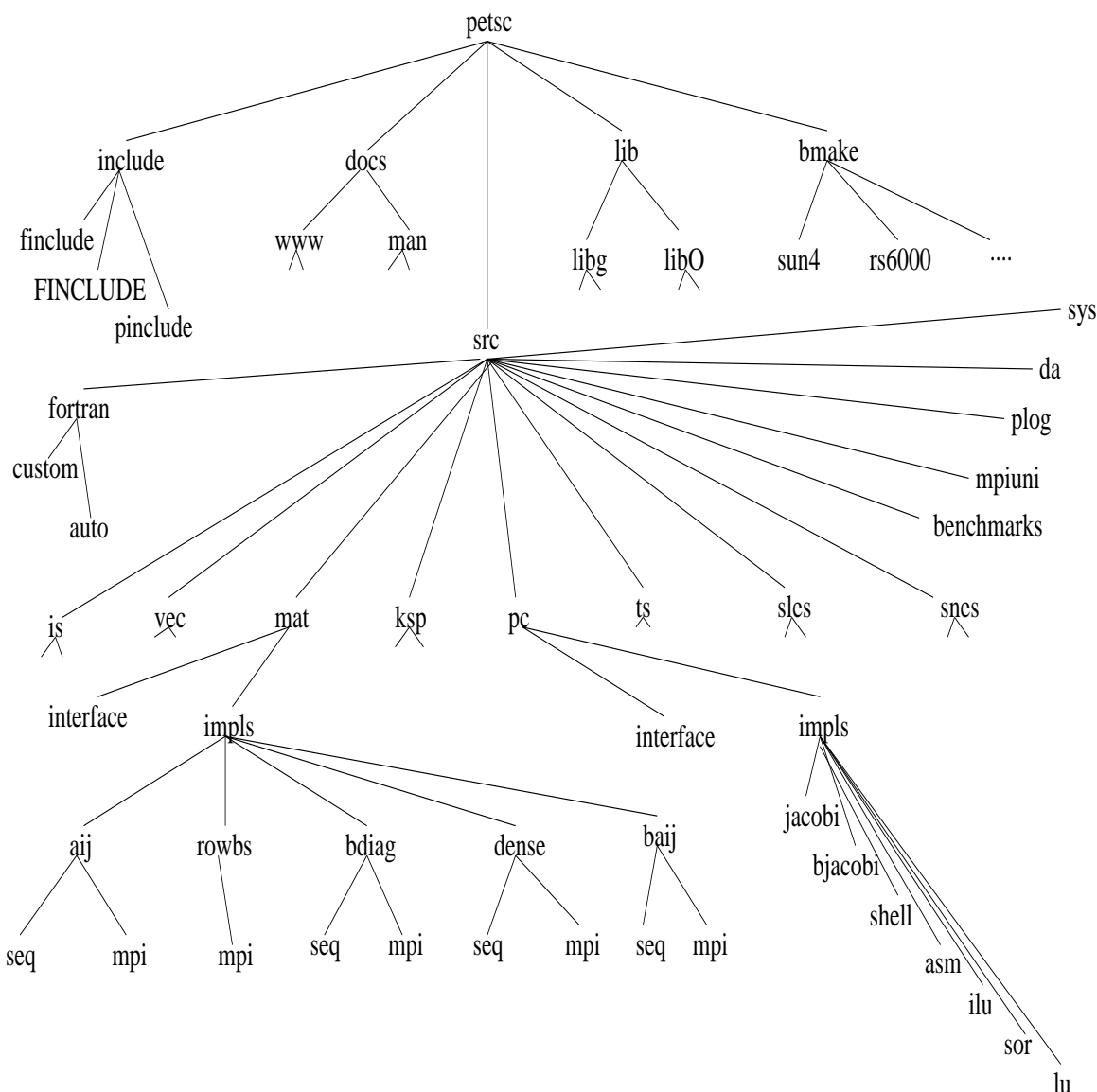


Figure 8: Schematic of the PETSc Directory Structure

Each PETSc source code component directory has the following subdirectories:

- **examples** - Example programs for the component, including
 - **tutorials** - Programs designed to teach users about PETSc. These codes can serve as templates for the design of custom applications.
 - **tests** - Programs designed for thorough testing of PETSc. As such, these codes are not intended for examination by users.
- **interface** - The calling sequences for the abstract interface to the component. Code here does not know about particular implementations.
- **impls** - Source code for one or more implementations.
- **utils** - Utility routines. Source here may know about the implementations, but ideally will not know about implementations for other components.

Part II

Programming with PETSc

Chapter 2

Vectors

The vector (denoted by **Vec**) is one of the simplest PETSc objects. We discuss the basic usage of vectors in Sections 2.1 through 2.3. Sections 2.4 and 2.5 focus on index sets and their use in scatters and gathers. Section 2.6 describes the use of application-defined orderings with vectors, Section 2.7 discusses routines to map indices from a local to the PETSc global numbering scheme, and Section 2.8 discusses the use of PETSc distributed arrays for the manipulation of vectors associated with regular grids. Section 2.9 concludes with a brief discussion of the discrete function component of PETSc.

2.1 Creating and Assembling Vectors

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

```
ierr = VecCreateSeq(MPI_COMM_SELF,int m,Vec *x);
```

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

```
ierr = VecCreateMPI(MPI_Comm comm,int m,int M,Vec *x);
```

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

```
ierr = VecCreate(MPI_Comm comm,int M,Vec *v);
```

which automatically generates the appropriate vector type (sequential or parallel) over all processors in **comm**. The option **-vec_mpi** can be used in conjunction with **VecCreate()** to specify the use of MPI vectors for the uniprocessor case.

We emphasize that all processors in **comm** *must* call the vector creation routines, since these routines are collective over all processors in the communicator. In addition, if a sequence of **VecCreateXXX()** routines is used, they must be called in the same order on each processor in the communicator.

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

```
ierr = VecSet(Scalar *value,Vec x);
```

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

```
ierr = VecSetValues(Vec x,int n,int *indices,Scalar *values,INSERT_VALUES);
```

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

```
ierr = VecAssemblyBegin(Vec x);
```

followed by

```
ierr = 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.

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

```
ierr = VecSetValues(Vec x,int n,int *indices, Scalar *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 processors generated values for the same location, with one processor 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.)

There is no routine called `VecGetValues()`, since we provide an alternative method for extracting some components of a vector using the vector scatter routines. See Section 2.5 for details.

One can examine a vector with the command

```
ierr = VecView(Vec x,Viewer v);
```

To print the vector to the screen, one can use the viewer `VIEWER_STDOUT_WORLD`, which ensures that parallel vectors are printed correctly to `stdout`. The viewer `VIEWER_STDOUT_SELF` can be employed if the user does not care in what order the individual processors print their segments of the vector. To display the vector in an X-window, one can use the default X-windows viewer `VIEWER_DRAWX_WORLD`, or one can create a viewer with the routine `ViewerDrawOpenX()`. A variety of viewers are discussed further in Section 12.2.

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

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

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

```
ierr = VecDuplicateVecs(Vec old,int n,Vec **new);
```

This routine creates an array of pointers to vectors. The two routines are very useful because they allow one to write library code that does not depend on the particular format of the vectors being used. Instead, the subroutines can automatically correctly create work vectors based on the specified existing vector. As discussed in Section 9.1.6, the Fortran interface for `VecDuplicateVecs()` differs slightly.

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

```
ierr = VecDestroy(Vec x);
```

To destroy an array of vectors, one should use the command

```
ierr = VecDestroyVecs(Vec *vecs,int n);
```

Note that the Fortran interface for `VecDestroyVecs()` differs slightly, as described in Section 9.1.6.

2.2 Basic Vector Operations

As listed in Table 1, 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, not because they have some inherent special properties.

Several of these operations could easily be constructed by a combination of the other operations. We implement them directly to avoid a potentially large loss of efficiency, mainly because in such cases the vectors would have to move from main memory to the CPU multiple times, rather than just once. Also, the arithmetic pipeline operations can be used more effectively in the hybrid vector operations.

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

```
ierr = VecGetOwnershipRange(Vec vec,int *low,int *high);
```

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

Table 1: PETSc Vector Operations

Function Name	Operation
<code>VecAXPY(Scalar *a,Vec x, Vec y);</code>	$y = y + a * x$
<code>VecAYPX(Scalar *a,Vec x, Vec y);</code>	$y = x + a * y$
<code>VecWAXPY(Scalar *a,Vec x,Vec y, Vec w);</code>	$w = a * x + y$
<code>VecAXPBY(Scalar *a,Scalar *,Vec x,Vec y);</code>	$y = a * x + b * y$
<code>VecScale(Scalar *a, Vec x);</code>	$x = a * x$
<code>VecDot(Vec x, Vec y, Scalar *r);</code>	$r = \bar{x}' * y$
<code>VecTDot(Vec x, Vec y, Scalar *r);</code>	$r = x' * y$
<code>VecNorm(Vec x, NormType type, double *r);</code>	$r = x _{type}$
<code>VecSum(Vec x, Scalar *r);</code>	$r = \sum x_i$
<code>VecCopy(Vec x, Vec y);</code>	$y = x$
<code>VecSwap(Vec x, Vec y);</code>	$y = x \text{ while } x = y$
<code>VecPointwiseMult(Vec x,Vec y, Vec w);</code>	$w_i = x_i * y_i$
<code>VecPointwiseDivide(Vec x,Vec y, Vec w);</code>	$w_i = x_i / y_i$
<code>VecMDot(int n,Vec x, Vec *y,Scalar *r);</code>	$r[i] = \bar{x}' * y[i]$
<code>VecMTDot(int n,Vec x, Vec *y,Scalar *r);</code>	$r[i] = x' * y[i]$
<code>VecMAXPY(int n, Scalar *a,Vec x, Vec *y);</code>	$y[i] = a_i * x + y[i]$
<code>VecMax(Vec x, int *idx, double *r);</code>	$r = \max x_i$
<code>VecMin(Vec x, int *idx, double *r);</code>	$r = \min x_i$
<code>VecAbs(Vec x);</code>	$x_i = x_i $
<code>VecReciprocal(Vec x);</code>	$x_i = 1/x_i$
<code>VecShift(Scalar *s,Vec x);</code>	$x_i = s + x_i$

2.3 Vector Internals

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

```
ierr = VecGetArray(Vec v,Scalar **array);
```

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

```
ierr = VecRestoreArray(Vec v, Scalar **array);
```

Minor differences exist in the Fortran interface for `VecGetArray()` and `VecRestoreArray()`, as discussed in Section 9.1.3. It is important to note that `VecGetArray()` and `VecRestoreArray()` do *not* copy the vector elements; they merely give users direct access to the vector elements. Thus, these routines require essentially no time to call and can be used efficiently.

The number of elements stored locally can be accessed with

```
ierr = VecGetLocalSize(Vec v,int *size);
```

The global vector length can be determined by

```
ierr = VecGetSize(Vec v,int *size);
```

2.4 Index Sets

To facilitate general vector scatters and gathers, PETSc employs the concept of an index set. An index set, which is a generalization of a set of integer indices, is used to define scatters, gathers, and similar operations on vectors and matrices.

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

```
ierr = ISCreateGeneral(MPI_Comm comm,int n,int *indices, IS *is);
```

This routine essentially copies the `n` indices passed to it by the integer array `indices`. Thus, the user should be sure to free the integer array `indices` when it is no longer needed, perhaps directly after the call to `ISCreateGeneral()`. The communicator, `comm`, should consist of only one processor (often `comm=MPI_COMM_SELF`).

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

```
ierr = ISCreateStride(MPI_Comm comm,int n,int first,int step,IS *is);
```

Again, the `comm` argument should always consist of a communicator with one processor, often `MPI_COMM_SELF`.

There is no parallel general index set. Instead, one should create a sequential index set on each processor that needs one. Index sets can be destroyed with the command

```
ierr = ISDestroy(IS is);
```

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

```
ierr = ISGetSize(IS is,int *size);
```

```
ierr = ISStrideGetInfo(IS is,int *first,int *stride);
```

```
ierr = ISGetIndices(IS is,int **indices);
```

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

```
ierr = ISRestoreIndices(IS is, int **indices);
```

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

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

```
ierr = ISCreateBlock(MPI_Comm comm,int bs,int n,int *indices, 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()`, `ISBlockGetBlockSize()`, and `ISBlock()`. See the man pages for details.

One may reasonably ask why the scatter routines are based on using index sets while the vector assembly routines simply use arrays of integers. The reason for using index sets is to allow certain implementations to deal with strides and dense blocks of elements efficiently. In vector assembly, we are assuming that in the majority of cases the gain from using index sets is more than lost by requiring the users to build index sets for each insertion.

2.5 Scatters and Gathers

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

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

```
ierr = VecScatterCreate(Vec x,IS ix,Vec y,IS iy,VecScatter *ctx);
```

```
ierr = VecScatterBegin(Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD,VecScatter ctx);
```

```
ierr = VecScatterEnd(Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD,VecScatter ctx);
```

```
ierr = 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`, equal 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. 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`.

To perform a conventional gather operation, the user simply makes the destination index set, `iy`, be a stride index set with a stride of one. Similarly, a conventional scatter can be done with an initial (sending) index set consisting of a stride. For parallel vectors, all processors that own the vector *must* call the scatter routines. When scattering from a parallel vector to sequential vectors, each processor 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 processor has its own sequential vector that makes contributions to the parallel vector.

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

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

```

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

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

```

Figure 9: Example Code for Vector Scatters

```

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

```

Note that one must swap the roles of the first two arguments to these routines, whenever one uses the `SCATTER_REVERSE` option.

There is no PETSc routine that is the opposite of `VecSetValues()`, that is, `VecGetValues()`. Instead, the user should create a new vector where the components are to be stored and 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 within Figure 9. In this example, the values of the 100th and 200th components are placed in the array `values`. In this example each processor now has the 100th and 200th component, but obviously each processor could gather any elements it needed, or none by creating an index set with no entries.

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

2.6 Application Orderings

In many applications it is desirable to work with one or more “orderings” of degrees of freedom, cells, nodes, and so on. Doing so in a parallel environment is complicated by the fact that each processor cannot keep complete lists of the mappings between different orderings. In addition, the orderings used in the PETSc linear algebra routines 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 (AO), one can call the routine

```

ierr = AOCreateDebug(MPI_Comm comm,int n,int *apordering,int *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 processor can provide whatever subset of the ordering it chooses, but multiple processors should never contribute duplicate values. The argument `n` indicates the number of local contributed values.

For example, consider a vector of length five, where node 0 in the application ordering corresponds to node 3 in the PETSc ordering. In addition, nodes 1, 2, 3, and 4 of the application ordering correspond, respectively, to nodes 2, 1, 4, and 0 of the PETSc ordering. We can write this correspondence as $\{0, 1, 2, 3, 4\} \rightarrow \{3, 2, 1, 4, 0\}$. The user can create the PETSc-AO mappings in a number of ways. For example, if using two processors, one could call

```
ierr = AOCreatDebug(MPI_COMM_WORLD,2,{0,3},{3,4},&ao);
```

on the first processor and

```
ierr = AOCreatDebug(MPI_COMM_WORLD,3,{1,2,4},{2,1,0},&ao);
```

on the other processor.

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

```
ierr = AOPetscToApplication(AO ao,int n,int *indices);
ierr = AOApplicationToPetsc(AO ao,int n,int *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 processors that called `AOCreatDebug()` also call these routines; these routines *cannot* be called by just a subset of processors.

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

```
ierr = AOCreatDebugIS(MPI_Comm comm,IS apordering,IS petscordering,AO *ao);
```

where index sets are used instead of integer arrays. The corresponding mapping routines are

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

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

2.7 Local to Global Mappings

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

```
ierr = ISLocalToGlobalMappingCreate(int N,int* globalnumbers,ISLocalToGlobalMapping* ctx);
ISLocalToGlobalMappingApply(ISLocalToGlobalMapping ctx,int n,int *in,int *out);
ierr = ISLocalToGlobalMappingApplyIS(ISLocalToGlobalMapping ctx,IS isin,IS* isout);
ierr = ISLocalToGlobalMappingDestroy(ISLocalToGlobalMapping ctx);
```

Here `N` denotes the number of local indices, `globalnumbers` represents 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 the routine `ISLocalToGlobalMappingApply()` or `ISLocalToGlobalMappingApplyIS()`.

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

2.8 Distributed Arrays

In this section we consider a distributed array (DA) to be, not a matrix, but a one-, two-, or three-dimensional array of numbers distributed across the processors, so that each processor contains a rectangular subarray. Distributed arrays, which are used in conjunction with PETSc vectors, are intended for use with *regular rectangular grids* when communication of nonlocal data is needed before certain local computations can

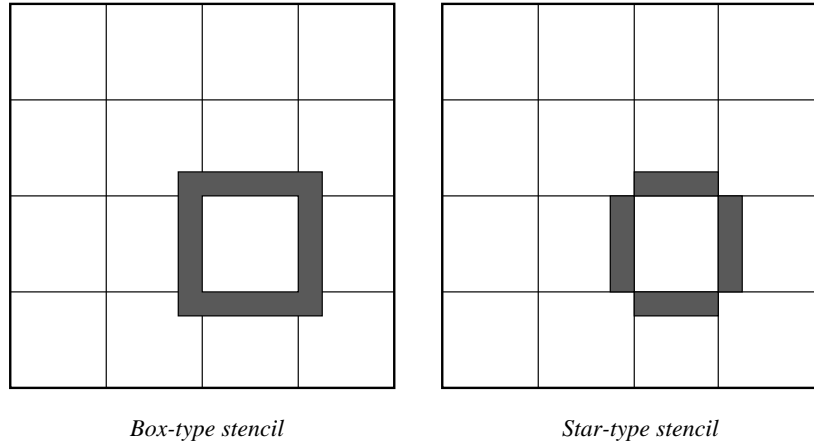


Figure 10: Ghost Points for Two Stencil Types on the Seventh Processor

occur. PETSc distributed arrays are designed only for the case in which data can be thought of as being stored in a standard multidimensional array; thus, **DA**s are *not* intended for parallelizing unstructured grid problems, and the like.

For example, a typical situation one encounters in solving parallel PDEs is that, to evaluate a local function, $f(\mathbf{x})$, each processor requires its local portion of the vector \mathbf{x} as well as its ghost points (the bordering portions of the vector that are owned by neighboring processors). Figure 10 illustrates the ghost points for the seventh processor of a two-dimensional, regular parallel grid. Each box represents a processor; the ghost points for the seventh processor's local part of a parallel array are shown in gray.

2.8.1 Creating Distributed Arrays

One creates a distributed array in two dimensions with the command

```
ierr = DACreate2d(MPI_Comm comm,DAPeriodicType wrap,DASTencilType st,int M,
                  int N,int m,int n,int dof,int s,int *lx,int *ly,DA *da);
```

The arguments **M** and **N** indicate the global numbers of grid points in each direction, while **m** and **n** denote the processor partition in each direction; **m*****n** must equal the number of processors in the MPI communicator, **comm**. Instead of specifying the processor layout, one may use **PETSC_DECIDE** for **m** and **n** so that PETSc will determine the partition. The type of periodicity of the array is specified by **wrap**, which can be **DA_NONPERIODIC** (no periodicity), **DA_XYPERIODIC** (periodic in both x- and y-directions), **DA_XPERIODIC**, or **DA_YPERIODIC**. 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; that is, the dimension of **lx** is **m** and the dimension of **ly** is **n**; or **PETSC_NULL** may be passed in.

Two types of distributed arrays can be created, as specified by **st**. Star-type stencils that radiate outward only in the coordinate directions are indicated by **DA_STENCIL_STAR**, while box-type stencils are specified by **DA_STENCIL_BOX**. For example, for the two-dimensional case, **DA_STENCIL_STAR** with width 1 corresponds to the standard 5-point stencil, while **DA_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, potentially decreasing substantially the number of messages sent. Note that the **DA_STENCIL_STAR** stencils can save interprocessor communication in two and three dimensions.

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

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

```
ierr = DACreate1d(MPI_Comm comm,DAPeriodicType wrap,int M,int w,int s,int *lc,DA *inra);
ierr = DACreate3d(MPI_Comm comm,DAPeriodicType wrap,DASTencilType stencil_type,
                  int M,int N,int P,int m,int n,int p,int w,int s,int *lx,
                  int *ly,int *lz,DA *inra);
```

`DA_ZPERIODIC`, `DA_XZPERIODIC`, `DA_YZPERIODIC`, and `DA_XYZPERIODIC` are additional options in three dimensions for `DAPeriodicType`. The routines to create distributed arrays are collective, so that all processors in the communicator `comm` must call `DACreateXXX()`.

2.8.2 Local/Global Vectors and Scatters

Each `DA` object contains two vectors: a distributed global vector and a local vector that includes the appropriate ghost points. These vectors can be accessed with the routines

```
ierr = DAGetDistributedVector(DA da, Vec *g);
ierr = DAGetLocalVector(DA da, Vec *l);
```

These two vectors will generally serve as the building blocks for local and global PDE solutions, and so on. Note that calling `DAGetDistributedVector()` or `DAGetLocalVector()` does *not* create a new vector object, but rather extracts the one existing vector of its type from the distributed array. Thus, if additional vectors are needed in a code, they can be obtained by duplicating `l` or `g` via `VecDuplicate()` or `VecDuplicateVecs()`.

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

```
ierr = DAGlobalToLocalBegin(DA da, Vec g, InsertMode iora, Vec l);
ierr = DAGlobalToLocalEnd(DA da, Vec g, InsertMode iora, Vec l);
```

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

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

```
ierr = DALocalToGlobal(DA da, Vec l, InsertMode mode, Vec g);
```

Note that this function is not subdivided into beginning and ending phases, since it is purely local.

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

```
ierr = DALocalToLocalBegin(DA da, Vec l1, InsertMode iora, Vec l2);
ierr = DALocalToLocalEnd(DA da, Vec l1, InsertMode iora, Vec l2);
```

Since both local vectors, `l1` and `l2`, must be compatible with the distributed array, `da`, they should be generated by `DAGetLocalVector()` (or be duplicates of such vectors obtained via `VecDuplicate()`). The `InsertMode` can be either `ADD_VALUES` or `INSERT_VALUES`.

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

```
ierr = DAGetScatter(DA da, VecScatter *ltog, VecScatter *gtol, VecScatter *ltol);
```

Most users should not need to use these contexts.

2.8.3 Grid Information

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

```
ierr = DAGetCorners(DA da, int *x, int *y, int *z, int *m, int *n, int *p);
ierr = DAGetGhostCorners(DA da, int *x, int *y, int *z, int *m, int *n, int *p);
```

The first version excludes any ghost points, while the second version includes them. Note that for interior subarrays the ghost corners can easily be calculated from the true corners. However, since the calculation of ghost corners for boundary subarrays is not so straightforward, both routines are provided. The routine `DAGetGhostCorners()` 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, `DA_STENCIL_STAR` or `DA_STENCIL_BOX`, the local vectors (with the ghost points) represent rectangular arrays, including the extra corner elements in the `DA_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 `DA_STENCIL_STAR` is used, the extra corner components are *not* scattered between the processors and thus contain undefined values that should *not* be used.

To assemble global stiffness matrices, one needs to be able to determine the global node number of each local node including the ghost nodes. The number may be determined by using the command

Processor 2	Processor 3	Processor 2	Processor 3
26 27 28	29 30	22 23 24	29 30
21 22 23	24 25	19 20 21	27 28
16 17 18	19 20	16 17 18	25 26
11 12 13	14 15	7 8 9	14 15
6 7 8	9 10	4 5 6	12 13
1 2 3	4 5	1 2 3	10 11
Processor 0	Processor 1	Processor 0	Processor 1
Natural Ordering		PETSc Ordering	

Figure 11: Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processors)

```
ierr = DAGetGlobalIndices(DA da,int *n,int **idx);
```

The output argument `n` contains the number of local nodes, including ghost nodes, while `idx` contains a list of the global indices that correspond to the local nodes. Note that the Fortran interface differs slightly; see Section 9.1.3 for details.

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 `DA` structure provides an application ordering `AO` (see Section 2.6) 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

```
ierr = DAGetAO(DA da,AO *ao);
```

In Figure 11 we indicate the orderings for a two-dimensional distributed array, divided among four processors.

Figure 12, which corresponds to `$(PETSC_DIR)/src/snes/examples/tutorials/ex5.c`, illustrates the use of a distributed array in the solution of a nonlinear problem. The analogous Fortran program is `$(PETSC_DIR)/src/snes/examples/tutorials/ex5f.F`; See Chapter 5 for a discussion of the nonlinear solvers.

```
#ifndef lint
static char vcid[] = "$Id: ex5.c,v 1.75 1997/02/05 22:04:41 bsmith Exp $";
#endif

static char help[] = "Solves a nonlinear system in parallel with SNES.\n\
We solve the Bratu (SFI - solid fuel ignition) problem in a 2D rectangular\n\
domain, using distributed arrays (DAs) to partition the parallel grid.\n\
The command line options include:\n\
  -par <parameter>, where <parameter> indicates the problem's nonlinearity\n\
    problem SFI: <parameter> = Bratu parameter (0 <= par <= 6.81)\n\
  -mx <xg>, where <xg> = number of grid points in the x-direction\n\
  -my <yg>, where <yg> = number of grid points in the y-direction\n\
  -Nx <npx>, where <npx> = number of processors in the x-direction\n\
  -Ny <npy>, where <npy> = number of processors in the y-direction\n\
/*T
  Concepts: SNES`Solving a system of nonlinear equations (parallel Bratu example);
  Concepts: DA`Using distributed arrays;
  Routines: SNESCreate(); SNESSetFunction(); SNESSetJacobian();
  Routines: SNESsolve(); SNESSetFromOptions(); DAView();
  Routines: DACreate2d(); DAdestroy(); DAGetDistributedVector(); DAGetLocalVector();
  Routines: DAGetCorners(); DAGetGhostCorners(); DALocalToGlobal();
  Routines: DAGlobalToLocalBegin(); DAGlobalToLocalEnd(); DAGetGlobalIndices();
```

```

Processors: n
T*/

/* -----

Solid Fuel Ignition (SFI) problem. This problem is modeled by
the partial differential equation


$$-\text{Laplacian } u - \lambda \exp(u) = 0, \quad 0 < x, y < 1,$$


with boundary conditions


$$u = 0 \quad \text{for } x = 0, x = 1, y = 0, y = 1.$$


A finite difference approximation with the usual 5-point stencil
is used to discretize the boundary value problem to obtain a nonlinear
system of equations.

The uniprocessor version of this code is snes/examples/tutorials/ex4.c
----- */

/*
Include "da.h" so that we can use distributed arrays (DAs).
Include "snes.h" so that we can use SNES solvers. Note that this
file automatically includes:
    petsc.h - base PETSc routines    vec.h - vectors
    sys.h   - system routines        mat.h - matrices
    is.h    - index sets             ksp.h - Krylov subspace methods
    viewer.h - viewers               pc.h  - preconditioners
    sles.h  - linear solvers
*/
#include "da.h"
#include "snes.h"
#include <math.h>
#include <stdio.h>

/*
User-defined application context - contains data needed by the
application-provided call-back routines, FormJacobian() and
FormFunction().
*/
typedef struct {
    double    param;                /* test problem parameter */
    int       mx,my;                /* discretization in x, y directions */
    Vec       localX, localF;        /* ghosted local vector */
    DA        da;                   /* distributed array data structure */
    int       rank;                 /* processor rank */
} AppCtx;

/*
User-defined routines
*/
int FormFunction(SNES,Vec,Vec,void*), FormInitialGuess(AppCtx*,Vec);
int FormJacobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);

int main( int argc, char **argv )
{
    SNES      snes;                  /* nonlinear solver */
    Vec       x, r;                  /* solution, residual vectors */

```

```

Mat      J;                      /* Jacobian matrix */
AppCtx   user;                  /* user-defined work context */
int       its;                  /* iterations for convergence */
int       Nx, Ny;              /* number of precessors in x- and y- directions */
int       matrix_free;         /* flag - 1 indicates matrix-free version */
int       size;                /* number of processors */
int       m, flg, N, ierr;
double    bratu_lambda_max = 6.81, bratu_lambda_min = 0.;

PetscInitialize( &argc, &argv, (char *)0, help );
MPI_Comm_rank(MPI_COMM_WORLD, &user.rank);

/*
   Initialize problem parameters
*/
user.mx = 4; user.my = 4; user.param = 6.0;
ierr = OptionsGetInt(PETSC_NULL, "-mx", &user.mx, &flg); CHKERRA(ierr);
ierr = OptionsGetInt(PETSC_NULL, "-my", &user.my, &flg); CHKERRA(ierr);
ierr = OptionsGetDouble(PETSC_NULL, "-par", &user.param, &flg); CHKERRA(ierr);
if (user.param >= bratu_lambda_max || user.param <= bratu_lambda_min) {
    SETERRA(1,0,"Lambda is out of range");
}
N = user.mx*user.my;

/* - - - - -
   Create nonlinear solver context
   - - - - - */

ierr = SNESCreate(MPI_COMM_WORLD, SNES_NONLINEAR_EQUATIONS, &snes); CHKERRA(ierr);

/* - - - - -
   Create vector data structures; set function evaluation routine
   - - - - - */

/*
   Create distributed array (DA) to manage parallel grid and vectors
*/
MPI_Comm_size(MPI_COMM_WORLD, &size);
Nx = PETSC_DECIDE; Ny = PETSC_DECIDE;
ierr = OptionsGetInt(PETSC_NULL, "-Nx", &Nx, &flg); CHKERRA(ierr);
ierr = OptionsGetInt(PETSC_NULL, "-Ny", &Ny, &flg); CHKERRA(ierr);
if (Nx*Ny != size && (Nx != PETSC_DECIDE || Ny != PETSC_DECIDE))
    SETERRA(1,0,"Incompatible number of processors: Nx * Ny != size");
ierr = DACreate2d(MPI_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR, user.mx,
    user.my, Nx, Ny, 1, 1, PETSC_NULL, PETSC_NULL, &user.da); CHKERRA(ierr);

/*
   Visualize the distribution of the array across the processors
*/
/* ierr = DAView(user.da, VIEWER_DRAWX_WORLD); CHKERRA(ierr); */

/*
   Extract global and local vectors from DA; then duplicate for remaining
   vectors that are the same types
*/
ierr = DAGetDistributedVector(user.da, &x); CHKERRA(ierr);
ierr = DAGetLocalVector(user.da, &user.localX); CHKERRA(ierr);
ierr = VecDuplicate(x, &r); CHKERRA(ierr);
ierr = VecDuplicate(user.localX, &user.localF); CHKERRA(ierr);

```

```

/*
    Set function evaluation routine and vector
*/
ierr = SNESSetFunction(snes,r,FormFunction,(void*)&user); CHKERRA(ierr);

/* -----
    Create matrix data structure; set Jacobian evaluation routine
    ----- */

/*
    Set Jacobian matrix data structure and default Jacobian evaluation
    routine. User can override with:
    -snes_fd : default finite differencing approximation of Jacobian
    -snes_mf : matrix-free Newton-Krylov method with no preconditioning
                (unless user explicitly sets preconditioner)
    -snes_mf_operator : form preconditioning matrix as set by the user,
                        but use matrix-free approx for Jacobian-vector
                        products within Newton-Krylov method

    Note: For the parallel case, vectors and matrices MUST be partitioned
    accordingly. When using distributed arrays (DAs) to create vectors,
    the DAs determine the problem partitioning. We must explicitly
    specify the local matrix dimensions upon its creation for compatibility
    with the vector distribution. Thus, the generic MatCreate() routine
    is NOT sufficient when working with distributed arrays.

    Note: Here we only approximately preallocate storage space for the
    Jacobian. See the users manual for a discussion of better techniques
    for preallocating matrix memory.
*/
ierr = OptionsHasName(PETSC_NULL,"-snes_mf",&matrix_free); CHKERRA(ierr);
if (!matrix_free) {
    if (size == 1) {
        ierr = MatCreateSeqAIJ(MPI_COMM_WORLD,N,N,5,PETSC_NULL,&J); CHKERRA(ierr);
    } else {
        ierr = VecGetLocalSize(x,&m); CHKERRA(ierr);
        ierr = MatCreateMPIAIJ(MPI_COMM_WORLD,m,m,N,N,5,PETSC_NULL,3,PETSC_NULL,&J); CHKERRA(ierr);
    }
    ierr = SNESSetJacobian(snes,J,J,FormJacobian,&user); CHKERRA(ierr);
}

/* -----
    Customize nonlinear solver; set runtime options
    ----- */

/*
    Set runtime options (e.g., -snes_monitor -snes_rtol <rtol> -ksp_type <type>)
*/
ierr = SNESSetFromOptions(snes); CHKERRA(ierr);

/* -----
    Evaluate initial guess; then solve nonlinear system
    ----- */

/*
    Note: The user should initialize the vector, x, with the initial guess
    for the nonlinear solver prior to calling SNESolve(). In particular,
    to employ an initial guess of zero, the user should explicitly set
    this vector to zero by calling VecSet().
*/

```

```

ierr = FormInitialGuess(&user,x); CHKERRA(ierr);
ierr = SNESolve(snes,x,&its); CHKERRA(ierr);
PetscPrintf(MPI_COMM_WORLD,"Number of Newton iterations = %d\n", its );

/* - - - - -
   Free work space. All PETSc objects should be destroyed when they
   are no longer needed.
   - - - - - */

if (!matrix_free) {
    ierr = MatDestroy(J); CHKERRA(ierr);
}
ierr = VecDestroy(user.localX); CHKERRA(ierr); ierr = VecDestroy(x); CHKERRA(ierr);
ierr = VecDestroy(user.localF); CHKERRA(ierr); ierr = VecDestroy(r); CHKERRA(ierr);
ierr = SNESDestroy(snes); CHKERRA(ierr); ierr = DADestroy(user.da); CHKERRA(ierr);
PetscFinalize();

return 0;
}
/* ----- */
/*
   FormInitialGuess - Forms initial approximation.

   Input Parameters:
   user - user-defined application context
   X - vector

   Output Parameter:
   X - vector
*/
int FormInitialGuess(AppCtx *user,Vec X)
{
    int    i, j, row, mx, my, ierr, xs, ys, xm, ym, gxm, gym, gxs, gys;
    double one = 1.0, lambda, temp1, temp, hx, hy, hxdhy, hydhx,sc;
    Scalar *x;
    Vec    localX = user->localX;

    mx = user->mx;          my = user->my;          lambda = user->param;
    hx = one/(double)(mx-1); hy = one/(double)(my-1);
    sc = hx*hy*lambda;      hxdhy = hx/hy;          hydhx = hy/hx;
    temp1 = lambda/(lambda + one);

    /*
       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.
    */
    ierr = VecGetArray(localX,&x); CHKERRQ(ierr);

    /*
       Get local grid boundaries (for 2-dimensional DA):
       xs, ys - starting grid indices (no ghost points)
       xm, ym - widths of local grid (no ghost points)
       gxs, gys - starting grid indices (including ghost points)
       gxm, gym - widths of local grid (including ghost points)
    */
    ierr = DAGetCorners(user->da,&xs,&ys,PETSC_NULL,&xm,&ym,PETSC_NULL); CHKERRQ(ierr);
    ierr = DAGetGhostCorners(user->da,&gxs,&gys,PETSC_NULL,&gxm,&gym,PETSC_NULL); CHKERRQ(ierr);

```

```

/*
    Compute initial guess over the locally owned part of the grid
*/
for (j=ys; j<ys+ym; j++) {
    temp = (double)(PetscMin(j,my-j-1))*hy;
    for (i=xs; i<xs+xm; i++) {
        row = i - gxs + (j - gys)*gxm;
        if (i == 0 || j == 0 || i == mx-1 || j == my-1 ) {
            x[row] = 0.0;
            continue;
        }
        x[row] = temp1*sqrt( PetscMin( (double)(PetscMin(i,mx-i-1))*hx,temp) );
    }
}

/*
    Restore vector
*/
ierr = VecRestoreArray(localX,&x); CHKERRQ(ierr);

/*
    Insert values into global vector
*/
ierr = DALocalToGlobal(user->da,localX,INSERT_VALUES,X); CHKERRQ(ierr);
return 0;
}
/* ----- */
/*
    FormFunction - Evaluates nonlinear function, F(x).

    Input Parameters:
.   snes - the SNES context
.   X - input vector
.   ptr - optional user-defined context, as set by SNESSetFunction()

    Output Parameter:
.   F - function vector
*/
int FormFunction(SNES snes,Vec X,Vec F,void *ptr)
{
    AppCtx *user = (AppCtx *) ptr;
    int      ierr, i, j, row, mx, my, xs, ys, xm, ym, gxs, gys, gx, gym;
    double   two = 2.0, one = 1.0, lambda,hx, hy, hxdhy, hydhx,sc;
    Scalar   u, uxx, uyy, *x,*f;
    Vec      localX = user->localX, localF = user->localF;

    mx = user->mx;          my = user->my;          lambda = user->param;
    hx = one/(double)(mx-1); hy = one/(double)(my-1);
    sc = hx*hy*lambda;      hxdhy = hx/hy;          hydhx = hy/hx;

/*
    Scatter ghost points to local vector, using the 2-step process
    DAGlobalToLocalBegin(), DAGlobalToLocalEnd().
    By placing code between these two statements, computations can be
    done while messages are in transition.
*/
ierr = DAGlobalToLocalBegin(user->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);
ierr = DAGlobalToLocalEnd(user->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);

```



```

/*
    Get pointers to vector data
*/
ierr = VecGetArray(localX,&x); CHKERRQ(ierr);
ierr = VecGetArray(localF,&f); CHKERRQ(ierr);

/*
    Get local grid boundaries
*/
ierr = DAGetCorners(user->da,&xs,&ys,PETSC_NULL,&xm,&ym,PETSC_NULL); CHKERRQ(ierr);
ierr = DAGetGhostCorners(user->da,&gxs,&gys,PETSC_NULL,&gxsm,&gysm,PETSC_NULL); CHKERRQ(ierr);

/*
    Compute function over the locally owned part of the grid
*/
for (j=ys; j<ys+ym; j++) {
    row = (j - gys)*gxsm + xs - gxs - 1;
    for (i=xs; i<xs+xm; i++) {
        row++;
        if (i == 0 || j == 0 || i == mx-1 || j == my-1 ) {
            f[row] = x[row];
            continue;
        }
        u = x[row];
        uxx = (two*u - x[row-1] - x[row+1])*hydhx;
        uyy = (two*u - x[row-gxsm] - x[row+gxsm])*hxdhy;
        f[row] = uxx + uyy - sc*exp(u);
    }
}

/*
    Restore vectors
*/
ierr = VecRestoreArray(localX,&x); CHKERRQ(ierr);
ierr = VecRestoreArray(localF,&f); CHKERRQ(ierr);

/*
    Insert values into global vector
*/
ierr = DALocalToGlobal(user->da,localF,INSERT_VALUES,F); CHKERRQ(ierr);
PLogFlops(11*ym*xm);
return 0;
}
/* ----- */
/*
    FormJacobian - Evaluates Jacobian matrix.

    Input Parameters:
.   snes - the SNES context
.   x - input vector
.   ptr - optional user-defined context, as set by SNESSetJacobian()

    Output Parameters:
.   A - Jacobian matrix
.   B - optionally different preconditioning matrix
.   flag - flag indicating matrix structure

    Notes:
    Due to grid point reordering with DAs, we must always work
    with the local grid points, and then transform them to the new

```

```

    global numbering with the "ltog" mapping (via DAGetGlobalIndices()).
    We cannot work directly with the global numbers for the original
    uniprocessor grid!
*/
int FormJacobian(SNES snes, Vec X, Mat *J, Mat *B, MatStructure *flag, void *ptr)
{
    AppCtx *user = (AppCtx *) ptr; /* user-defined application context */
    Mat      jac = *J;               /* Jacobian matrix */
    Vec      localX = user->localX; /* local vector */
    int      *ltog;                  /* local-to-global mapping */
    int      ierr, i, j, row, mx, my, col[5];
    int      nloc, xs, ys, xm, ym, gxs, gys, gxm, gym, grow;
    Scalar    two = 2.0, one = 1.0, lambda, v[5], hx, hy, hxdhy, hydhx, sc, *x;

    mx = user->mx;          my = user->my;          lambda = user->param;
    hx = one/(double)(mx-1); hy = one/(double)(my-1);
    sc = hx*hy;             hxdhy = hx/hy;          hydhx = hy/hx;

    /*
       Scatter ghost points to local vector, using the 2-step process
       DAGlobalToLocalBegin(), DAGlobalToLocalEnd().
       By placing code between these two statements, computations can be
       done while messages are in transition.
    */
    ierr = DAGlobalToLocalBegin(user->da, X, INSERT_VALUES, localX); CHKERRQ(ierr);
    ierr = DAGlobalToLocalEnd(user->da, X, INSERT_VALUES, localX); CHKERRQ(ierr);

    /*
       Get pointer to vector data
    */
    ierr = VecGetArray(localX, &x); CHKERRQ(ierr);

    /*
       Get local grid boundaries
    */
    ierr = DAGetCorners(user->da, &xs, &ys, PETSC_NULL, &xm, &ym, PETSC_NULL); CHKERRQ(ierr);
    ierr = DAGetGhostCorners(user->da, &gxs, &gys, PETSC_NULL, &gxm, &gym, PETSC_NULL); CHKERRQ(ierr);

    /*
       Get the global node numbers for all local nodes, including ghost points
    */
    ierr = DAGetGlobalIndices(user->da, &nloc, &ltog); CHKERRQ(ierr);

    /*
       Compute entries for the locally owned part of the Jacobian.
       - Currently, all PETSc parallel matrix formats are partitioned by
       contiguous chunks of rows across the processors. The "grow"
       parameter computed below specifies the global row number
       corresponding to each local grid point.
       - 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 row and columns of matrix entries.
       - Here, we set all entries for a particular row at once.
    */
    for (j=ys; j<ys+ym; j++) {
        row = (j - gys)*gxm + xs - gxs - 1;
        for (i=xs; i<xs+xm; i++) {
            row++;
            grow = ltog[row];

```

```

/* boundary points */
if ( i == 0 || j == 0 || i == mx-1 || j == my-1 ) {
    ierr = MatSetValues(jac,1,&grow,1,&grow,&one,INSERT_VALUES); CHKERRQ(ierr);
    continue;
}
/* interior grid points */
v[0] = -hxdhy; col[0] = ltog[row - gxm];
v[1] = -hydhx; col[1] = ltog[row - 1];
v[2] = two*(hydhx + hxdhy) - sc*lambda*exp(x[row]); col[2] = grow;
v[3] = -hydhx; col[3] = ltog[row + 1];
v[4] = -hxdhy; col[4] = ltog[row + gxm];
ierr = MatSetValues(jac,1,&grow,5,col,v,INSERT_VALUES); CHKERRQ(ierr);
}
}

/*
Assemble matrix, using the 2-step process:
    MatAssemblyBegin(), MatAssemblyEnd().
By placing code between these two statements, computations can be
done while messages are in transition.
*/
ierr = MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = VecRestoreArray(localX,&x); CHKERRQ(ierr);
ierr = MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

/*
Set flag to indicate that the Jacobian matrix retains an identical
nonzero structure throughout all nonlinear iterations (although the
values of the entries change). Thus, we can save some work in setting
up the preconditioner (e.g., no need to redo symbolic factorization for
ILU/ICC preconditioners).
- If the nonzero structure of the matrix is different during
successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
must be used instead. If you are unsure whether the matrix
structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
- Caution: If you specify SAME_NONZERO_PATTERN, PETSc
believes your assertion and does not check the structure
of the matrix. If you erroneously claim that the structure
is the same when it actually is not, the new preconditioner
will not function correctly. Thus, use this optimization
feature with caution!
*/
*flag = SAME_NONZERO_PATTERN;
return 0;
}

/*
Demonstrates how you can restrict the linking in of solvers, etc
to those that you KNOW you are going to use. This decreases the size
of your executable and decreases the time it takes to link your program.
*/
/* ----- Note currently these are commented out -----
extern int SNESCreate_EQ_LS(SNES);
int SNESRegisterAll()
{
    SNESRegisterAllCalled = 1;

    SNESRegister(SNES_EQ_LS,          0,"ls",          SNESCreate_EQ_LS);
    return 0;
}

```

```

extern int KSPCreate_GMRES(KSP);
int KSPRegisterAll()
{
    KSPRegisterAllCalled = 1;

    KSPRegister(KSPGMRES, 0, "gmres", KSPCreate_GMRES);
    return 0;
}

extern int PCCreate_BJacobi(PC);
extern int PCCreate_ILU(PC);
int PCRegisterAll()
{
    PCRegisterAllCalled = 1;

    PCRegister(PCBJACOBI, 0, "bjacobi", PCCreate_BJacobi);
    PCRegister(PCILU, 0, "ilu", PCCreate_ILU);
    return 0;
}

extern int MatLoad_SeqAIJ(Viewer, MatType, Mat*);
extern int MatLoad_MPIAIJ(Viewer, MatType, Mat*);
int MatLoadRegisterAll()
{
    int ierr;

    ierr = MatLoadRegister(MATSEQAIJ, MatLoad_SeqAIJ); CHKERRQ(ierr);
    ierr = MatLoadRegister(MATMPIAIJ, MatLoad_MPIAIJ); CHKERRQ(ierr);
    return 0;
}

int MatConvertRegisterAll()
{
    return 0;
}

extern int MatOrder_Natural(Mat, MatReordering, IS*, IS*);
int MatReorderingRegisterAll()
{
    int ierr;

    MatReorderingRegisterAllCalled = 1;
    ierr = MatReorderingRegister(ORDER_NATURAL, 0, "natural", MatOrder_Natural); CHKERRQ(ierr);
    return 0;
}
-----*/

```

Figure 12: Use of Distributed Arrays

2.9 Discrete Functions

The discrete functions component of PETSc will be replaced in a future PETSc release.

In mathematical terms, vectors are linear algebraic objects that do not incorporate any concept of geometry, grid, or ordering. But often users want to work with numerical values (stored as a vector or matrix) in relation to where they lie on a grid or graph. PETSc thus provides an enhanced vector object, known as

a *discrete function vector* or **DFVec**, that is, a vector of values and its associated geometry or grid. Thus, any vector action that requires the geometric information (e.g., contour plotting) can be performed *only* on discrete function vectors and not on basic vectors.

The vectors created by PETSc distributed arrays (as discussed in Section 2.8) automatically are **DFVec**'s, as they are associated with a regular grid in one, two, or three dimensions. Thus, any vector obtained from **DAGetDistributedVector()** or **DAGetLocalVector()** (or copied from such vectors via **VecDuplicate()** or **VecDuplicateVecs()**) can immediately be used with the **DFVec** routines.

In addition to all of the conventional vector operations, the support for discrete function vectors includes viewing, contour plotting, interpolating, and extracting various components for problems with multiple degrees of freedom for each node of a grid. One useful routine for viewing discrete function vectors is

```
ierr = DFVecView(DFVec dfv, Viewer viewer);
```

which views the components in the same ordering that would be used for the single processor case, independent of the number of processors, layout, and so forth. This routine supports all of the standard vector viewers; the only difference is that in the parallel case, the display *always* employs the standard uniprocessor ordering. **DFVecView()** is particularly useful for checking correctness of parallel vector computations for different processor configurations and for restarting calculations on different numbers of processors. One routine for visualization of vector fields is

```
ierr = DFVecDrawTensorContoursX(DFVec dfv, int width, int height);
```

which draws in an X-window a contour plot for each component within a multicomponent vector. Another routine for viewing vector fields is

```
ierr = DFVecDrawTensorSurfaceContoursVRML(DFVec dfv);
```

which generates surface contour data view use with a VRML reader. See the man page for details.

Chapter 3

Matrices

PETSc 2.0 provides a variety of matrix implementations because no single matrix format is appropriate for all problems. Currently we support dense storage and compressed sparse row storage (both sequential and parallel versions), as well as several specialized formats. Additional formats can be added easily.

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 uniprocessor and parallel matrix types. Details regarding the ever-expanding suite of PETSc matrices are given in Section 15.5. 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.

3.1 Creating and Assembling Matrices

The simplest routine for forming a PETSc matrix, **A**, is

```
ierr = MatCreate(MPI_Comm comm,int M,int N,Mat *A)
```

This routine generates a sequential matrix when running on one processor and a parallel matrix for two or more processors; the particular matrix format is set by the user via options database commands. The user specifies only the global matrix dimensions, given by **M** and **N**, while PETSc determines the appropriate local dimensions and 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 Section 3.1.1. See the man page for further information about available matrix formats.

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

```
ierr = MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar *values,INSERT_VALUES);
```

or

```
ierr = MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar *values,ADD_VALUES);
```

This routine inserts or adds a logically dense subblock of dimension **m*n** into the matrix. The integer indices **im** and **in**, respectively, indicate the global row and column numbers to be inserted. **MatSetValues()** uses the standard C convention, where the row and column matrix indices begin with zero *regardless of the storage format employed*. The array **values** is logically two dimensional, containing the values that are to be inserted. By default the values are given in row major order, which is the opposite of the Fortran 77 convention. To allow the insertion of values in column major order, one can call the command

```
ierr = MatSetOption(Mat A,MAT_COLUMN_ORIENTED);
```

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()**.

The function **MatSetOption()** accepts several other inputs. We discuss two of these, which are related to the efficiency of the assembly process. To indicate to PETSc that the row (**im**) or column (**in**) indices set with **MatSetValues()** are sorted, one uses the command

```
ierr = MatSetOption(Mat A,MAT_ROWS_SORTED);
```

or

```
ierr = MatSetOption(Mat A,MAT_COLUMNS_SORTED);
```

After the matrix elements have been inserted or added into the matrix, it must be processed before it can be used. The routines for matrix processing are

```
ierr = MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
```

```
ierr = MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

By placing other code between these two calls, the user can perform computations while messages are in transition. 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 processor eventually stores them, for efficiency reasons we usually recommend generating most entries on the processor where they are destined to be stored. To help the application programmer with this task for matrices that are distributed across the processors by ranges, the routine

```
ierr = MatGetOwnershipRange(Mat A,int *first_row,int *last_row);
```

informs the user that all rows from `first_row` to `last_row-1` will be stored on the local processor.

In the sparse matrix implementations, once the assembly routines have been called, the matrices are compressed and can be used for matrix-vector multiplication, and so on. Inserting new values into the matrix at this point will be expensive, since it requires copies and possible memory allocation. Thus, whenever possible one should completely set the values in the matrices before calling the final assembly routines.

If one wishes to repeatedly assemble matrices that retain the same nonzero pattern (such as within a nonlinear or time-dependent problem), the option

```
ierr = MatSetOption(Mat mat,MAT_NO_NEW_NONZERO_LOCATIONS);
```

should be specified after the first matrix has been fully assembled. This option ensures that certain data structures and communication information will be reused (instead of regenerated) during successive steps, thereby increasing efficiency. See `$(PETSC_DIR)/src/sles/examples/tutorials/ex5.c` for a simple example of solving two linear systems that use the same matrix data structure.

3.1.1 Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the Yale sparse matrix format or compressed sparse row format, CSR). This section discusses tips for *efficiently* using this matrix format for large-scale applications. Additional formats (such as block compressed row and block diagonal storage, which are generally much more efficient for problems with multiple degrees of freedom per node) are further discussed in Section 15.5. Beginning users need not concern themselves initially with such details and may wish to proceed directly to Section 3.2. 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

```
ierr = MatCreateSeqAIJ(MPI_COMM_SELF,int m,int n,int nz,int *nzz,Mat *A);
```

where `nz` or `nnz` can be used to preallocate matrix memory, as discussed below. The user can set `nz=0` and `nzz=PETSC_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_aij_no_inode` (do not use inodes) and `-mat_aij_inode_limit <limit>` (set inode limit (max limit=5)).

By default the internal data representation for the AIJ formats employs zero-based indexing. For compatibility with standard Fortran 77 storage, thus enabling use of external Fortran software packages such as SPARSKIT, the option `-mat_aij_oneindex` enables one-based indexing, where the stored row and column indices begin at one, not zero. All user calls to PETSc routines, regardless of this option, use 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 `nnz` 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,

Thus, 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,

```
int 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 mallocs and moves of data that are needed for dynamic allocation during matrix assembly.

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

- Allocate integer array `nnz`.
- Loop over grid, counting the expected number of nonzeros for the row(s) associated with the various grid points.
- Create the sparse matrix via `MatCreateSeqAIJ()` or alternative.
- 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:

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

The `-log_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 100 X 100; storage space: 2000 unneeded, 1000 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 0
```


The first line indicates that the user preallocated 3000 spaces but only 1000 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 1000 X 1000; storage space: 47 unneeded, 100000 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

```
ierr = MatCreateMPIAIJ(MPI_Comm comm,int m,int n,int M,int N,int d_nz,
                        int *d_nnz, int o_nz,int *o_nnz,Mat *A);
```

A is the newly created matrix, while 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 processor, given by **m**, or determined by PETSc if **m** is **PETSC_DECIDE**.

If one does not use **PETSC_DECIDE** for **m** and **n**, then one must ensure that they are chosen to be compatible with the vectors. To do this, one first considers the product $y = Ax$. The **m** that one uses in **MatCreateMPIAIJ()** must match the local size used in the **VecCreateMPI()** for **y**. 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=PETSC_NULL**, and **o_nnz=PETSC_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 processor's diagonal portion to be its local rows and the corresponding columns (a square submatrix); each processor'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 **MatCreateMPIAIJ** 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 processors 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 processor, 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 processor is given by

$$\left(\begin{array}{ccc} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{array} \right),$$

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 processor one could set **d_nz** to 2 (since each row has 2 nonzeros) or, alternatively, set **d_nzz** to {2,2,2}. The **o_nz** could be set to 2 (since each row of the **o** matrix has 2 nonzeros), or **o_nzz** could be set to {2,2,2}.

For the second processor 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_nzz** 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 processor 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_nzz** 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 **-log_info** will print information about the success of preallocation during matrix assembly. For the **MATMPIAIJ** format, PETSc will also list the number of elements owned by on each processor that were generated on a different processor. For example, the statements

```
[0]MatAssemblyBegin_MPIAIJ: Number of off processor values 10
```

```
[1]MatAssemblyBegin_MPIAIJ: Number of off processor values 7
```

```
[2]MatAssemblyBegin_MPIAIJ: Number of off processor values 5
```

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

```
[0]MatAssemblyBegin_MPIAIJ: Number of off processor values 100000
```

```
[1]MatAssemblyBegin_MPIAIJ: Number of off processor values 77777
```

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

Note: It is fine to generate some entries on the “wrong” processor. 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.

3.1.2 Dense Matrices

PETSc provides both sequential and parallel dense matrix formats, where each processor stores its entries in a column-major array in the usual Fortran 77 style. To create a sequential, dense PETSc matrix, **A** of dimensions **m** by **n**, the user should call

```
ierr = MatCreateSeqDense(MPI_COMM_SELF,int m,int n,Scalar *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 **PETSC_NULL** for PETSc to control matrix memory allocation. To create a parallel, dense matrix, **A**, the user should call

```
ierr = MatCreateMPIDense(MPI_Comm comm,int m,int n,int M,int N,Scalar *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 processor, given by **m**, or determined by PETSc if **m** is **PETSC_DECIDE**.

PETSc does not currently provide parallel dense direct solvers. Our focus is on sparse iterative solvers.

3.2 Basic Matrix Operations

Table 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

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

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

```
ierr = VecCreateMPI(MPI_Comm comm,n,N,&x);
ierr = 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,

```
ierr = MatMultTrans(Mat A,Vec x,Vec y);
```

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

```
ierr = MatMultAdd(Mat A,Vec x,Vec y,Vec w);
ierr = MatMultTransAdd(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 77, 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

```
ierr = MatView(Mat mat,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

```
ierr = MatView(Mat mat,VIEWER_DRAWX_WORLD);
```

or

```
ierr = MatView(Mat mat,Viewer viewer);
```

where **viewer** was obtained with `ViewerDrawOpenX()`. Additional viewers and options are given in the `MatView()` man page and Section 12.2.

Table 2: PETSc Matrix Operations

Function Name	Operation
<code>MatAXPY(Scalar *a,Vec X, Vec Y);</code>	$Y = Y + a * X$
<code>MatMult(Mat A,Vec x, Vec y);</code>	$y = A * x$
<code>MatMultAdd(Mat A,Vec x, Vec y,Vec z);</code>	$z = y + A * x$
<code>MatMultTrans(Mat A,Vec x, Vec y);</code>	$y = A^T * x$
<code>MatMultTransAdd(Mat A,Vec x, Vec y,Vec z);</code>	$z = y + A^T * x$
<code>MatNorm(Mat A, NormType type, double *r);</code>	$r = A _{type}$
<code>MatDiagonalScale(Mat A,Vec l,Vec r);</code>	$A = \text{diag}(l) * A * \text{diag}(r)$
<code>MatScale(Scalar *a,Mat A);</code>	$A = a * A$
<code>MatConvert(Mat A,MatType type,Mat *B);</code>	$B = A$
<code>MatCopy(Mat A,Mat B);</code>	$B = A$
<code>MatGetDiagonal(Mat A,Vec x);</code>	$x = \text{diag}(A)$
<code>MatTranspose(Mat A,Mat* B);</code>	$B = A^T$
<code>MatZeroEntries(Mat A);</code>	$A = 0$

3.3 Matrix-Free Matrices

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

```
ierr = MatCreateShell(MPI_Comm comm,int m,int n,int M,int N,void *ctx,Mat *mat);
```

Here `M` and `N` are the global matrix dimensions (rows and columns), `m` and `n` are the local matrix dimensions, and `ctx` is a pointer to data needed by any user-defined shell matrix operations; the man 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

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

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

```
ierr = MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,  
                           int (*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 shell matrix operations as well. The file `$(PETSC_DIR)/include/mat.h` 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()` \rightarrow `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 interface (for example, `MatMult(Mat,Vec,Vec) \rightarrow UserMult(Mat,Vec,Vec)`).

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.

3.4 Other Matrix Operations

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

```
ierr = MatZeroEntries(Mat A);
```

For sparse matrices 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()`.

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

```
ierr = MatZeroRows(Mat A,IS rows,Scalar *diag_value);
```

For sparse matrices this removes the data structures for certain rows of the matrix. If the pointer `diag_value` is `PETSC_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.

Another matrix routine of interest is

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

which converts the matrix `mat` to new matrix, `M`, that has either the same or different format. The user should set `newtype` to `MATSAME` to copy the matrix, keeping the same matrix format. See `$(PETSC_DIR)/include/mat.h` for other available matrix types.

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

```
ierr = MatGetRow(Mat A, int row, int *ncols, int **cols, Scalar **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 `PETSC_NULL` for the `vals` argument. Similarly, one can use `PETSC_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

```
ierr = MatRestoreRow(Mat A, int row, int *ncols, int **cols, Scalar **vals);
```

to free any space that was allocated during the call to `MatGetRow()`. The reason for the `MatRestoreRow()` command is that most of the sparse matrix storage formats require `MatGetRow()` to allocate some space for reorganizing matrix data before presenting it to the user.

Chapter 4

SLES: Linear Equations Solvers

SLES is the heart of PETSc, because it provides uniform and efficient access to all of the package's linear system solvers, both parallel and sequential, direct and iterative. SLES is intended for solving nonsingular systems of the form

$$Ax = b, \tag{4.1}$$

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

The combination of a Krylov subspace method and a preconditioner is at the center of most modern numerical codes for the iterative solution of linear systems. See, for example, [6] for an overview of the theory of such methods. SLES creates a simplified interface to the lower-level KSP and PC modules within the PETSc package. The KSP component, discussed in Section 4.3, provides many popular Krylov subspace iterative methods; the PC module, described in Section 4.4, includes a variety of preconditioners. Although both KSP and PC can be used directly, users should employ the interface of SLES.

4.1 Using SLES

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

```
ierr = SLESCreate(MPI_Comm comm,SLES *sles);
```

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

```
ierr = SLESSetOperators(SLES sles,Mat Amat,Mat Pmat,MatStructure flag);
```

The argument `Amat`, representing the matrix that defines the linear system, is a symbolic place holder for any kind of matrix. In particular, SLES *does* support matrix-free methods. The routine `MatCreateShell()` in Section 3.3 provides further information regarding matrix-free methods. Typically the preconditioning matrix, `Pmat`, is the same as the matrix that defines the linear system, `Amat`; however, occasionally these matrices differ (for instance, when preconditioning a matrix obtained from a high order method with that from a low order method). The argument `flag` can be used to eliminate unnecessary work when repeatedly solving linear systems of the same size with the same preconditioning method; when solving just one linear system, this flag is ignored. The user can set `flag` as follows:

- **SAME_NONZERO_PATTERN** - the preconditioning matrix has the same nonzero structure during successive linear solves,
- **DIFFERENT_NONZERO_PATTERN** - the preconditioning matrix does not have the same nonzero structure during successive linear solves,
- **SAME_PRECONDITIONER** - the preconditioner matrix is identical to that of the previous linear solve.

If in doubt about the structure of a matrix, one should use the flag **DIFFERENT_NONZERO_PATTERN**.

Much of the power of SLES can be accessed through the single routine

```
ierr = SLESSetFromOptions(SLES sles);
```

This routine accepts the options `-h` and `-help` as well as any of the KSP and PC options discussed below. To solve a linear system, one merely executes the command

```
ierr = SLESSolve(SLES sles,Vec b,Vec x,int *its);
```

where `b` and `x` respectively denote the right-hand-side and solution vectors. On return, the parameter `its` contains either the iteration number at which convergence was successfully reached, or the *negative* of the iteration at which divergence or breakdown was detected. Section 4.3.2 gives for details regarding convergence testing. Note that multiple linear solves can be performed by the same SLES context. Once the SLES context is no longer needed, it should be destroyed with the command

```
ierr = SLESDestroy(SLES sles);
```

The above procedure is sufficient for general use of the SLES package. One additional step is required for users who wish to customize certain preconditioners (e.g., see Section 4.4.4) or to log certain performance data using the PETSc profiling facilities (as discussed in Chapter 10). In this case, the user can optionally explicitly call

```
ierr = SLESSetUp(SLES sles,Vec b,Vec x);
```

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

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,

```
ierr = SLESGetPC(SLES sles,PC *pc);
```

```
ierr = SLESGetKSP(SLES sles,KSP *ksp);
```

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

To solve a linear system with a direct solver (currently supported only for sequential matrices), one may use the options `-pc_type lu -ksp_type preonly` (see below).

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

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* preconditioner matrix (i.e., the same data structure with exactly the same matrix elements) but different right-hand-side vectors, the user should simply call `SLESSolve()` multiple times. The preconditioner setup operations (e.g., factorization for ILU) will be done during the first call to `SLESSolve()` only; such operations will *not* be repeated for successive solves.

To solve successive linear systems that have *different* preconditioner matrices (i.e., the matrix elements and/or the matrix data structure change), the user *must* call `SLESSetOperators()` and `SLESSolve()` for each solve. See Section 4.1 for a description of various flags for `SLESSetOperators()` that can save work for such cases.

4.3 KSP Component

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

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

The type can be one of `KSPRICHARDSON`, `KSPCHEBYCHEV`, `KSPCG`, `KSPGMRES`, `KSPQCQMR`, `KSPBCGS`, `KSPCGS`, `KSPTFQMR`, `KSPCR`, `KSPLSQR`, or `KSPPREONLY`. The KSP method can also be set with the options database command `-ksp_type`, followed by one of the options `richardson`, `chebychev`, `cg`, `gmres`, `tcqmr`, `bcgs`, `cgs`, `tfqmr`, `cr`, `lsqr`, or `preonly`. There are method-specific options for the Richardson, Chebychev, and GMRES methods.

```

ierr = KSPRichardsonSetScale(KSP ksp,double damping_factor);
ierr = KSPChebychevSetEigenvalues(KSP ksp,double emax,double emin);
ierr = KSPGMRESRestart(KSP ksp,int max_steps);

```

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

The default technique for orthogonalization of the Hessenberg matrix in GMRES is the modified Gram-Schmidt method, which employs many `VecDot()` operations and can thus be slow in parallel. A fast approach is to use the unmodified Gram-Schmidt method, which can be set with

```

ierr = KSPGMRESSetOrthogonalization(KSP ksp,
                                     KSPGMRESUnmodifiedGramSchmidtOrthogonalization);

```

or the options database command `-ksp_gmres_unmodifiedgramschmidt`. Note that this algorithm is numerically unstable, but may deliver much better speed performance. One can also use unmodified Gram-Schmidt with iterative refinement, by setting the orthogonalization routine, `KSPGMRESIROrthog()`, by using the command line option `-ksp_gmres_iorthog`.

By default, KSP assumes an initial guess of zero by zeroing the initial value for the solution vector that is given. To use a nonzero initial guess, the user *must* call

```

ierr = KSPSetInitialGuessNonzero(KSP ksp);

```

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

```

ierr = KSPCGSetType(KSP ksp,KSPCGType KSP_CG_SYMMETRIC);

```

4.3.1 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 (4.1) by

$$(M_L^{-1} A M_R^{-1}) (M_R x) = M_L^{-1} b, \quad (4.2)$$

where M_L and M_R indicate preconditioning matrices. If $M_L = I$ in (4.2), right preconditioning results, and the residual of (4.1),

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

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, all KSP implementations use left preconditioning. Right preconditioning can be activated for some methods by using the options database command `-ksp_right_pc` or calling the routine

```

ierr = KSPSetPreconditionerSide(KSP ksp,PCSide PC_RIGHT);

```

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

```

KSPSetUp_Richardson:No right preconditioning for KSPRICHARDSON

```

We summarize the defaults for the residuals used in KSP convergence monitoring within Table 3. Details regarding specific convergence tests and monitoring routines are presented in the following sections. The preconditioned residual is used by default for convergence testing of all left-preconditioned KSP methods *except* for the conjugate gradient, Richardson, and Chebyshev methods. For these three cases the true residual is used by default, but the preconditioned residual can be employed instead with the options database command `ksp_preres` or by calling the routine

```

ierr = KSPSetUsePreconditionedResidual(KSP ksp);

```


Table 3: KSP Defaults. All methods use left preconditioning by default.

Method	KSPType	Options Database Name	Default Convergence Monitor †
Richardson	KSPRICHARDSON	richardson	true
Chebyshev	KSPCHEBYCHEV	chebychev	true
Conjugate Gradient [10]	KSPCG	cg	true
Generalized Minimal Residual [15]	KSPGMRES	gmres	precond
BiCGSTAB [19]	KSPBCGS	bcgs	precond
Conjugate Gradient Squared [17]	KSPCGS	cgs	precond
Transpose-Free Quasi-Minimal Residual (1) [7]	KSPTFQMR	tfqmr	precond
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr	precond
Conjugate Residual	KSPCR	cr	precond
Least Squares Method	KSPLSQR	lsqr	precond
Shell for no KSP method	KSPPREONLY	preonly	precond

† true - denotes true residual norm, precondition - denotes preconditioned residual norm

4.3.2 Convergence Tests

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

$$\|r_k\|_2 < \max(\text{rtol} * \|r_0\|_2, \text{atol}),$$

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

$$\|r_k\|_2 > \text{dtol} * \|r_0\|_2.$$

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

```
ierr = KSPSetTolerances(KSP ksp, double rtol, double atol, double dtol, int maxits);
```

The user can retain the default value of any of these parameters by specifying `PETSC_DEFAULT` as the corresponding tolerance; the defaults are `rtol`= 10^{-5} , `atol`= 10^{-50} , `dtol`= 10^5 , and `maxits`= 10^5 . These parameters can also be set from the options database with the commands `-ksp_rtol <rtol>`, `-ksp_atol <atol>`, `-ksp_divtol <dtol>`, and `-ksp_max_it <its>`.

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

```
ierr = KSPSetConvergenceTest(KSP ksp, int (*test)(KSP ksp, int it, double rnorm, void *ctx), void *ctx);
```

The final routine argument, `ctx`, is an optional context for private data for the user-defined convergence routine, `test`. Other `test` routine arguments are the iteration number, `it`, and the residual's l_2 norm, `rnorm`. The routine for detecting convergence, `test`, should return the integer 1 for convergence, 0 for no convergence, and -1 on error or failure to converge.

4.3.3 Convergence Monitoring

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

```

ierr = KSPSetMonitor(KSP ksp,int (*mon)(KSP ksp,int it,double rnorm,void *ctx),
                    void *ctx);

```

The final routine argument, `ctx`, is an optional context for private data for the user-defined monitoring routine, `mon`. Other `mon` routine arguments are the iteration number (`it`) and the residual's l_2 norm (`rnorm`). A helpful routine within user-defined monitors is `PetscObjectGetComm((PetscObject)ksp,MPI_Comm *comm)`, which returns in `comm` the MPI communicator for the KSP context. See Chapter 15 for more discussion of the use of MPI communicators within PETSc.

Several monitoring routines are supplied with PETSc, including

```

ierr = KSPDefaultMonitor(KSP,int,double, void *);
ierr = KSPSingularValueMonitor(KSP,int,double, void *);
ierr = KSPTrueMonitor(KSP,int,double, void *);

```

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

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

```

DrawLG lg;
ierr = KSPLGMonitorCreate(char *display,char *title,int x,int y,int w,int h,DrawLG *lg);
ierr = KSPSetMonitor(KSP ksp,KSPLGMonitor,(void *)lg);

```

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

```

ierr = KSPLGMonitorDestroy(DrawLG lg);

```

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

Once can cancel all hardwired monitoring routines for KSP at runtime with `-ksp_cancelmonitors`.

4.3.4 Understanding the Operators Spectrum

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

```

ierr = KSPSetComputeEigenvalues(KSP ksp);

```

Then after the SLES solve one calls

```

ierr = KSPComputeEigenvalues(KSP ksp, int n,double *realpart,double *complexpart);

```

Here, `n` is the size of the two arrays and the eigenvalues are inserted into those two arrays. There is an additional routine

```

ierr = KSPComputeEigenvaluesExplicitly(KSP ksp, int n,double *realpart,double *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 used only for matrices of size up to a couple of hundred. The `DrawSP*()` 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_plot_eigenvalues` and `-ksp_plot_eigenvalues_explicitly`. Or they can be dumped to the screen in ASCII text via `-ksp_compute_eigenvalues` and `-ksp_compute_eigenvalues_explicitly`.

4.3.5 Other KSP Options

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

```

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

```

Table 4: PETSc Preconditioners

Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
Block Gauss-Seidel (sequential only)	PCBGS	bgs
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
LU	PCLU	lu
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

These routines return the original vectors that the user set with `KSPSetSolution()` and `KSPSetRhs()`. 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

```
ierr = 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 `PETSC_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 requires not even a vector copy.

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

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

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

4.4 Preconditioners

As discussed in Section 4.3.1, the Krylov space methods are typically used in conjunction with a preconditioner. To employ a particular preconditioning method, the user can either select it from the options database using input of the form `-pc_type <methodname>` or set the method with the command

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

In Table 4 we summarize the most basic preconditioning methods supported in PETSc. The `PCSHELL` preconditioner uses a specific, application-provided preconditioner. The direct preconditioner, `PCLU`, is, in fact, a direct solver for the linear system that uses LU factorization. `PCLU` is included as a preconditioner so that PETSc has a consistent interface among direct and iterative linear solvers.

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

4.4.1 ILU and ICC Preconditioners

Some of the options for ILU preconditioner are

```
ierr = PCILUSetLevels(PC pc, int levels);
ierr = PCILUSetReuseReordering(PC pc, PetscTruth flag);
ierr = PCILUSetUseDropTolerance(PC pc, double dt, int dtcount);
ierr = PCILUSetReuseFill(PC pc, PetscTruth flag);
ierr = PCILUSetUseInPlace(PC pc);
```

When repeatedly solving linear systems with the same SLES context, one can reuse some information computed during the first linear solve. In particular, `PCILUSetReuseReordering()` causes the reordering (for example, set with `-mat_order order`) computed in the first factorization to be reused for later factorizations. The `PCILUSetReuseFill()` causes the fill computed during the first drop tolerance factorization to be reused in later factorizations. `PCILUSetUseInPlace()` 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_ilu_levels <levels>
-pc_ilu_reuse_reordering
-pc_ilu_use_drop_tolerance <dt>,<dtcount>
-pc_ilu_reuse_fill
-pc_ilu_in_place
-pc_ilu_nonzeros_along_diagonal
```

See Section 11.4.2 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.

We support incomplete factorization preconditioners for several matrix types for the uniprocessor case. In addition, for the parallel case we provide an interface to the ILU and ICC preconditioners of BlockSolve95 [11]. BlockSolve95 is available by anonymous ftp at [info.mcs.anl.gov](http://info.mcs.anl.gov/pub/BlockSolve95) in the directory `pub/BlockSolve95`; for further information see the WWW address: <http://www.mcs.anl.gov/blocksolve95/index.html>. PETSc enables users to employ the preconditioners within BlockSolve95 by using the BlockSolve95 matrix format `MATMPIROWBS` and invoking either the `PCILU` or `PCICC` method within the linear solvers. Since PETSc automatically handles matrix assembly, preconditioner setup, profiling, and so on, users who employ BlockSolve95 through the PETSc interface need not concern themselves with many details provided within the BlockSolve95 users manual.

One can create a matrix that is compatible with BlockSolve95 by using `MatCreate()` with the option `-mat_mpirowbs`, or by directly calling

```
ierr = MatCreateMPIRowbs(MPI_Comm comm,int m,int M,int nz,int *nnz,void *proci,Mat *A)
```

`A` is the newly created matrix, while the arguments `m` and `M` indicate the number of local and global rows, respectively. Either the local or global parameter can be replaced with `PETSC_DECIDE`, so that PETSc will determine it. The matrix is stored with a fixed number of rows on each processor, given by `m`, or determined by PETSc if `m` is `PETSC_DECIDE`. The arguments `nz` and `nnz` can be used to preallocate storage space, as discussed in Section 3.1 for increasing the efficiency of matrix assembly; one sets `nz=0` and `nnz=PETSC_NULL` for PETSc to control all matrix memory allocation. The argument `proci` is an optional BlockSolve95 `BSprocinfo` context; most users should set this parameter to `PETSC_NULL`, so that PETSc will create and initialize this context.

If the matrix is symmetric, one *may* call

```
ierr = MatSetOption(Mat mat,MAT_SYMMETRIC);
```

to improve efficiency, but in this case one cannot use the ILU preconditioner, only ICC.

Internally, PETSc inserts zero elements into matrices of the `MATMPIROWBS` format if necessary, so that nonsymmetric matrices are considered to be symmetric in terms of their sparsity structure; this format is required for use of the parallel communication routines within BlockSolve95. In particular, if the matrix element $A[i, j]$ exists, then PETSc will internally allocate a 0 value for the element $A[j, i]$ during `MatAssemblyEnd()` if the user has not already set a value for the matrix element $A[j, i]$.

When manipulating a preconditioning matrix, A , BlockSolve95 internally works with a scaled and permuted matrix, $\hat{A} = PD^{-1/2}AD^{-1/2}$, where D is the diagonal of A , and P is a permutation matrix determined by a graph coloring for efficient parallel computation. Thus, when solving a linear system, $Ax = b$, using ILU/ICC preconditioning and the matrix format `MATMPIROWBS` for *both* the linear system matrix and the preconditioning matrix, one actually solves the scaled and permuted system $\hat{A}\hat{x} = \hat{b}$, where $\hat{x} = PD^{1/2}x$ and $\hat{b} = PD^{-1/2}b$. PETSc handles the internal scaling and permutation of x and b , so the user does *not* deal with these conversions, but instead always works with the original linear system. In this case, by default the scaled residual norm is monitored; one must use the option `-ksp_bsmonitor` to print both the scaled and unscaled residual norms. *Note:* If one is using ILU/ICC via BlockSolve95 and the `MATMPIROWBS` matrix

format for the preconditioner matrix, but using a different format for a different linear system matrix, this scaling and permuting are done only internally during the application of the preconditioner; `ksp_bsmonitor` should not be used in this case.

Users who wish to use these preconditioners via the PETSc interface and who have not already installed BlockSolve95 should see the file `$(PETSC_DIR)/Installation` for details on building PETSc when BlockSolve95 is being used. In particular, one must edit the file `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base.site` so that `BS_LIB` and `BS_INCLUDE` indicate the library location, and the `PCONF` variable specifies the flag `-DHAVE_BLOCKSOLVE`. Then the PETSc library must be (re)compiled.

4.4.2 SOR and SSOR Preconditioners

The options for SOR preconditioning are

```
ierr = PCSORSetOmega(PC pc, double omega);
ierr = PCSORSetIterations(PC pc, int its);
ierr = 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 of SOR, given by `its`, to use between steps of the Krylov space method. 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 processor 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_backward`, `-pc_sor_symmetric`, `-pc_sor_local_forward`, `-pc_sor_local_backward`, and `-pc_sor_local_symmetric`.

The Eisenstat trick [4] 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_diagonal_scaling` (or the routine `PCEisenstatUseDiagonalScaling()`) activates diagonal scaling in conjunction with Eisenstat SSOR method, while the option `-pc_eisenstat_omega <omega>` (or the routine `PCEisenstatSetOmega(PC pc, double omega)`) sets the SSOR relaxation coefficient, `omega`, as discussed above.

4.4.3 LU Factorization

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

```
ierr = PCLUSetUseInPlace(PC pc);
```

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

```
-mat_order <ordering>
```

The possible orderings are

- `ORDER_NATURAL` - Natural
- `ORDER_ND` - Nested Dissection
- `ORDER_1WD` - One-way Dissection
- `ORDER_RCM` - Reverse Cuthill-McKee
- `ORDER_QMD` - Quotient Minimum Degree

These orderings can also be set through the options database by specifying one of the following: `-mat_order natural`, `-mat_order nd`, `-mat_order 1wd`, `-mat_order rcm`, `-mat_order qmd`. In addition, see `MatGetReordering()`, discussed in Section 7.1.

The sparse LU factorization provided in PETSc does not perform pivoting for numerical stability (since they are designed to preserve nonzero structure); thus, occasionally a LU factorization will fail with a zero

pivot when, in fact, the matrix is nonsingular. The option `-pc_lu_nonzeros_along_diagonal` will often help eliminate the zero pivot, by preprocessing the column ordering to remove small values from the diagonal.

In addition, Section 11.4.2 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.

4.4.4 Block Jacobi, Block Gauss-Seidel, and Overlapping Additive Schwarz Preconditioners

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

```
ierr = PCBJacobiGetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles);
ierr = PCBGSSetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles);
ierr = PCASMSetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles);
```

extract the SLES context for each local block. The argument `n_local` is the number of blocks on the calling processor, and `first_local` indicates the global number of the first block on the processor. The blocks are numbered successively by processors from zero through $gb - 1$, where gb is the number of global blocks. The array of SLES contexts for the local blocks is given by `subsles`. This mechanism enables the user to set completely different solvers for the various blocks. To set the appropriate data structures, the user *must* explicitly call `SLESSetUp()` before calling `PCBJacobiGetSubSLES()`, `PCBGSSetSubSLES()`, or `PCASMSetSubSLES()`. For further details, see the example `$(PETSC_DIR)/src/sles/examples/tutorials/ex7.c`.

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

```
ierr = PCBJacobiSetTotalBlocks(PC pc,int blocks,int *size);
ierr = PCBGSSetTotalBlocks(PC pc,int blocks,int *size);
ierr = PCASMSetTotalSubdomains(PC pc,int n,IS *is);
ierr = PCASMSetType(PC pc,PCASMSType 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 processor is supported. However, the `MATMPIAIJ` and `MATMPIBAIJ` formats support the use of general blocks as long as no blocks are shared among processors. The `is` argument contains the index sets that define the subdomains.

`PCASMSType` is one of `PC_ASM_BASIC`, `PC_ASM_INTERPOLATE`, `PC_ASM_RESTRICT`, `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-processor 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-processor values for both restriction and interpolation. The ASM types with limited restriction or interpolation were suggested by Xiao-Chuan Cai. `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-processor basis with the commands

```
ierr = PCBJacobiSetLocalBlocks(PC pc,int blocks,int *size);
ierr = PCBGSSetLocalBlocks(PC pc,int blocks,int *size);
ierr = PCASMSetTotalSubdomains(PC pc,int N,IS *is);
```

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

```
ierr = PCASMSetOverlap(PC pc,int overlap);
```

The overlap defaults to 1, so if one desires that no additional overlap be computed, one must set an **overlap** of 0. Note that one can define initial index sets **is** with any overlap; the **PCASMSetOverlap()** merely allows PETSc to extend that overlap further, if desired.

4.4.5 Shell Preconditioners

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

```
ierr = PCShellSetApply(PC pc,int (*apply)(void *ctx,Vec,Vec),void *ctx);
```

The final argument **ctx** is a pointer to the application-provided data structure needed by the preconditioner routine. The three routine arguments of **apply()** are this context, the input vector, and the output vector, respectively.

4.4.6 Multigrid Preconditioners

A large suite of routines is available for using multigrid as a preconditioner. In the **PC** framework the user is required to provide the coarse grid solver, smoothers, restriction, and interpolation, as well as the code to calculate residuals. The **PC** component allows all of that to be wrapped up into a PETSc compliant preconditioner. We fully support both matrix-free and matrix-based multigrid solvers.

A multigrid preconditioner is created with the four commands

```
ierr = SLESCreate(MPI_Comm comm,SLES *sles);
ierr = SLESGetPC(SLES sles,PC *pc);
ierr = PCSetType(PC pc,PCMG);
ierr = MGSetLevels(pc,int levels);
```

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

```
ierr = MGSetType(PC pc,MGType mode);
```

indicates which form of multigrid to apply [16]. For standard V or W-cycle multigrids, one sets the **mode** to be **MGMULTIPLICATIVE**; for the additive form (which in certain cases reduces to the BPX method, or additive multilevel Schwarz, or multilevel diagonal scaling), one uses **MGADDITIVE** as the **mode**. For a variant of full multigrid, one can use **MGFULL**, and for the Kaskade algorithm **MGKASKADE**. For the multiplicative and full multigrid options, one can use a W-cycle by calling

```
ierr = MGSetCycles(PC pc,int cycles);
```

with a value of **MG_WCYCLE** for **cycles**. The commands above can also be set from the options database. The option names are **-pc_mg_method** [**multiplicative**, **additive**, **full**, **kaskade**], and **-pc_mg_cycles** **cycles**.

The user can control the amount of pre- and postsmoothing by using either the options **-pc_mg_smoothup m** and **-pc_mg_smoothdown n** or the routines

```
ierr = MGSetNumberSmoothUp(PC pc,int m);
ierr = MGSetNumberSmoothDown(PC pc,int n);
```

Note that if the command **MGSetSmoother()** (discussed below) has been employed, the same amounts of pre- and postsmoothing must be used.

The remainder of 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

```
ierr = MGGetCoarseSolve(PC pc,SLES *sles);
```

and set the appropriate options in **sles**. Similarly, the smoothers are set by calling

```
ierr = MGGetSmoother(PC pc,int level,SLES *sles);
```

and setting the various options in **sles**. To use a different pre- and postsmoother, one should call the following routines instead operations to be matrix free (see Section 3.3), he or she should make sure that these operations are defined. Note that this system is arranged so that if the interpolation is the transpose of the restriction, the same **mat** argument can be passed to both **MGSetRestriction()** and **MGSetInterpolation()**.

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

```
MGSetResidual(PC pc,int level,int (*residual)(Mat,Vec,Vec,Vec),Mat mat);
```

The `residual()` function can be set to be `MGDefaultResidual()` if one's operator is stored in a `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 must 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

```
ierr = MGSetRhs(PC pc,int level,Vec b);  
ierr = MGSetX(PC pc,int level,Vec x);  
ierr = MGSetR(PC pc,int level,Vec r);
```

The user is responsible for freeing these vectors once the iteration is complete.

Chapter 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 component 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 uniprocessor and parallel cases; the only difference in the parallel version is that each processor typically forms only its local contribution to various matrices and vectors.

SNES includes methods for solving systems of nonlinear equations of the form

$$\mathbf{F}(\mathbf{x}) = 0, \quad (5.1)$$

where $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$. SNES also contains solvers for unconstrained minimization problems of the form

$$\min\{f(\mathbf{x})\}, \quad (5.2)$$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Newton-like methods provide the core of the package, including both line search and trust region techniques, which are discussed further in Section 5.2. 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.

The general form of the n -dimensional Newton's method for solving (5.1) is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{F}'(\mathbf{x}_k)]^{-1} \mathbf{F}(\mathbf{x}_k), \quad k = 0, 1, \dots, \quad (5.3)$$

where \mathbf{x}_0 is an initial approximation to the solution and $\mathbf{F}'(\mathbf{x}_k)$ is nonsingular. In practice, the Newton iteration (5.3) is implemented by the following two steps:

1. (Approximately) solve $\mathbf{F}'(\mathbf{x}_k) \Delta \mathbf{x}_k = -\mathbf{F}(\mathbf{x}_k)$. (5.4)

2. Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$. (5.5)

Similarly, the general form of Newton's method for solving (5.2) is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\nabla^2 f(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k), \quad k = 0, 1, \dots, \quad (5.6)$$

where $\mathbf{x}_0 \in \mathbb{R}^n$ is an initial approximation to the solution, and $\nabla^2 f(\mathbf{x}_k)$ is positive definite. The iteration (5.6) is usually implemented by

1. (Approximately) solve $\nabla^2 f(\mathbf{x}_k) \Delta \mathbf{x}_k = -\nabla f(\mathbf{x}_k)$. (5.7)

2. Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$. (5.8)

5.1 Basic Usage

In the simplest usage of the nonlinear solvers, the user must merely provide a C, C++, or Fortran routine to evaluate the nonlinear function of Equation (5.1) or (5.2). The corresponding Jacobian matrix (or gradient

and Hessian 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 (or gradient and Hessian); details regarding these application-provided routines are discussed below. To provide an overview of the use of the nonlinear solvers, we first introduce a complete and simple example in Figure 13, corresponding to `$(PETSC_DIR)/src/snes/examples/tutorials/ex1.c`. Note that the procedures for solving systems of nonlinear equations and unconstrained minimization problems are quite similar. We present the details unique to each class of problems in Sections 5.1.1 and 5.1.2.

```
#ifndef lint
static char vcid[] = "$Id: ex1.c,v 1.5 1997/01/01 03:41:24 bsmith Exp $";
#endif

static char help[] = "Uses Newton's method to solve a two-variable system.\n\n";

/*T
  Concepts: SNES`Solving a system of nonlinear equations (basic uniprocessor example);
  Routines: SNESCreate(); SNESSetFunction(); SNESSetJacobian(); SNESGetSLES();
  Routines: SNESsolve(); SNESSetFromOptions();
  Routines: SLESGetPC(); SLESGetKSP(); KSPSetTolerances(); PCSetType();
  Processors: 1
T*/

/*
  Include "snes.h" so that we can use SNES solvers. Note that this
  file automatically includes:
    petsc.h - base PETSc routines    vec.h - vectors
    sys.h   - system routines        mat.h - matrices
    is.h    - index sets             ksp.h - Krylov subspace methods
    viewer.h - viewers               pc.h  - preconditioners
    sles.h   - linear solvers
*/
#include "snes.h"
#include <stdio.h>

/*
  User-defined routines
*/
int FormJacobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int FormFunction(SNES,Vec,Vec,void*);

int main( int argc, char **argv )
{
  SNES      snes;          /* nonlinear solver context */
  SLES      sles;          /* linear solver context */
  PC        pc;            /* preconditioner context */
  KSP       ksp;           /* Krylov subspace method context */
  Vec       x, r;          /* solution, residual vectors */
  Mat       J;             /* Jacobian matrix */
  int       ierr, its, size;
  Scalar    pfive = .5;

  PetscInitialize( &argc, &argv,(char *)0,help );
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  if (size != 1) SETERRA(1,0,"This is a uniprocessor example only!");

  /* - - - - -
     Create nonlinear solver context
     - - - - - */

  ierr = SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&snes); CHKERRA(ierr);
```

```

/* - - - - -
   Create matrix and vector data structures; set corresponding routines
   - - - - - */

/*
   Create vectors for solution and nonlinear function
*/
ierr = VecCreateSeq(MPI_COMM_SELF,2,&x); CHKERRA(ierr);
ierr = VecDuplicate(x,&r); CHKERRA(ierr);

/*
   Create Jacobian matrix data structure
*/
ierr = MatCreate(MPI_COMM_SELF,2,2,&J); CHKERRA(ierr);

/*
   Set function evaluation routine and vector.
*/
ierr = SNESSetFunction(snes,r,FormFunction,PETSC_NULL); CHKERRA(ierr);

/*
   Set Jacobian matrix data structure and Jacobian evaluation routine
*/
ierr = SNESSetJacobian(snes,J,J,FormJacobian,PETSC_NULL); CHKERRA(ierr);

/* - - - - -
   Customize nonlinear solver; set runtime options
   - - - - - */

/*
   Set linear solver defaults for this problem. By extracting the
   SLES, KSP, and PC contexts from the SNES context, we can then
   directly call any SLES, KSP, and PC routines to set various options.
*/
ierr = SNESGetSLES(snes,&sles); CHKERRA(ierr);
ierr = SLESGetKSP(sles,&ksp); CHKERRA(ierr);
ierr = SLESGetPC(sles,&pc); CHKERRA(ierr);
ierr = PCSetType(pc,PCNONE); CHKERRA(ierr);
ierr = KSPSetTolerances(ksp,1.e-4,PETSC_DEFAULT,PETSC_DEFAULT,20); CHKERRA(ierr);

/*
   Set SNES/SLES/KSP/PC runtime options, e.g.,
       -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
   These options will override those specified above as long as
   SNESSetFromOptions() is called _after_ any other customization
   routines.
*/
ierr = SNESSetFromOptions(snes); CHKERRA(ierr);

/* - - - - -
   Evaluate initial guess; then solve nonlinear system
   - - - - - */

/*
   Note: The user should initialize the vector, x, with the initial guess
   for the nonlinear solver prior to calling SNESolve(). In particular,
   to employ an initial guess of zero, the user should explicitly set
   this vector to zero by calling VecSet().
*/

```

```

ierr = VecSet(&pfive,x); CHKERRA(ierr);
ierr = SNESolve(snes,x,&its); CHKERRA(ierr);
PetscPrintf(MPI_COMM_SELF,"number of Newton iterations = %d\n\n", its);

/* - - - - -
   Free work space. All PETSc objects should be destroyed when they
   are no longer needed.
   - - - - - */

ierr = VecDestroy(x); CHKERRA(ierr); ierr = VecDestroy(r); CHKERRA(ierr);
ierr = MatDestroy(J); CHKERRA(ierr); ierr = SNESDestroy(snes); CHKERRA(ierr);

PetscFinalize();
return 0;
}
/* ----- */
/*
   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
*/
int FormFunction(SNES snes,Vec x,Vec f,void *dummy)
{
    int    ierr;
    Scalar *xx, *ff;

    /*
       Get pointers to vector data.
       - For default PETSc vectors, VecGetArray() returns a pointer to
       the data array. Otherwise, the routine is implementation dependent.
       - You MUST call VecRestoreArray() when you no longer need access to
       the array.
    */
    ierr = VecGetArray(x,&xx); CHKERRQ(ierr);
    ierr = VecGetArray(f,&ff); CHKERRQ(ierr);

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

    /*
       Restore vectors
    */
    ierr = VecRestoreArray(x,&xx); CHKERRQ(ierr);
    ierr = VecRestoreArray(f,&ff); CHKERRQ(ierr);

    return 0;
}
/* ----- */
/*
   FormJacobian - Evaluates Jacobian matrix.

```

```

    Input Parameters:
.   snes - the SNES context
.   x - input vector
.   dummy - optional user-defined context (not used here)

    Output Parameters:
.   jac - Jacobian matrix
.   B - optionally different preconditioning matrix
.   flag - flag indicating matrix structure
*/
int FormJacobian(SNES snes,Vec x,Mat *jac,Mat *B,MatStructure *flag,void *dummy)
{
    Scalar *xx, A[4];
    int ierr, idx[2] = {0,1};

    /*
       Get pointer to vector data
    */
    ierr = VecGetArray(x,&xx); CHKERRQ(ierr);

    /*
       Compute Jacobian entries and insert into matrix.
       - Since this is such a small problem, we set all entries for
         the matrix at once.
    */
    A[0] = 2.0*xx[0] + xx[1]; A[1] = xx[0];
    A[2] = xx[1]; A[3] = xx[0] + 2.0*xx[1];
    ierr = MatSetValues(*jac,2,idx,2,idx,A,INSERT_VALUES); CHKERRQ(ierr);
    *flag = SAME_NONZERO_PATTERN;

    /*
       Restore vector
    */
    ierr = VecRestoreArray(x,&xx); CHKERRQ(ierr);

    /*
       Assemble matrix
    */
    ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

    return 0;
}

```

Figure 13: Example of Uniprocessor SNES Code

To create a SNES solver, one must first call `SNESCreate()` and indicate the class of problem being solved, using one of the following:

```

ierr = SNESCreate(MPI_Comm comm,SNES_NONLINEAR_EQUATIONS,SNES *snes);
ierr = SNESCreate(MPI_Comm comm,SNES_UNCONSTRAINED_MINIMIZATION,SNES *snes);

```

When solving a system of nonlinear equations, the user must then set routines for evaluating the function of equation (5.1) and its associated Jacobian matrix. Likewise, when solving an unconstrained minimization problem, the user must indicate routines for computing the function of Equation (5.2), as well as the corresponding gradient and Hessian. Such details are discussed in Sections 5.1.1 and 5.1.2.

To choose a nonlinear solution method, the user can either call

```

ierr = SNESSetType(SNES snes,SNESType method);

```

or use the option `-snes_type <method>`, where details regarding the available methods are presented in Section 5.2. The application code can take complete control of the linear and nonlinear techniques used in the Newton-like method by calling

```
ierr = 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 SLES, KSP, and PC modules, as discussed in Chapter 4.

After having set these routines and options, the user solves the problem by calling

```
ierr = SNESolve(SNES snes, Vec x, int *iters);
```

where `iters` is the number of nonlinear iterations required for convergence and `x` indicates the solution vector. The user should initialize this vector to the initial guess for the nonlinear solver prior to calling `SNESolve()`. In particular, to employ an initial guess of zero, the user should explicitly set this vector to zero by calling `VecSet()`. Finally, after solving the nonlinear system (or several systems), the user should destroy the SNES context with

```
ierr = SNESDestroy(SNES snes);
```

5.1.1 Solving Systems of Nonlinear Equations

When solving a system of nonlinear equations, the user must provide a vector, `f`, for storing the function of Equation (5.1), as well as a routine that evaluates this function at the vector `x`. This information should be set with the command

```
ierr = SNESSetFunction(SNES snes, Vec f,
    int (*FormFunction)(SNES snes, Vec x, Vec f, void *ctx), void *ctx);
```

The argument `ctx` is an optional user-defined context, which can store any private, application-specific data required by the function evaluation routine; `PETSC_NULL` should be used if such information is not needed. In C and C++, a user-defined context is merely a structure in which various objects can be stashed; in Fortran a user context can be an integer array that contains both parameters and pointers to PETSc objects. `$(PETSC_DIR)/src/snes/examples/tutorials/ex5.c` and `$(PETSC_DIR)/src/snes/examples/ex5f.F` give examples of user-defined application contexts in C and Fortran, respectively.

The user must also specify a routine to form some approximation of the Jacobian matrix, `A`, at the current iterate, `x`, as is typically done with

```
ierr = SNESSetJacobian(SNES snes, Mat A, Mat B, int (*FormJacobian)(SNES snes, Vec x,
    Mat *A, Mat *B, MatStructure *flag, void *ctx), void *ctx);
```

The arguments of the routine `FormJacobian()` are the current iterate, `x`; the Jacobian matrix, `A`; the preconditioner matrix, `B` (which is usually the same as `A`); a `flag` indicating information about the preconditioner matrix structure; and an optional user-defined Jacobian context, `ctx`, for application-specific data. The options for `flag` are identical to those for the flag of `SLESSetOperators()`, discussed in Section 4.1. 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. Chapter 3 discusses information regarding available matrix formats and options, while Section 5.5 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.

During successive calls to `FormJacobian()`, the user can either insert new matrix contexts or reuse old ones, depending on the application requirements. For many sparse matrix formats, reusing the old space (and merely changing the matrix elements) is more efficient; however, if the matrix structure completely changes, then 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 Section 3.4 for details on the reuse of the matrix context.

If the preconditioning matrix retains identical nonzero structure during successive nonlinear iterations, setting the parameter, `flag`, in the `FormJacobian()` routine to be `SAME_NONZERO_PATTERN` and reusing the matrix context can save considerable overhead. For example, when one is using a parallel preconditioner such as incomplete factorization in solving the linearized Newton systems for such problems, matrix colorings and communication patterns can be determined a single time and then reused repeatedly throughout the solution process. In addition, if using different matrices for the actual Jacobian and the preconditioner, the user can hold the preconditioner matrix fixed for multiple iterations by setting `flag` to `SAME_PRECONDITIONER`. See the discussion of `SLESSetOperators()` in Section 4.1 for details.

The directory `$(PETSC_DIR)/src/snes/examples/tutorials` provides a variety of examples.

Table 5: PETSc Nonlinear Solvers

Method	SNES Type	Options Name	Default Convergence Test
Line search	SNES_EQ_LS	ls	SNESConverged_EQ_LS()
Trust region	SNES_EQ_TR	tr	SNESConverged_EQ_TR()
Test Jacobian	SNES_EQ_TEST	test	
Line search	SNES_UM_LS	umls	SNESConverged_UM_LS()
Trust region	SNES_UM_TR	umtr	SNESConverged_UM_TR()

5.1.2 Solving Unconstrained Minimization Problems

As previously discussed, use of SNES for solving systems of nonlinear equations and unconstrained minimization problems is quite similar. When solving minimization problems, the user typically provides routines for evaluating the function, gradient, and Hessian corresponding to Equation (5.2). The routine to evaluate the scalar minimization function, $f(\mathbf{x})$, should be set with

```
ierr = SNESSetMinimizationFunction(SNES snes,
    int (*FormMinFunction)(SNES snes,Vec x,double *f,void *ctx),void *ctx);
```

The gradient vector, $g(\mathbf{x})$, and gradient evaluation routine should be set with

```
ierr = SNESSetGradient(SNES snes,Vec g,
    int (*FormGradient)(SNES snes,Vec x,Vec g,void *ctx),void *ctx);
```

In these routines, the argument `ctx` specifies an optional context for application-specific data, as described in Section 5.1.1.

The user must also set a routine to form some approximation of the Hessian matrix, A , as is typically done with

```
ierr = SNESSetHessian(SNES snes,Mat A,Mat B,int (*FormHessian)(SNES snes,Vec x,
    Mat *A,Mat *B,MatStructure *flag,void *ctx),void *ctx);
```

The arguments of the routine `FormHessian()` are the current iterate, \mathbf{x} ; the Hessian matrix, A ; the preconditioner matrix, B (which is usually the same as A); a `flag` indicating information about the preconditioner structure; and an optional user-defined Hessian context, `ctx`. Reuse of matrix and preconditioner data during successive iterations of the nonlinear solvers is often critical for achieving good performance. This topic is discussed in detail for the case of solving systems of nonlinear equations in Section 5.1.1; the options are identical for solving unconstrained minimization problems, and thus are not repeated here.

The directory `$(PETSC_DIR)/src/snes/examples/tests/umin` provides examples of solving unconstrained minimization problems.

5.2 The Various Nonlinear Solvers

As summarized in Table 5, SNES includes several Newton-like nonlinear solvers based on line search techniques and trust region methods. The methods for solving systems of nonlinear equations and unconstrained minimization problems employ the prefixes `SNES_EQ` and `SNES_UM`, respectively.

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.

5.2.1 Line Search Techniques

The method `SNES_EQ_NLS` (`-snes_type ls`) provides a line search Newton method for solving systems of nonlinear equations. By default, this technique employs cubic backtracking [3]. An alternative line search routine can be set with the command

```
ierr = SNESSetLineSearch(SNES snes,
    int (*ls)(SNES,Vec,Vec,Vec,Vec,double,double*,double*));
```

Other line search methods provided by PETSc are `SNESNoLineSearch()` and `SNESQuadraticLineSearch()`, which can be set with the option `-snes_line_search [basic,quadratic,cubic]`. 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 options `-snes_line_search_alpha <alpha>`, `-snes_line_search_maxstep <max>`, and `-snes_line_search_steptol <tol>`.

The method `SNES_UM_NLS` (`-snes_type umls`) provides a line search Newton method for solving unconstrained minimization problems. The default line search algorithm is taken from Moré and Thiente [13]. Again, the user can set a variety of parameters to control the line search; one should run a SNES 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 by PETSc.

5.2.2 Trust Region Methods

The most basic trust region method in SNES for solving systems of nonlinear equations, `SNES_EQ_NTR` (`-snes_type tr`), is taken from the MINPACK project [12]. 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_trust_region_delta0 <delta0>`.

The default trust region method for unconstrained minimization, `SNES_UM_NTR` (`-snes_type umtr`), is based on the work of Steihaug [18]. This method uses the preconditioned conjugate gradient method via the KSP solver `KSPQCG` to determine the approximate minimizer of the resulting quadratic at each nonlinear iteration. This formulation requires the use of a symmetric preconditioner, where the currently available options are Jacobi, incomplete Cholesky, and the null preconditioners, which can be set with the options `-pc_type jacobi`, `-pc_type icc`, and `-pc_type none`, respectively.

5.3 General Options

This section discusses options and routines that apply to all SNES solvers and problem classes. In particular, we focus on convergence tests, monitoring routines, and tools for checking derivative computations.

5.3.1 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. The default convergence routines for the various nonlinear solvers within SNES are listed in Table 5; see the corresponding man pages for detailed descriptions. Each of these 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 (or gradient for a minimization problem). 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:

```
ierr = SNESSetTolerances(SNES snes, double rtol, double atol, double stol,
                        int its, int 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>`, and `-snes_max_funcs <fcts>`. A related routine is `SNESGetTolerances()`.

Convergence tests for trust regions methods often use an additional parameter that indicates the minimum allowable trust region radius. The user can set this parameter with the option `-snes_trtol <trtol>` or with the routine

```
ierr = SNESSetTrustRegionTolerance(SNES snes, double trtol);
```


An additional parameter is sometimes used for unconstrained minimization problems, namely, the minimum function tolerance, `ftol`, which can be set with the option `-snes_fmin <ftol>` or with the routine

```
ierr = SNESSetMinimizationFunctionTolerance(SNES snes, double ftol);
```

Users can set their own customized convergence tests in SNES by using the command

```
ierr = SNESSetConvergenceTest(SNES snes, int (*test)(SNES snes, double xnorm,
double gnorm, double f, void *cctx), void *cctx);
```

The final argument of the convergence test routine, `cctx`, denotes an optional user-defined context for private data. When solving systems of nonlinear equations, the arguments `xnorm`, `gnorm`, and `f` are the current iterate norm, current step norm, and function norm, respectively. Likewise, when solving unconstrained minimization problems, the arguments `xnorm`, `gnorm`, and `f` are the current iterate norm, current gradient norm, and the function value.

5.3.2 Convergence Monitoring

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

```
ierr = SNESSetMonitor(SNES snes, int (*mon)(SNES, int its, double norm, void* mctx),
void *mctx);
```

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 (or gradient norm for unconstrained minimization problems).

The routine set by `SNESSetMonitor()` 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, `SNESDefaultMonitor()`, while `-snes_xmonitor` draws a simple line graph of the residual norm's convergence.

One can cancel all hardwired monitoring routines for SNES at runtime with `-snes_cancelmonitors`.

The routines

```
ierr = SNESGetSolution(SNES snes, Vec *x);
```

```
ierr = SNESGetFunction(SNES snes, Vec *r);
```

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.

5.3.3 Checking Accuracy of Derivatives

Since hand-coding routines for Jacobian and Hessian 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_type test` 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_display`, which causes the two matrices to be printed to the screen.

Another means for verifying the correctness of a code for Jacobian or Hessian computation is running the problem with either the finite difference or matrix-free variant, `-snes_fd` or `-snes_mf` (see Section 5.6 or Section 5.5). 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.

5.4 Inexact Newton-like Methods

Since exact solution of the linear Newton systems within (5.3) and (5.6) 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 Chapter 4, such as `-ksp_type <ksp_method>` and `-pc_type <pc_method>`, to set the Krylov subspace and preconditioner methods.

Two levels of iterations occur for the inexact techniques, where during each global or outer Newton iteration a sequence of subsidiary inner iterations of a linear solver is performed. Appropriate control of the accuracy to which the subsidiary iterative method solves the Newton system at each global iteration is critical, since these inner iterations determine the asymptotic convergence rate for inexact Newton techniques. While the Newton systems must be solved well enough to retain fast local convergence of the Newton's iterates, use of excessive inner iterations, particularly when $\|\mathbf{x}_k - \mathbf{x}_*\|$ is large, is neither necessary nor economical. Thus, the number of required inner iterations typically increases as the Newton process progresses, so that the truncated iterates approach the true Newton iterates.

A sequence of nonnegative numbers $\{\eta_k\}$ can be used to indicate the variable convergence criterion. In this case, when solving a system of nonlinear equations, the update step of the Newton process remains unchanged, and direct solution of the linear system is replaced by iteration on the system until the residuals

$$\mathbf{r}_k^{(i)} = \mathbf{F}'(\mathbf{x}_k)\Delta\mathbf{x}_k + \mathbf{F}(\mathbf{x}_k)$$

satisfy

$$\frac{\|\mathbf{r}_k^{(i)}\|}{\|\mathbf{F}(\mathbf{x}_k)\|} \leq \eta_k \leq \eta < 1.$$

Here \mathbf{x}_0 is an initial approximation of the solution, and $\|\cdot\|$ denotes an arbitrary norm in \mathbb{R}^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 [5] to compute η_k at each step of the nonlinear solver by using the option `-snes_ksp_ew_conv`. In addition, by adding one's own KSP convergence test (see Section 4.3.2), one can easily create one's own problem-dependent, inner convergence tests.

5.5 Matrix-Free Methods

SNES fully supports matrix-free methods. The matrices specified in the Jacobian and Hessian evaluation routine need not be conventional matrices; instead, they can point to the data required to implement a particular matrix-free method. The matrix-free variant is allowed *only* when the linear systems are solved by an iterative method in combination with no preconditioning (`PCNONE` or `-pc_type none`), a user-provided preconditioner matrix, or a user-provided preconditioner shell (`PCSHELL`, discussed in Section 4.4); that is, obviously matrix-free methods cannot be used if a direct solver is to be employed.

The user can create a matrix-free context for use within SNES with the routine

```
ierr = SNESDefaultMatrixFreeMatCreate(SNES snes, Vec x, Mat *mat);
```

This routine creates the data structures needed for the matrix-vector products that arise within Krylov space iterative methods [2] by employing the matrix type `MATSHELL`, discussed in Section 3.3. 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`.

The user can set two parameters to control the Jacobian-vector product approximation with the command

```
ierr = SNESSetMatrixFreeParameters(SNES snes, double rerror, double umin);
```

The parameter `rerror` should be set to the square root of the relative error in the function evaluations, e_{rel} ; the default is $1.0e-8$, which assumes that the functions are evaluated to full double precision accuracy. The second parameter, `umin` (or u_{min}), is a bit more involved; its default is $1.0e-8$. 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

$$h = e_{rel} * u^T a / \|a\|^2 \quad \text{if } |u^T a| > u_{min} * \|a\|_1 \\ = e_{rel} * u_{min} * \text{sign}(u^T a) * \|a\|_1 / \|a\|^2 \quad \text{otherwise.}$$

This approach is taken from Brown and Saad [2]. These parameters can also be set from the options database with

```
-snes_mf_err <err>
-snes_mf_umin <umin>
```

Note that setting these parameter appropriately is crucial for achieving fast convergence with matrix-free Newton-Krylov methods.

We include an example in Figure 14 that explicitly uses a matrix-free approach. 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.

Table 6 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. All combinations are possible, as demonstrated by the example, `$(PETSC_DIR)/src/snes/-examples/ex5.c`, in Figure 14.

Table 6: Jacobian and Hessian Matrix Options

Matrix Use	Conventional Matrix Formats	Matrix-Free Versions
Jacobian (or Hessian) Matrix	Create matrix with <code>MatCreate()</code> . * Assemble matrix with user-defined routine. †	Create matrix with <code>MatCreateShell()</code> . Use <code>MatShellSetOperation()</code> to set various matrix actions.
Preconditioning Matrix	Create matrix with <code>MatCreate()</code> . * Assemble matrix with user-defined routine. †	Create PC with <code>PCShellCreate()</code> . Use <code>SNESGetSLES()</code> and <code>SLESGetPC()</code> to set the preconditioner.

* Use either the generic `MatCreate()` or a format-specific variant such as `MatCreateMPIAIJ()`.

† Set user-defined matrix formation routine with `SNESSetJacobian()` or `SNESSetHessian()`.

```
#ifndef lint
static char vcid[] = "$Id: ex6.c,v 1.45 1997/01/28 20:24:15 balay Exp $";
#endif

static char help[] = "Uses Newton-like methods to solve u'' + u^2 = f. Different\n\
matrices are used for the Jacobian and the preconditioner. The code also\n\
demonstrates the use of matrix-free Newton-Krylov methods in conjunction\n\
with a user-provided preconditioner. Input arguments are:\n\
  -snes_mf : Use matrix-free Newton methods\n\
  -user_precond : Employ a user-defined preconditioner. Used only with\n\
                  matrix-free methods in this example.\n\n";

/*T
  Concepts: SNES~Using different matrices for the Jacobian and preconditioner;
  Concepts: SNES~Using matrix-free methods and a user-provided preconditioner;
```

```

    Routines: SNESCreate(); SNESSetFunction(); SNESSetJacobian();
    Routines: SNESsolve(); SNESSetFromOptions(); SNESGetSLES();
    Routines: SLESGetPC(); PCSetType(); PCShellSetApply(); PCSetType();
    Processors: 1
T*/

/*
    Include "snes.h" so that we can use SNES solvers. Note that this
    file automatically includes:
        petsc.h - base PETSc routines    vec.h - vectors
        sys.h   - system routines        mat.h - matrices
        is.h    - index sets             ksp.h - Krylov subspace methods
        viewer.h - viewers                pc.h  - preconditioners
        sles.h  - linear solvers
*/
#include "snes.h"
#include <math.h>

/*
    User-defined routines
*/
int FormJacobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int FormFunction(SNES,Vec,Vec,void*);
int MatrixFreePreconditioner(void*,Vec,Vec);

int main( int argc, char **argv )
{
    SNES    snes;                /* SNES context */
    SLES    sles;                /* SLES context */
    PC      pc;                  /* PC context */
    Vec     x, r, F;             /* vectors */
    Mat     J, JPrec;            /* Jacobian, preconditioner matrices */
    int     ierr, its, n = 5, i, size, flg;
    double  h, xp = 0.0;
    Scalar  v, pfive = .5;

    PetscInitialize( &argc, &argv, (char *)0, help );
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 1) SETERRA(1,0,"This is a uniprocessor example only!");
    ierr = OptionsGetInt(PETSC_NULL, "-n", &n, &flg); CHKERRA(ierr);
    h = 1.0/(n-1);

    /* - - - - -
       Create nonlinear solver context
       - - - - - */

    ierr = SNESCreate(MPI_COMM_WORLD, SNES_NONLINEAR_EQUATIONS, &snes); CHKERRA(ierr);

    /* - - - - -
       Create vector data structures; set function evaluation routine
       - - - - - */

    ierr = VecCreate(MPI_COMM_SELF, n, &x); CHKERRA(ierr);
    ierr = VecDuplicate(x, &r); CHKERRA(ierr);
    ierr = VecDuplicate(x, &F); CHKERRA(ierr);

    ierr = SNESSetFunction(snes, r, FormFunction, (void*)F); CHKERRA(ierr);

    /* - - - - -
       Create matrix data structures; set Jacobian evaluation routine

```

```

- - - - - */

ierr = MatCreateSeqAIJ(MPI_COMM_SELF,n,n,3,PETSC_NULL,&J); CHKERRA(ierr);
ierr = MatCreateSeqAIJ(MPI_COMM_SELF,n,n,1,PETSC_NULL,&JPrec); CHKERRA(ierr);

/*
   Note that in this case we create separate matrices for the Jacobian
   and preconditioner matrix. Both of these are computed in the
   routine FormJacobian()
*/
ierr = SNESSetJacobian(snes,J,JPrec,FormJacobian,0); CHKERRA(ierr);

/* - - - - -
   Customize nonlinear solver; set runtime options
- - - - - */

/* Set preconditioner for matrix-free method */
ierr = OptionsHasName(PETSC_NULL,"-snes_mf",&flg); CHKERRA(ierr);
if (flg) {
    ierr = SNESGetSLES(snes,&sles); CHKERRA(ierr);
    ierr = SLESGetPC(sles,&pc); CHKERRA(ierr);
    ierr = OptionsHasName(PETSC_NULL,"-user_precond",&flg); CHKERRA(ierr);
    if (flg) { /* user-defined precondition */
        ierr = PCSetType(pc,PCSHELL); CHKERRA(ierr);
        ierr = PCShellSetApply(pc,MatrixFreePreconditioner,PETSC_NULL);CHKERRA(ierr);
    } else {ierr = PCSetType(pc,PCNONE); CHKERRA(ierr);}
}

ierr = SNESSetFromOptions(snes); CHKERRA(ierr);

/* - - - - -
   Initialize application:
   Store right-hand-side of PDE and exact solution
- - - - - */

xp = 0.0;
for ( i=0; i<n; i++ ) {
    v = 6.0*xp + pow(xp+1.e-12,6.0); /* +1.e-12 is to prevent 0^6 */
    ierr = VecSetValues(F,1,&i,&v,INSERT_VALUES); CHKERRA(ierr);
    xp += h;
}

/* - - - - -
   Evaluate initial guess; then solve nonlinear system
- - - - - */

ierr = VecSet(&pfive,x); CHKERRA(ierr);
ierr = SNESolve(snes,x,&its); CHKERRA(ierr);
PetscPrintf(MPI_COMM_SELF,"number of Newton iterations = %d\n\n", its );

/* - - - - -
   Free work space. All PETSc objects should be destroyed when they
   are no longer needed.
- - - - - */

ierr = VecDestroy(x); CHKERRA(ierr);      ierr = VecDestroy(r); CHKERRA(ierr);
ierr = VecDestroy(F); CHKERRA(ierr);      ierr = MatDestroy(J); CHKERRA(ierr);
ierr = MatDestroy(JPrec); CHKERRA(ierr);  ierr = SNESDestroy(snes); CHKERRA(ierr);
PetscFinalize();

```

```

    return 0;
}
/* ----- */
/*
    FormInitialGuess - Forms initial approximation.

    Input Parameters:
    user - user-defined application context
    X - vector

    Output Parameter:
    X - vector
*/
int FormFunction(SNES snes, Vec x, Vec f, void *dummy)
{
    Scalar *xx, *ff, *FF, d;
    int i, ierr, n;

    ierr = VecGetArray(x, &xx); CHKERRQ(ierr);
    ierr = VecGetArray(f, &ff); CHKERRQ(ierr);
    ierr = VecGetArray((Vec)dummy, &FF); CHKERRQ(ierr);
    ierr = VecGetSize(x, &n); CHKERRQ(ierr);
    d = (double) (n - 1); d = d*d;
    ff[0] = xx[0];
    for (i = 1; i < n - 1; i++) {
        ff[i] = d*(xx[i-1] - 2.0*xx[i] + xx[i+1]) + xx[i]*xx[i] - FF[i];
    }
    ff[n-1] = xx[n-1] - 1.0;
    ierr = VecRestoreArray(x, &xx); CHKERRQ(ierr);
    ierr = VecRestoreArray(f, &ff); CHKERRQ(ierr);
    ierr = VecRestoreArray((Vec)dummy, &FF); CHKERRQ(ierr);
    return 0;
}
/* ----- */
/*
    FormJacobian - This routine demonstrates the use of different
    matrices for the Jacobian and preconditioner

    Input Parameters:
    . snes - the SNES context
    . x - input vector
    . ptr - optional user-defined context, as set by SNESSetJacobian()

    Output Parameters:
    . A - Jacobian matrix
    . B - different preconditioning matrix
    . flag - flag indicating matrix structure
*/
int FormJacobian(SNES snes, Vec x, Mat *jac, Mat *prejac, MatStructure *flag,
                void *dummy)
{
    Scalar *xx, A[3], d;
    int i, n, j[3], ierr;

    ierr = VecGetArray(x, &xx); CHKERRQ(ierr);
    ierr = VecGetSize(x, &n); CHKERRQ(ierr);
    d = (double) (n - 1); d = d*d;

    /* Form Jacobian. Also form a different preconditioning matrix that
       has only the diagonal elements. */

```

```

i = 0; A[0] = 1.0;
ierr = MatSetValues(*jac,1,&i,1,&i,&A[0],INSERT_VALUES); CHKERRQ(ierr);
ierr = MatSetValues(*prejac,1,&i,1,&i,&A[0],INSERT_VALUES); CHKERRQ(ierr);
for ( i=1; i<n-1; i++ ) {
    j[0] = i - 1; j[1] = i;                j[2] = i + 1;
    A[0] = d;    A[1] = -2.0*d + 2.0*xx[i]; A[2] = d;
    ierr = MatSetValues(*jac,1,&i,3,j,A,INSERT_VALUES); CHKERRQ(ierr);
    ierr = MatSetValues(*prejac,1,&i,1,&i,&A[1],INSERT_VALUES); CHKERRQ(ierr);
}
i = n-1; A[0] = 1.0;
ierr = MatSetValues(*jac,1,&i,1,&i,&A[0],INSERT_VALUES); CHKERRQ(ierr);
ierr = MatSetValues(*prejac,1,&i,1,&i,&A[0],INSERT_VALUES); CHKERRQ(ierr);

ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyBegin(*prejac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(*prejac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

ierr = VecRestoreArray(x,&xx); CHKERRQ(ierr);
*flag = SAME_NONZERO_PATTERN;
return 0;
}
/* ----- */
/*
MatrixFreePreconditioner - This routine demonstrates the use of a
user-provided preconditioner. This code implements just the null
preconditioner, which of course is not recommended for general use.

Input Parameters:
. ctx - optional user-defined context, as set by PCShellSetApply()
. x - input vector

Output Parameter:
. y - preconditioned vector
*/
int MatrixFreePreconditioner(void *ctx,Vec x,Vec y)
{
    int ierr;
    ierr = VecCopy(x,y); CHKERRQ(ierr);
    return 0;
}

```

Figure 14: Example of Uniprocessor SNES Code - Both Conventional and Matrix-Free Jacobians

5.6 Finite-Difference Jacobian Approximations

PETSc provides some tools to help approximate the Jacobian matrices efficiently via finite differences. These tools are intended for use in certain situations where one is unable to compute Jacobian matrices analytically, and matrix-free methods do not work because of 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 SNES to use the **SNESDefaultComputeJacobianWithColoring()** routine to compute the Jacobians.

A code fragment that demonstrates this process is given below.

```

ISColoring    iscoloring;
MatFDColoring fdcoloring;
MatStructure  str;

/*
   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,&str,&user);

/*
   Color the matrix, i.e. determine groups of columns that share no common
   rows. These columns in the Jacobian can all be computed simulataneously.
*/
MatGetColoring(J,COLORING_NATURAL,&iscoloring);

/*
   Create the data structure that SNESDefaultComputeJacobianWithColoring() uses
   to compute the actual Jacobians via finite differences.
*/
MatFDColoringCreate(J,iscoloring,&fdcoloring);
MatFDColoringSetFromOptions(fdcoloring);

/*
   Tell SNES to use the routine SNESDefaultComputeJacobianWithColoring()
   to compute Jacobians.
*/
SNESSetJacobian(snes,J,J,SNESDefaultComputeJacobianWithColoring,fdcoloring);
ISColoringDestroy(iscoloring);

```

Of course, we are cheating a bit. If we do not have an analytic formula for computing the Jacobian, 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 grid-based problems (the class of problems for which PETSc is designed) if one knows the stencil used for the nonlinear problem, one can usually fairly easily obtain an estimate of the location of nonzeros in the matrix.

One need not necessarily use the routine `MatGetColoring()` to determine a coloring. For example, if a grid can be colored directly (without using the associated matrix), 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.

For sequential matrices PETSc provides three matrix coloring routines from the MINPACK package [12]. These may be accessed with the command line options

```
-mat_coloring sl, id, or lf
```

Alternatively, one can set a coloring type of `COLORING_SL`, `COLORING_ID`, or `COLORING_LF` when calling `MatGetColoring()`.

As for the matrix-free computation of Jacobians (see Section 5.5), two parameters affect the accuracy of the finite difference Jacobian approximation. These are set with the command

```
ierr = MatFDColoringSetParameters(MatFDColoring fdcoloring,double rerror,double umin);
```

The parameter `rerror` is the square root of the relative error in the function evaluations, e_{rel} ; the default is $1.0e-8$, which assumes that the functions are evaluated to full double-precision accuracy. The second parameter, `umin`, is a bit more involved; its default is $1.0e-8$. 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

$$h = e_{rel} * u_i \quad \text{if } u_i > u_{min}$$
$$h = e_{rel} * u_{min} * \text{sign}(u_i) \quad \text{otherwise.}$$

These parameters may be set from the options database with

```
-mat_fd_coloring_err <err>  
-mat_fd_coloring_umin <umin>
```

Note that these coloring and finite difference Jacobian calculation routines currently work only on sequential routines. Extensions may be forthcoming.

Chapter 6

TS: Scalable ODE Solvers

This chapter introduces an early release of the TS component of PETSc. We encourage users to give us input on needed functionality and the interface. We expect to refine the component based on such feedback.

The TS component provides a framework for the scalable solution of ODEs arising from the discretization of time-dependent PDEs and of steady-state problems using pseudo-timestepping.

Time-Dependent Problems: Consider the ODE

$$u_t = F(u, t),$$

where u is a finite-dimensional vector, usually obtained from discretizing a PDE with finite differences, finite elements, and so forth. For example, discretizing the heat equation

$$u_t = u_{xx}$$

with centered finite differences results in

$$(u_i)_t = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

The TS component provides code to solve these equations (currently using the forward or backward Euler method), in a clean and easy manner, where the user need only provide code for the evaluation of $F(u, t)$ and (optionally) its associated Jacobian matrix.

Steady-State Problems: In addition, 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

$$F(u) = 0,$$

we can use pseudo-transient continuation by solving

$$u_t = F(u).$$

By using time differencing with the backward Euler method, we obtain

$$\frac{u^{n+1} - u^n}{dt^n} = F(u^{n+1}).$$

More generally we can consider a diagonal matrix Dt^n that has a pseudo-timestep for each node point to obtain the series of nonlinear equations

$$Dt^{n-1}(u^{n+1} - u^n) = F(u^{n+1}).$$

For this problem the user must provide $F(u)$ and the diagonal matrix Dt^n (or optionally, if the timestep is position independent, a scalar timestep); in addition, the Jacobian of $F(u)$ may be provided.

6.1 Basic Usage

The user first creates a TS object with the command

```
ierr = int TSCreate(MPI_Comm comm,TSProblemType problemtype,TS *ts);
```

The `TSProblemType` is one of `TS_LINEAR` or `TS_NONLINEAR`, to indicate whether $F(u, t)$ is given by a matrix A , or $A(t)$, or a function $F(u, t)$.

One can set the solution method with the routine

```
ierr = TSSetType(TS ts,TSType type);
```

Currently supported types are `TS_EULER`, `TS_BEULER`, and `TS_PSEUDO` or the command line option `-ts_type euler, beuler`.

Set the initial time and timestep with the command

```
ierr = TSSetInitialTimeStep(TS ts,double time,double dt);
```

One can change the timestep with the command

```
ierr = TSSetTimeStep(TS ts,double dt);
```

One can determine the current timestep with the routine

```
ierr = TSGetTimeStep(TS ts,double* dt);
```

Here, “current” refers to the timestep being used to attempt to promote the solution from 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 command

```
ierr = TSSetDuration(TS ts,int maxsteps,double maxtime);
```

One sets up the timestep context with

```
ierr = TSSetUp(TS ts);
```

destroys it with

```
ierr = TSDestroy(TS ts);
```

and views it with

```
ierr = TSView(TS ts,Viewer viewer);
```

6.1.1 Solving Time-dependent Problems

To set up TS for solving an ODE, one must set the following:

- Solution:

```
ierr = TSSetSolution(TS ts, Vec initialsolution);
```

The vector `initialsolution` should contain the “initial conditions” for the PDE.

- Function:

- For linear functions (solved with implicit timestepping), the user must call

```
ierr = TSSetRHSMatrix(TS ts,Mat A, Mat B,int (*f)(TS,double,Mat*,Mat*,
MatStructure*,void*),void *fP);
```

The matrix `B` (although usually the same as `A`) allows one to provide a different matrix to be used in the construction of the preconditioner. The function `f` is used to form the matrices `A` and `B` at each timestep if the matrices are time dependent. If the matrix does not depend on time, the user should pass in `PETSC_NULL` for `f`. The variable `fP` allows users to pass in an application context that is passed to the `f()` function whenever it is called, as the final argument. The user must provide the matrices `A` and `B`; if they have the right-hand side only as a linear function, they must construct a `MatShell` matrix. Note that this is the same interface as that for `SNESSetJacobian()`.

- For nonlinear problems (or linear problems solved using explicit timestepping methods) the user passes the function with the routine

```
ierr = TSSetRHSFunction(TS ts,int (*f)(TS,double,Vec,Vec,void*),void *fP);
```

The arguments to the function `f()` are the timestep context, the current time, the input for the function, the output for the function, and the (optional) user-provided context variable `fP`.

- Jacobian: For nonlinear problems the user must also provide the (approximate) Jacobian matrix of $F(u, t)$ and a function to compute it at each Newton iteration. This is done with the command

```
ierr = TSSetRHSJacobian(TS ts, Mat A, Mat B, int (*f)(TS, double, Vec, Mat*, Mat*,
MatStructure*, void*), void *fP);
```

The arguments for the function `f()` are the timestep context, the current time, the location where the Jacobian is to be computed, the Jacobian matrix, an alternative approximate Jacobian matrix used as a 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 is used, the user must create a `MatShell` matrix. Again, note the similarity to `SNESetJacobian()`.

6.1.2 Solving Steady-State Problems with Pseudo-Timestepping

For solving steady-state problems with pseudo-timestepping, one proceeds as follows.

- Provide the function $F(u)$ with the routine

```
ierr = TSSetRHSFunction(TS ts, int (*f)(TS, double, Vec, Vec, void*), void *fP);
```

The arguments to the function `f()` are the timestep context, the current time, the input for the function, the output for the function and the (optional) user-provided context variable `fP`.

- Provide the (approximate) Jacobian matrix of $F(u, t)$ and a function to compute it at each Newton iteration. This is done with the command

```
ierr = TSSetRHSJacobian(TS ts, Mat A, Mat B, int (*f)(TS, double, Vec, Mat*, Mat*,
MatStructure*, void*), void *fP);
```

The arguments for the function `f()` are the timestep context, the current time, the location where the Jacobian is to be computed, the Jacobian matrix, an alternative approximate Jacobian matrix used as a preconditioner, and the optional user-provided context, passed in as `fP`. The user must provide the Jacobian as a matrix; thus, if one is 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 whether one is using a constant timestep over the entire grid or one that varies with location.

- For location-independent pseudo-timestepping, one uses the routine

```
ierr = TSPseudoSetTimeStep(TS ts, int(*dt)(TS, double*, void*), void* dtctx);
```

The function `dt` is a user-provided function that computes the next pseudo-timestep. As a default one can use `TSPseudoDefaultTimeStep(TS, double*, void*)` for `dt`. This routine updates the pseudo-timestep with one of two strategies: the default

$$dt^n = dt_increment * dt^{n-1} * \frac{\|F(u^{n-1})\|}{\|F(u^n)\|}$$

or the alternative

$$dt^n = dt_increment * dt^0 * \frac{\|F(u^0)\|}{\|F(u^n)\|},$$

which can be set with the call

```
ierr = TSPseudoIncrementDtFromInitialDt(TS ts);
```

or the option `-ts_pseudo_increment_dt_from_initial_dt`. The value `dt_increment` is by default 1.1, but can be reset with the call

```
ierr = TSPseudoSetTimeStepIncrement(TS ts, double inc);
```

or the option `-ts_pseudo_increment <inc>`.

- For location-dependent pseudo-timestepping, the interface function has not yet been created.

Chapter 7

Advanced Features of Matrices and Solvers

This chapter introduces additional features of the PETSc matrices and solvers. Since most PETSc users should not need to use these features, we recommend skipping this chapter during an initial reading.

7.1 Matrix Factorization

Normally, PETSc users will access the matrix solvers through the SLES interface, as discussed in Chapter 4, but the underlying factorization and triangular solve routines are also directly accessible to the user.

The LU and Cholesky matrix factorizations are split into two or three stages depending on the user's needs. The first stage is to calculate an ordering for the matrix. The ordering generally is done to reduce fill in a sparse factorization; it does not make much sense for a dense matrix.

```
ierr = MatGetReordering(Mat matrix, MatReordering type, IS* rowperm, IS* colperm);
```

The currently available alternatives for the ordering `type` are

- `ORDER_NATURAL` - Natural
- `ORDER_ND` - Nested Dissection
- `ORDER_1WD` - One-Way Dissection
- `ORDER_RCM` - Reverse Cuthill-McKee
- `ORDER_QMD` - Quotient Minimum Degree

These orderings can also be set through the options database by specifying one of the following: `-mat_order natural`, `-mat_order nd`, `-mat_order 1wd`, `-mat_order rcm`, `-mat_order qmd`. Certain matrix formats may support only a subset of these; more options may be added. Check the man pages for up-to-date information. All of these orderings are symmetric at the moment; ordering routines that are not symmetric may be added. Currently we support reorderings only for sequential matrices.

Users can add their own reordering routines by providing a function with the calling sequence

```
int reorder(Mat A, MatReordering 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 reordering routine. The user registers the reordering routine with the command

```
ierr = MatReorderingRegister(MatReordering inname, MatReordering *name, char *sname,  
                             int (*reorder)(Mat, MatReordering, IS*, IS*));
```

The input argument `*sname` is a string of the user's choice; `iname` is either an ordering defined in `mat.h` or `ORDER_NEW`, to indicate one is introducing a new ordering; and the output argument `*name` is the registration number returned to 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 `-mat_order sname`. If reordering directly, the user should provide the `name` as the second input argument of `MatGetReordering()`.

The following routines perform complete, in-place, symbolic, and numerical factorizations for symmetric and nonsymmetric matrices, respectively:

```
ierr = MatCholeskyFactor(Mat matrix, IS permutation, double pf);
ierr = MatLUFactor(Mat matrix, IS rowpermutation, IS columnpermutation, double pf);
```

The argument `pf` ≥ 1 is the predicted fill expected in the factored matrix, as a ratio of the original fill. For example, `pf=2.0` would indicate that one expects the factored matrix to have twice as many nonzeros as the original.

For sparse matrices it is very unlikely that the factorization is actually done in-place. More likely, new space is allocated for the factored matrix and the old space deallocated, but to the user it appears in-place because the factored matrix replaces the unfactored matrix.

The two factorization stages can also be performed separately, by using the out-of-place mode:

```
ierr = MatCholeskyFactorSymbolic(Mat matrix, IS perm, double pf, Mat *result);
ierr = MatLUFactorSymbolic(Mat matrix, IS rowperm, IS colperm, double pf, Mat *result);
ierr = MatCholeskyFactorNumeric(Mat matrix, Mat *result);
ierr = MatLUFactorNumeric(Mat matrix, Mat *result);
```

In this case, the contents of the matrix `result` is undefined between the symbolic and numeric factorization stages. It is possible to reuse the symbolic factorization. For the second and succeeding factorizations, one simply calls the numerical factorization with a new input `matrix` and the *same* factored `result` matrix. It is *essential* that the new input matrix have exactly the same nonzero structure as the original factored matrix. (The numerical factorization merely overwrites the numerical values in the factored matrix and does not disturb the symbolic portion, thus enabling reuse of the symbolic phase.) In general, calling `XXXFactorSymbolic` with a dense matrix will do nothing except allocate the new matrix; the `XXXFactorNumeric` routines will do all of the work.

Why provide the plain `XXXfactor` routines when one could simply call the two-stage routines? The answer is that if one desires in-place factorization of a sparse matrix, the intermediate stage between the symbolic and numeric phases cannot be stored in a `result` matrix, and it does not make sense to store the intermediate values inside the original matrix that is being transformed. We originally made the combined factor routines do either in-place or out-of-place factorization, but then decided that this approach was not needed and could easily lead to confusion.

We do not currently support sparse matrix factorization with pivoting for numerical stability. This is because trying to both reduce fill and do pivoting can become quite complicated. Instead, we provide a poor stepchild substitute. After one has obtained a reordering, with `MatGetRordering(Mat A, MatOrdering type, IS *row, IS *col)` one may call

```
ierr = MatReorderForNonzeroDiagonal(Mat A, double tol, IS row, IS col);
```

which will try to reorder the columns to ensure that no values along the diagonal are smaller than `tol` in an absolute value. If small values are detected and corrected for, a nonsymmetric permutation of the rows and columns will result. This is not guaranteed to work, but may help if one was simply unlucky in the original ordering. When using the SLES solver interface the options `-pc_ilu_nonzeros_along_diagonal` and `-pc_ilu_nonzeros_along_diagonal` may be used.

Once a matrix has been factored, it is natural to solve linear systems. The following four routines enable this process:

```
ierr = MatSolve(Mat A, Vec x, Vec y);
ierr = MatSolveTrans(Mat A, Vec x, Vec y);
ierr = MatSolveAdd(Mat A, Vec x, Vec y, Vec w);
ierr = MatSolveTransAdd(Mat A, Vec x, Vec y, Vec w);
```

The 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 SLES solvers introduced in the next chapter rather than using these factorization and solve routines directly.

7.2 Unimportant Details of KSP

Again, virtually all users should use KSP through the SLES interface and, thus, will not need to know the details that follow.

It is possible to generate a Krylov subspace context with the command

```
ierr = KSPCreate(MPI_Comm comm,KSP *kps);
```

Before using the Krylov context, one must set the matrix-vector multiplication routine and the preconditioner with the commands

```
ierr = PCSetOperators(PC pc,Mat mat,Mat pmat,MatStructure flag);
ierr = KSPSetPC(KSP ksp,PC pc);
```

In addition, the KSP solver must be initialized with

```
ierr = KSPSetUp(KSP ksp);
```

Solving a linear system is done with the command

```
ierr = KSPSolve(KSP ksp,int *its);
```

Finally, the KSP context should be destroyed with

```
ierr = KSPDestroy(KSP ksp);
```

It may seem strange to put the matrix in the preconditioner rather than directly in the KSP; this decision was the result of much agonizing. The reason is that for SSOR with Eisenstat's trick, and certain other preconditioners, the preconditioner has to change the matrix-vector multiply. This procedure could not be done cleanly if the matrix were stashed in the KSP context that PC cannot access.

Any preconditioner can supply not only the preconditioner, but also a routine that essentially performs a complete Richardson step. The reason for this is mainly SOR. To use SOR in the Richardson framework, that is,

$$u^{n+1} = u^n + B(f - Au^n),$$

is much more expensive than just updating the values. With this addition it is reasonable to state that *all* our iterative methods are obtained by combining a preconditioner from the PC component with a Krylov method from the KSP component. This strategy makes things much simpler conceptually, so (we hope) clean code will result. *Note:* We had this idea already implicitly in older versions of SLES, but, for instance, just doing Gauss-Seidel with Richardson in old SLES was much more expensive than it had to be. With PETSc 2.0 this should not be a problem.

7.3 Unimportant Details of PC

Most users will obtain their preconditioner contexts from the SLES context with the command `SLESGetPC()`. 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

```
ierr = PCCreate(MPI_Comm comm,PC *pc);
```

The routine

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

sets the preconditioner method to be used. The two routines

```
ierr = PCSetOperators(PC pc,Mat mat,Mat pmat,MatStructure flag);
ierr = PCSetVector(PC pc,Vec vec);
```

set the matrices and type of vector that are to be used with the preconditioner. The `vec` argument is needed by the PC routines to determine the format of the vectors. The routine

```
ierr = PCGetOperators(PC pc,Mat *mat,Mat *pmat,MatStructure *flag);
```

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

```
ierr = PCApply(PC pc,Vec x,Vec y);
ierr = PCApplyTrans(PC pc,Vec x,Vec y);
```

In particular, for a preconditioner matrix, **B**, that has been set via `PCSetOperators(pc,A,B,flag)`, 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

```
ierr = PCApplyBAorAB(PC pc,int right,Vec x,Vec y,Vec work,int its);
ierr = PCApplyBAorABTrans(PC pc,int right,Vec x,Vec y,Vec work,int its);
ierr = PCApplyRichardson(PC pc,Vec x,Vec y,Vec work,int its);
```

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 integer `right` is zero or one. 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

```
ierr = PCDestroy(PC pc);
```


Chapter 8

Graphics

PETSc graphics components are not intended to compete with high-quality graphics packages. Instead, they are 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 in PETSc, he or she should send a message to petsc-maint@mcs.anl.gov, and we will see whether it is reasonable to try to provide direct interfaces.

8.1 Windows as Viewers

For drawing predefined PETSc objects such as matrices and vectors, one must first create a viewer using the command

```
ierr = ViewerDrawOpenX(MPI_Comm comm, char *display, char *title, int x, int y, int w,
                        int h, Viewer *viewer);
```

This viewer may be passed to any of the `XXXView()` routines. To draw into the viewer, one must obtain the `Draw` object with the command

```
ierr = ViewerDrawGetDraw(Viewer viewer, Draw *draw);
```

Then one can call any of the `DrawXXX` commands on the `draw` object. If one obtains the `draw` object in this manner, one does not call the `DrawOpenX()` command discussed below.

Predefined viewers, `VIEWER_DRAWX_WORLD` and `VIEWER_DRAWX_SELF`, may be used at any time. Their initial use will cause the appropriate window to be created.

If the colormap on one's machine is incorrect, so colors in contour plots and so on are incorrect, one can use the option `-draw_x_private_colormap` to have PETSc use a separate colormap for its windows. This will correct the color problem, but one will get flashing of colors as one moves the mouse between the PETSc windows and other windows. The user can also try stopping programs (like Netscape) that change the default colormap.

8.2 Simple Drawing

One can open a window under the X11 Window System with the command

```
ierr = DrawOpenX(MPI_Comm comm, char *display, char *title, int x, int y, int w,
                  int h, Draw *win);
```

All drawing routines are done relative to the windows 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

```
ierr = DrawSetCoordinates(Draw win, double xl, double yl, double xr, double 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

```
ierr = DrawSetViewPort(Draw win, double xl, double yl, double xr, double yr);
```

These arguments, which indicate the fraction of the window in which the drawing should be done, must satisfy $0 \leq x1 \leq xr \leq 1$ and $0 \leq y1 \leq yr \leq 1$.

To draw a line, one uses the command

```
ierr = DrawLine(Draw win,double x1,double y1,double xr,double yr,int cl);
```

The argument `cl` indicates the color of the line.

To ensure that all graphics actually have been displayed, one should use the command

```
ierr = DrawFlush(Draw win);
```

When displaying by using double buffering, which is set with the command

```
ierr = DrawSetDoubleBuffer(Draw win);
```

all processors must call

```
ierr = DrawSyncFlush(Draw 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 `DrawPause()`. A time of `-1` indicates that the application should pause until receiving mouse input from the user.

Text can be drawn with either of the two commands

```
ierr = DrawText(Draw win,double x,double y,int color,char *text);
```

```
ierr = DrawTextVertical(Draw win,double x,double y,int color,char *text);
```

The user can set the text font size or determine it with the commands

```
ierr = DrawTextSetSize(Draw win,double width,double height);
```

```
ierr = DrawTextGetSize(Draw win,double *width,double *height);
```

8.3 Line Graphs

PETSc includes a set of routines for manipulating simple two-dimensional graphs. These routines, which begin with `DrawAxisDraw()`, 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 program in Figure 15, line graphs are created with the command

```
ierr = DrawLGCreate(Draw win,int ncurves,DrawLG *ctx);
```

The argument `ncurves` indicates how many curves are to be drawn. Points can be added to each of the curves with the command

```
ierr = DrawLGAddPoint(DrawLG ctx,double *x,double *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

```
ierr = DrawLGAddPoints(DrawLG ctx,int n,double **x,double **y);
```

The line graph is drawn (or redrawn) with the command

```
ierr = DrawLGDraw(DrawLG ctx);
```

A line graph that is no longer needed can be destroyed with the command

```
ierr = DrawLGDestroy(DrawLG ctx);
```

To plot new curves, one can reset a linegraph with the command

```
ierr = DrawLGReset(DrawLG 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

```
ierr = DrawLGSetLimits(DrawLG ctx,double xmin,double xmax,double ymin,double ymax);
```

It is also possible to change the display of the axes and to label on them. This procedure is done by first obtaining the axes context with the command

```
ierr = DrawLGGetAxis(DrawLG ctx,DrawAxis *axis);
```

One can set the axes' colors and labels, respectively, by using the commands

```
ierr = DrawAxisSetColors(DrawAxis axis,int axis_lines,int ticks,int text);
```

```
ierr = DrawAxisSetLabels(DrawAxis axis,char *top,char *x,char *y);
```

```

#ifndef lint
static char vcid[] = "$Id: ex3.c,v 1.24 1996/03/19 21:28:29 bsmith Exp $";
#endif

static char help[] = "Plots a simple line graph\n";

#include "draw.h"
#include <math.h>

int main(int argc, char **argv)
{
    Draw      draw;
    DrawLG    lg;
    DrawAxis  axis;
    int       n = 20, i, ierr, x = 0, y = 0, width = 300, height = 300, flg;
    char      *xlabel, *ylabel, *toplabel;
    double    xd, yd;

    xlabel = "X-axis Label"; toplevel = "Top Label"; ylabel = "Y-axis Label";

    PetscInitialize(&argc, &argv, (char *) 0, help);
    OptionsGetInt(PETSC_NULL, "-width", &width, &flg);
    OptionsGetInt(0, "-height", &height, &flg);
    OptionsGetInt(PETSC_NULL, "-n", &n, &flg);
    OptionsHasName(PETSC_NULL, "-nolabels", &flg);
    if (flg) {
        xlabel = (char *) 0; toplevel = (char *) 0;
    }
    ierr = DrawOpenX(MPI_COMM_SELF, 0, "Title", x, y, width, height, &draw); CHKERRA(ierr);
    ierr = DrawLGCreate(draw, 1, &lg); CHKERRA(ierr);
    ierr = DrawLGGetAxis(lg, &axis); CHKERRA(ierr);
    ierr = DrawAxisSetColors(axis, DRAW_BLACK, DRAW_RED, DRAW_BLUE); CHKERRA(ierr);
    ierr = DrawAxisSetLabels(axis, toplevel, xlabel, ylabel); CHKERRA(ierr);

    for ( i=0; i<n ; i++ ) {
        xd = (double)( i - 5 ); yd = xd*xd;
        ierr = DrawLGAddPoint(lg, &xd, &yd); CHKERRA(ierr);
    }
    ierr = DrawLGIndicateDataPoints(lg); CHKERRA(ierr);
    ierr = DrawLGDraw(lg); CHKERRA(ierr);
    ierr = DrawFlush(draw); CHKERRA(ierr); PetscSleep(2);

    ierr = DrawLGDestroy(lg); CHKERRA(ierr);
    ierr = DrawDestroy(draw); CHKERRA(ierr);
    PetscFinalize();
    return 0;
}

```

Figure 15: Example of Drawing Plots

It is possible to turn off all graphics with the option `-nox`. This will prevent any windows from being open or any drawing actions to be done. This is useful for timing or for running large jobs when the graphics overhead is too large.

8.4 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_xmonitor` and `-snes_xmonitor`.

See also Sections 4.3.3 and 5.3.2.

The two functions used are `KSPLGMonitor()` and `KSPLGMonitorCreate()` . These can easily be modified to serve specialized needs.

8.5 Other Graphical Output Types

PETSc contains some code to generate output in Postscript and VRML (Virtual Reality Modeling Language). This code is currently undergoing revision but is available for the adventurous.

Chapter 9

PETSc Fortran Users

Most of the functionality of PETSc can be obtained by people who program purely in Fortran. Note, however, that we recommend the use of C and/or C++ because these languages contain several extremely powerful concepts that the Fortran 77/90 family does not.

Since Fortran 77 does not provide type checking of routine input/output parameters, we find that many errors encountered within PETSc Fortran 77 programs result from accidentally using incorrect calling sequences. Such mistakes are immediately detected during compilation when using C/C++. Thus, using a mixture of C/C++ and Fortran often works well for programmers who wish to employ Fortran for the core numerical routines within their applications. In particular, one can effectively write PETSc driver routines in C/C++, thereby preserving flexibility within the program, and still use Fortran when desired for underlying numerical computations.

9.1 Differences between PETSc Interfaces for C and Fortran

Only a few differences exist between the C and Fortran PETSc interfaces, all of which are due to limitations in Fortran syntax. All Fortran routines have the same names as the corresponding C versions, and PETSc command line options are fully supported on most machines. The routine arguments follow the usual Fortran conventions; the user need not worry about passing pointers or values. The calling sequences for the Fortran version are in most cases identical to the C version, except for the error checking variable discussed in Section 9.1.2 and a few routines listed in Section 9.1.10. Note that use of the PETSc Fortran interface requires first compiling the interface library, which is discussed in Section 9.1.9.

9.1.1 Include Files

PETSc Fortran users have two choices for including the PETSc header files.

Recommended Approach: In the first approach, the Fortran include files for PETSc are located in the directory `$(PETSC_DIR)/include/FINCLUDE` and should be used via statements such as the following:

```
#include "include/FINCLUDE/includefile.h"
```

Since one must be very careful to include each file no more than once in a Fortran routine, application programmers must manually include each file needed for the various PETSc components within their program. This approach differs from the PETSc C/C++ interface, where the user need only include the highest level file, for example, `snes.h`, which then automatically includes all of the required lower level files. As shown in the examples of Section 9.2, in Fortran one must explicitly list *each* of the include files. If using this approach, one must employ the Fortran file suffix `.F` rather than `.f`. This convention enables use of the CPP preprocessor, which allows the use of the `#include` statements that define PETSc objects and variables. (Familiarity with the CPP preprocessor is not needed for writing PETSc Fortran code; one can simply begin by copying a PETSc Fortran example and its corresponding makefile.)

Alternative Approach: If working with `.f` files is absolutely essential (perhaps as part of a heritage code), the conventional Fortran style include statement can be employed. The weakness of this approach is that either the complete path of the include file must be hardwired with a statement such as

`include '/home/username/petsc/include/finclude/includefile.h'`
or a link must be established in the directory containing the Fortran source file to the file
`ln -s /home/username/petsc/include/finclude/includefile.h includefile.h`
Some Fortran compilers will accept a `-I<directory>`, but depending on the Fortran compiler, they may use the `-I` list only for the `#include` style of include. In addition, the user must declare all PETSc objects as `integer` rather than by their name. For example, declarations within Fortran `.F` files have the form

```
SLES      solver
Mat       A, B
Vec       x, y
integer i
```

while the analogous statements within `.f` files are

```
integer solver
integer A, B
integer x, y
integer i
```

9.1.2 Error Checking

In the Fortran version, each PETSc routine has as its final argument an integer error variable, in contrast to the C convention of providing the error variable as the routine's return value. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For example, the Fortran and C variants of `SLESSolve()` are given, respectively, below, where `ierr` denotes the error variable:

```
call SLESSolve(SLES sles,Vec b,Vec x,int its,int ierr)
ierr = SLESSolve(SLES sles,Vec b,Vec x,int *its);
```

Fortran programmers using the `.F` file suffix, as discussed in Section 9.1.1, can check these error codes with `CHKERRA(ierr)`, which terminates all process when an error is encountered. Likewise, one can set error codes within Fortran programs by using `SETERRA(ierr,p,'')`, which again terminates all processes upon detection of an error. Note that complete error tracebacks with `CHKERRQ()` and `SETERRQ()`, as described in Section 1.3 for C routines, are *not* directly supported for Fortran routines; however, Fortran programmers can easily use the error codes in writing their own tracebacks. For example, one could use code such as the following:

```
call SLESSolve(sles,x,y,ierr)
if ( ierr .ne. 0 ) then
    print*, 'Error in routine ...'
    return
endif
```

Note that users of the Fortran `.f` suffix *cannot* employ the macros `SETERRA()` and `CHKERRA()`.

9.1.3 Array Arguments

Since Fortran does not allow arrays to be returned in routine arguments, all PETSc routines that return arrays, such as `VecGetArray()`, `MatGetArray()`, `ISGetIndices()`, and `DAGetGlobalIndices()` are defined slightly differently in Fortran than in C. Instead of returning the array itself, these routines accept as input a user-specified array of dimension one and return an integer index to the actual array used for data storage within PETSc. The Fortran interface for several routines is as follows:

```
double precision xx_v(1), aa_v(1)
integer          ss_v(1), dd_v(1), dd_i, ss_i, xx_i, aa_i, ierr, nloc
Vec x
Mat A
IS s
DA d

call VecGetArray(x,xx_v,xx_i,ierr)
call MatGetArray(A,aa_v,aa_i,ierr)
call ISGetIndices(s,ss_v,ss_i,ierr)
call DAGetGlobalIndices(d,nloc,dd_v,dd_i,ierr)
```

To access array elements directly, both the user-specified array and the integer index *must* then be used together. For example, the following Fortran program fragment illustrates directly setting the values of a vector array instead of using `VecSetValues()`. Note the (optional) use of the preprocessor `#define` statement to enable array manipulations in the conventional Fortran manner.

```
#define xx_a(ib)  xx_v(xx_i + (ib))

double precision xx_v(1)
integer          xx_i, i, ierr, n
Vec              x
call VecGetArray(x,xx_v,xx_i,ierr)
call VecGetLocalSize(x,n,ierr)
do 10, i=1,n
    xx_a(i) = 3*i + 1
10 continue
call VecRestoreArray(x,xx_v,xx_i,ierr)
```

Figure 17 contains an example of using `VecGetArray()` within a Fortran routine.

Note: If using `VecGetArray()`, `MatGetArray()`, `ISGetIndices()`, or `DAGetGlobalIndices()` from Fortran, the user *must not* compile the Fortran code with options to check for “array entries out of bounds” (e.g., on the IBM RS/6000 this is done with the `-C` compiler option, so never use the `-C` option with this).

9.1.4 Calling Fortran Routines from C (and C Routines from Fortran)

Since the use of both Fortran and C routines is sometimes needed in application codes, we provide two PETSc commands to facilitate passing PETSc objects (such as `Mat` and `SLES`) between the two languages. These routines *must* be called within any C/C++ routines that pass/receive PETSc objects to/from Fortran routines to ensure that the objects are properly handled, since Fortran treats PETSc objects simply as integers.

To pass a PETSc object from a C routine to a Fortran routine, one must first convert the C pointer to a Fortran integer with `PetscCObjectToFortranObject(PetscObject,int *)`, for example,

```
int fmat;
Mat mat;
....
ierr = PetscCObjectToFortranObject(mat,&fmat);
fortranroutine(&fmat,...other arguments);
```

The Fortran routine can then directly use the received object. When calling C from Fortran, the user should perform the pointer conversion within the C routine using `PetscFortranObjectToCObject(int,PetscObject*)`, for example,

```
int cfunction(int *fmat,...)
Mat mat;
ierr = PetscFortranObjectToCObject(*fmat,&mat);
```

See `$(PETSC_DIR)/src/vec/examples/tests/ex24.c` for a complete example demonstrating this interface. Note that the pointer conversion is always done in the C/C++ routines, but not in the Fortran routines.

Different machines have different methods of naming Fortran routines called from C (or C routines called from Fortran). Most Fortran compilers change all the capital letters in Fortran routines to small. On some machines, the Fortran compiler appends an underscore to the end of each Fortran routine name; for example, the Fortran routine `Dabsc()` would be called from C with `dabsc_()`. Other machines change all the letters in Fortran routine names to capitals.

PETSc provides two macros (defined in C/C++) to help write portable code that mixes C/C++ and Fortran. They are `HAVE_FORTTRAN_UNDERSCORE` and `HAVE_FORTTRAN_CAPS`, which are defined in the file `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base.site`. The macros are used, for example, as follows:

```
#if defined(HAVE_FORTTRAN_CAPS)
#define dabsc_ DABSC
#elif !defined(HAVE_FORTTRAN_UNDERSCORE)
#define dabsc_ dabsc
```

```
#endif
.....
dabsc_(&n,x,y); /* call the Fortran function */
```

9.1.5 Passing Null Pointers

In several PETSc C functions, one has the option of passing a 0 (null) argument (for example, the fifth argument of `MatCreateSeqAIJ()`). From Fortran, users *must* pass `PETSC_NULL` to indicate a null argument; passing 0 from Fortran will crash the code. When passing character strings as routine arguments, Fortran programmers should be sure to pass `PETSC_NULL_CHARACTER` whenever a null character is desired. Note that the C convention of passing `PETSC_NULL` (or 0) *cannot* be used. For example, when no options prefix is desired in the routine `OptionsGetInt()`, one must use the following command in Fortran:

```
call OptionsGetInt(PETSC_NULL_CHARACTER, '-name', N, flg, ierr)
```

This Fortran requirement is inconsistent with C, where the user can employ `PETSC_NULL` for all null arguments. The reason for this difference is that some machines (e.g., the Cray T3D) handle strings in Fortran in an unusual manner, so that the only way we can provide support for portable code is to require the use of `PETSC_NULL_CHARACTER`.

9.1.6 Duplicating Multiple Vectors

The Fortran interface to `VecDuplicateVecs()` differs slightly from the C/C++ variant because Fortran does not allow arrays to be returned in routine arguments. To create `n` vectors of the same format as an existing vector, the user must declare a vector array, `v_new` of size `n`. Then, after `VecDuplicateVecs()` has been called, `v_new` will contain (pointers to) the new PETSc vector objects. When finished with the vectors, the user should destroy them by calling `VecDestroyVectors()`. For example, the following code fragment duplicates `v_old` to form two new vectors, `v_new(1)` and `v_new(2)`.

```
Vec      v_old, v_new(2)
integer ierr
Scalar  alpha
call VecDuplicateVecs(v_old,2,v_new,ierr)
alpha = 4.3
call VecSet(alpha,v(1),ierr)
alpha = 6.0
call VecSet(alpha,v(2),ierr)
call VecDestroyVecs(v_new,2,ierr)
```

9.1.7 Matrix and Vector Indices

All matrices and vectors in PETSc use zero-based indexing, regardless of whether C or Fortran is being used. The interface routines, such as `MatSetValues()` and `VecSetValues()`, always use zero indexing. See Section 3.2 for further details.

9.1.8 Setting Routines

When a routine is set from within a Fortran program by a routine such as `KSPSetConvergenceTest()`, that routine is assumed to be a Fortran routine. Likewise, when a routine is set from within a C program, that routine is assumed to be written in C.

9.1.9 Compiling and Linking Fortran Programs

Before any PETSc Fortran programs are compiled, the Fortran interface library must be built from the PETSc home directory, `$(PETSC_DIR)`, with the command

```
make BOPT=[g,0,0pg] fortran
```


The Fortran interface library for a particular architecture and `BOPT` value resides in the same directory as the other PETSc libraries, as given by `$(PETSC_DIR)/lib/lib$(BOPT)/$(PETSC_ARCH)/libpetscfortran.a`.

Figure 22 shows a sample makefile that can be used for PETSc programs. In this makefile, one can compile and run a debugging version of the Fortran program `ex3.F` with the actions `make BOPT=g ex3` and `make runex3`, respectively. The compilation command is restated below:

```
ex3: ex3.o
    -$(FLINKER) -o ex3 ex3.o $(PETSC_FORTRAN_LIB) $(PETSC_LIB)
    $(RM) ex3.o
```

Note that the PETSc Fortran interface library, given by `$(PETSC_FORTRAN_LIB)`, *must* precede the base PETSc libraries, given by `$(PETSC_LIB)`, on the link line.

9.1.10 Routines with Different Fortran Interfaces

The following Fortran routines differ slightly from their C counterparts; see the man pages and previous discussion in this chapter for details:

- `PetscInitialize(char *filename,int ierr)`
- `PetscError(int errno,char *message,int ierr)`
- `VecGetArray(), MatGetArray()`
- `ISGetIndices(), DAGetGlobalIndices()`
- `VecDuplicateVecs(), VecDestroyVecs()`

The following functions are not supported in Fortran:

- `PetscBinaryRead(), PetscBinaryWrite()`
- `PetscFClose(), PetscFOpen(), PetscFPrintf(), PetscPrintf()`
- `PetscPopErrorHandler(), PetscPushErrorHandler()`
- `PLogInfo()`
- `PetscSetDebugger()`
- `VecGetArrays(), VecRestoreArrays()`
- `ViewerASCIIGetPointer(), ViewerBinaryGetDescriptor()`
- `ViewerStringOpen(), ViewerStringSPrintf()`

9.2 Sample Fortran 77 Programs

Sample programs that illustrate the PETSc interface for Fortran are given in Figures 16 - 19, corresponding to `$(PETSC_DIR)/src/vec/examples/tests/ex19.F`, `$(PETSC_DIR)/src/vec/examples/tutorials/ex4f.F`, `$(PETSC_DIR)/src/dw/examples/tests/ex5.F`, and `$(PETSC_DIR)/src/snes/examples/ex1f.F`, respectively. We also refer Fortran programmers to the C examples listed throughout the manual, since PETSc usage within the two languages differs only slightly.

```
C    "$Id: ex19.F,v 1.25 1996/08/27 20:24:13 curfman Exp $";

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
C
C This example demonstrates basic use of the PETSc Fortran interface
C to vectors.
```

```

C
integer      n, ierr, flg
Scalar       one, two, three, dot
Double       norm, rdot
Vec          x, y, w

n      = 20
one    = 1.0
two    = 2.0
three  = 3.0

call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
call OptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)

C Create a vector, then duplicate it
call VecCreate(MPI_COMM_WORLD,n,x,ierr)
call VecDuplicate(x,y,ierr)
call VecDuplicate(x,w,ierr)

call VecSet(one,x,ierr)
call VecSet(two,y,ierr)

call VecDot(x,y,dot,ierr)
rdot = PetscReal(dot)
write(6,100) rdot
100 format('Result of inner product ',f10.4)

call VecScale(two,x,ierr)
call VecNorm(x,NORM_2,norm,ierr)
write(6,110) norm
110 format('Result of scaling ',f10.4)

call VecCopy(x,w,ierr)
call VecNorm(w,NORM_2,norm,ierr)
write(6,120) norm
120 format('Result of copy ',f10.4)

call VecAXPY(three,x,y,ierr)
call VecNorm(y,NORM_2,norm,ierr)
write(6,130) norm
130 format('Result of axpy ',f10.4)

call VecDestroy(x,ierr)
call VecDestroy(y,ierr)
call VecDestroy(w,ierr)
call PetscFinalize(ierr)
stop
end

```

Figure 16: Sample Fortran Program: Using PETSc Vectors

```

C      "$Id: ex4f.F,v 1.16 1996/11/27 22:51:13 bsmith Exp $";

C Description: Illustrates the use of VecSetValues() to set
C multiple values at once; demonstrates VecGetArray().
C

```

```

C/*T
C  Concepts: Vectors^Assembling vectors; Using vector arrays;
C  Routines: VecCreateSeq(); VecDuplicate(); VecSetValues(); VecView();
C  Routines: VecCopy(); VecView(); VecGetArray(); VecRestoreArray();
C  Routines: VecAssemblyBegin(); VecAssemblyEnd(); VecDestroy();
C  Processors: 1
CT*/
C -----

      program ex4f
      implicit none

C -----
C              Include files
C -----
C
C  The following include statements are required for Fortran programs
C  that use PETSc vectors:
C      petsc.h - base PETSc routines
C      vec.h   - vectors

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"

C -----
C              Macro definitions
C -----
C
C  Macros to make clearer the process of setting values in vectors and
C  getting values from vectors.
C
C  - The element xx_a(ib) is element ib+1 in the vector x
C  - Here we add 1 to the base array index to facilitate the use of
C    conventional Fortran 1-based array indexing.
C
#define xx_a(ib)  xx_v(xx_i + (ib))
#define yy_a(ib)  yy_v(yy_i + (ib))

C -----
C              Beginning of program
C -----

      Scalar  xwork(6)
      Scalar  xx_v(1), yy_v(1)
      integer i, n, ierr, loc(6), xx_i, yy_i
      Vec      x, y
      common xx_v, yy_v

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      n = 6

C  Create initial vector and duplicate it

      call VecCreateSeq(MPI_COMM_SELF,n,x,ierr)
      call VecDuplicate(x,y,ierr)

C  Fill work arrays with vector entries and locations. Note that
C  the vector indices are 0-based in PETSc (for both Fortran and
C  C vectors)

```

```

        do 10 i=1,n
            loc(i) = i-1
            xwork(i) = 10.0*i
10    continue

C Set vector values. Note that we set multiple entries at once.
C Of course, usually one would create a work array that is the
C natural size for a particular problem (not one that is as long
C as the full vector).

        call VecSetValues(x,6,loc,xwork,INSERT_VALUES,ierr)

C Assemble vector

        call VecAssemblyBegin(x,ierr)
        call VecAssemblyEnd(x,ierr)

C View vector

        write(6,20)
20    format('initial vector:')
        call VecView(x,VIEWER_STDOUT_SELF,ierr)
        call VecCopy(x,y,ierr)

C Get a pointer to vector data.
C - For default PETSc vectors, VecGetArray() returns a pointer to
C   the data array. Otherwise, the routine is implementation dependent.
C - You MUST call VecRestoreArray() when you no longer need access to
C   the array.
C - Note that the Fortran interface to VecGetArray() differs from the
C   C version. See the users manual for details.

        call VecGetArray(x,xx_v,xx_i,ierr)
        call VecGetArray(y,yy_v,yy_i,ierr)

C Modify vector data

        do 30 i=1,n
            xx_a(i) = 100.0*i
            yy_a(i) = 1000.0*i
30    continue

C Restore vectors

        call VecRestoreArray(x,xx_v,xx_i,ierr)
        call VecRestoreArray(y,yy_v,yy_i,ierr)

C View vectors

        write(6,40)
40    format('new vector 1:')
        call VecView(x,VIEWER_STDOUT_SELF,ierr)

        write(6,50)
50    format('new vector 2:')
        call VecView(y,VIEWER_STDOUT_SELF,ierr)

C Free work space. All PETSc objects should be destroyed when they
C are no longer needed.

```

```

call VecDestroy(x,ierr)
call VecDestroy(y,ierr)
call PetscFinalize(ierr)
end

```

Figure 17: Sample Fortran Program: Using VecSetValues() and VecGetArray()

```

C    "$Id: ex5.F,v 1.15 1996/09/12 16:27:09 bsmith Exp $";

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/draw.h"
C
C  This example demonstrates basic use of the Fortran interface for
C  Draw routines.
C
      Draw          draw
      DrawLG        lg
      DrawAxis      axis
      integer       n,i, ierr, x, y, width, height,flg
      Scalar        xd,yd

      n      = 20
      x      = 0
      y      = 0
      width  = 300
      height = 300

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)

      call OptionsGetInt(PETSC_NULL_CHARACTER,'-width',width,flg,ierr)
      call OptionsGetInt(PETSC_NULL_CHARACTER,'-height',height,flg,ierr)
      call OptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)

      call DrawOpenX(MPI_COMM_SELF,PETSC_NULL_CHARACTER,
&                  PETSC_NULL_CHARACTER,x,y,width,height,draw,ierr)

      call DrawLGCreate(draw,1,lg,ierr)
      call DrawLGGetAxis(lg,axis,ierr)
      call DrawAxisSetColors(axis,DRAW_BLACK,DRAW_RED,DRAW_BLUE,ierr)
      call DrawAxisSetLabels(axis,'toplabel','xlabel','ylabel',ierr)

      do 10, i=0,n-1
         xd = i - 5.0
         yd = xd*xd
         call DrawLGAddPoint(lg,xd,yd,ierr)
10    continue

      call DrawLGIndicateDataPoints(lg,ierr)
      call DrawLGDraw(lg,ierr)
      call DrawFlush(draw,ierr)

      call PetscSleep(10)

      call DrawLGDestroy(lg,ierr)
      call DrawDestroy(draw,ierr)
      call PetscFinalize(ierr)
      stop

```

end

Figure 18: Sample Fortran Program: Using PETSc Draw Routines

```
C "$Id: ex1f.F,v 1.10 1997/01/01 03:41:24 bsmith Exp $";
C
C/*T
C Concepts: SNES~Solving a system of nonlinear equations (basic uniprocessor example)
C Routines: SNESCreate(); SNESSetFunction(); SNESSetJacobian();
C Routines: SNESsolve(); SNESSetFromOptions(); SNESGetSLES();
C Routines: SLESGetPC(); SLESGetKSP(); KSPSetTolerances(); PCSetType();
C Processors: 1
CT*/
C
C Description: Uses the Newton method to solve a two-variable system.
C
C -----

      program main
      implicit none

C - - - - -
C              Include files
C - - - - -
C
C The following include statements are generally used in SNES Fortran
C programs:
C   petsc.h - base PETSc routines
C   vec.h   - vectors
C   mat.h   - matrices
C   ksp.h   - Krylov subspace methods
C   pc.h    - preconditioners
C   sles.h  - SLES interface
C   snes.h  - SNES interface
C Other include statements may be needed if using additional PETSc
C routines in a Fortran program, e.g.,
C   viewer.h - viewers
C   is.h     - index sets
C
#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/mat.h"
#include "include/FINCLUDE/kspace.h"
#include "include/FINCLUDE/pc.h"
#include "include/FINCLUDE/sles.h"
#include "include/FINCLUDE/snes.h"
C
C - - - - -
C              Variable declarations
C - - - - -
C
C Variables:
C   snes      - nonlinear solver
C   sles      - linear solver
C   pc        - preconditioner context
C   ksp       - Krylov subspace method context
C   x, r      - solution, residual vectors
```

```

C      J          - Jacobian matrix
C      its        - iterations for convergence
C
C      SNES      snes
C      SLES      sles
C      PC        pc
C      KSP       ksp
C      Vec       x, r
C      Mat       J
C      integer   ierr, its, size, rank
C      Scalar    pfive
C      Double    tol

C  Note: Any user-defined Fortran routines (such as FormJacobian)
C  MUST be declared as external.

      external FormFunction, FormJacobian

C  -----
C      Macro definitions
C  -----
C
C  Macros to make clearer the process of setting values in vectors and
C  getting values from vectors. These vectors are used in the routines
C  FormFunction() and FormJacobian().
C  - The element lx_a(ib) is element ib in the vector x
C
C  #define lx_a(ib) lx_v(lx_i + (ib))
C  #define lf_a(ib) lf_v(lf_i + (ib))
C
C  -----
C      Beginning of program
C  -----

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      call MPI_Comm_size(MPI_COMM_WORLD,size,ierr)
      if (size .ne. 1) then
        call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
        if (rank .eq. 0)
          &      write(6,*) 'This is a uniprocessor example only!'
          SETERRA(1,0,' ')
        endif

C  -----
C  Create nonlinear solver context
C  -----

      call SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,snex,ierr)

C  -----
C  Create matrix and vector data structures; set corresponding routines
C  -----

C  Create vectors for solution and nonlinear function

      call VecCreateSeq(MPI_COMM_SELF,2,x,ierr)
      call VecDuplicate(x,r,ierr)

C  Create Jacobian matrix data structure

```

```

        call MatCreate(MPI_COMM_SELF,2,2,J,ierr)

C Set function evaluation routine and vector

        call SNESSetFunction(snes,r,FormFunction,PETSC_NULL,ierr)

C Set Jacobian matrix data structure and Jacobian evaluation routine

        call SNESSetJacobian(snes,J,J,FormJacobian,PETSC_NULL,ierr)

C - - - - -
C Customize nonlinear solver; set runtime options
C - - - - -

C Set linear solver defaults for this problem. By extracting the
C SLES, KSP, and PC contexts from the SNES context, we can then
C directly call any SLES, KSP, and PC routines to set various options.

        call SNESGetSLES(snes,sles,ierr)
        call SLESGetKSP(sles,ksp,ierr)
        call SLESGetPC(sles,pc,ierr)
        call PCSetType(pc,PCNONE,ierr)
        tol = 1.e-4
        call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_DOUBLE_PRECISION,
&                                PETSC_DEFAULT_DOUBLE_PRECISION,20,ierr)

C Set SNES/SLES/KSP/PC runtime options, e.g.,
C   -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
C These options will override those specified above as long as
C SNESSetFromOptions() is called _after_ any other customization
C routines.

        call SNESSetFromOptions(snes,ierr)

C - - - - -
C Evaluate initial guess; then solve nonlinear system
C - - - - -

C Note: The user should initialize the vector, x, with the initial guess
C for the nonlinear solver prior to calling SNESsolve(). In particular,
C to employ an initial guess of zero, the user should explicitly set
C this vector to zero by calling VecSet().

        pfive = 0.5
        call VecSet(pfive,x,ierr)
        call SNESsolve(snes,x,its,ierr)
        if (rank .eq. 0) then
            write(6,100) its
        endif
100 format('Number of Newton iterations = ',i5)

C - - - - -
C Free work space. All PETSc objects should be destroyed when they
C are no longer needed.
C - - - - -

        call VecDestroy(x,ierr)
        call VecDestroy(r,ierr)
        call MatDestroy(J,ierr)
        call SNESDestroy(snes,ierr)

```



```

        call PetscFinalize(ierr)

        stop
        end
C -----
C
C FormFunction - Evaluates nonlinear function, F(x).
C
C Input Parameters:
C   snes - the SNES context
C   x - input vector
C   dummy - optional user-defined context (not used here)
C
C Output Parameter:
C   f - function vector
C
        subroutine FormFunction(snes,x,f,dummy)
            implicit none

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/snes.h"

            SNES      snes
            Vec        x, f
            integer    ierr, dummy(*)

C  Declarations for use with local arrays

            Scalar    lx_v(1), lf_v(1)
            common    lx_v, lf_v
            integer    lx_i, lf_i

C  Get pointers to vector data.
C  - For default PETSc vectors, VecGetArray() returns a pointer to
C    the data array. Otherwise, the routine is implementation dependent.
C  - You MUST call VecRestoreArray() when you no longer need access to
C    the array.
C  - Note that the Fortran interface to VecGetArray() differs from the
C    C version. See the Fortran chapter of the users manual for details.

            call VecGetArray(x,lx_v,lx_i,ierr)
            call VecGetArray(f,lf_v,lf_i,ierr)

C  Compute function

            lf_a(1) = lx_a(1)*lx_a(1)
            &          + lx_a(1)*lx_a(2) - 3.0
            lf_a(2) = lx_a(1)*lx_a(2)
            &          + lx_a(2)*lx_a(2) - 6.0

C  Restore vectors

            call VecRestoreArray(x,lx_v,lx_i,ierr)
            call VecRestoreArray(f,lf_v,lf_i,ierr)

            return
            end
C -----

```

```

C
C FormJacobian - Evaluates Jacobian matrix.
C
C Input Parameters:
C snes - the SNES context
C x - input vector
C dummy - optional user-defined context (not used here)
C
C Output Parameters:
C A - Jacobian matrix
C B - optionally different preconditioning matrix
C flag - flag indicating matrix structure
C
      subroutine FormJacobian(snes,X,jac,B,flag,dummy)
      implicit none

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/mat.h"
#include "include/FINCLUDE/pc.h"
#include "include/FINCLUDE/snes.h"

      SNES      snes
      Vec       X
      Mat       jac, B
      MatStructure flag
      Scalar    A(4)
      integer   ierr, idx(2), dummy(*)

C Declarations for use with local arrays

      Scalar lx_v(1)
      integer lx_i

C Get pointer to vector data

      call VecGetArray(x,lx_v,lx_i,ierr)

C Compute Jacobian entries and insert into matrix.
C - Since this is such a small problem, we set all entries for
C   the matrix at once.
C - Note that MatSetValues() uses 0-based row and column numbers
C   in Fortran as well as in C (as set here in the array idx).

      idx(1) = 0
      idx(2) = 1
      A(1) = 2.0*lx_a(1) + lx_a(2)
      A(2) = lx_a(1)
      A(3) = lx_a(2)
      A(4) = lx_a(1) + 2.0*lx_a(2)
      call MatSetValues(jac,2,idx,2,idx,A,INSERT_VALUES,ierr)
      flag = SAME_NONZERO_PATTERN

C Restore vector

      call VecRestoreArray(x,lx_v,lx_i,ierr)

C Assemble matrix

      call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)

```

```
call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)

return
end
```

Figure 19: Sample Fortran Program: Using PETSc Nonlinear Solvers

Part III

Additional Information

Chapter 10

Profiling

PETSc includes a consistent, lightweight scheme to allow the profiling of application programs. The PETSc routines automatically log performance data if certain options are specified at runtime. The user can also log information about application codes for a complete picture of performance. In addition, as described in Section 10.5, PETSc provides a mechanism for printing informative messages about computations. Section 10.1 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. See Section 15.4 for implementation details.

10.1 Basic Profiling Information

If an application code and the PETSc libraries have been compiled with the `-DPETSC_LOG` flag (which is the default for all versions), then various kinds of profiling of code between calls to `PetscInitialize()` and `PetscFinalize()` can be activated at runtime. Note that the flag `-DPETSC_LOG` can be specified for an installation of PETSc in the file `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base.$(BOPT)`, as discussed in Section 13.2. The profiling options include the following:

- **-log_summary** - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, **-log_summary** is intended as the primary means of monitoring the performance of PETSc codes.
- **-log [logfile]** - Generates a log file of basic data for examination with PETScView, a GUI utility described in Chapter 14 that provides a high-level view of the interrelationships among various code modules. Since the overhead for this monitoring is minor, it can be used for production runs.
- **-log_all [logfile]** - Generates a log file with extensive data for examination with PETScView. This option is thus intended to provide an overview of the computations within a program. Since the detailed event logging can significantly slow program execution, **-log_all** is not recommended for production runs.
- **-log_info** - Prints verbose information about code to stdout. This option provides details about algorithms, data structures, and so on. Since the overhead of printing such output slows a code, this option should not be used when evaluating a program's performance.
- **-log_trace [logfile]** - Traces the beginning and ending of all PETSc events. This option, which can be used in conjunction with **-log_info**, is useful to see where a program is hanging, without running in the debugger.

For the options **-log** and **-log_file**, the file `logfile` stores profiling information for later interpretation with PETScView; if `logfile` is not given, the file `Log.processor_rank` is used, where `processor_rank` is 0 for the first processor, etc. As discussed in Section 10.1.4, additional profiling can be done with of MPE.

10.1.1 Interpreting -log_summary Output: The Basics

As shown in Figure 7 (in Part I), the option `-log_summary` activates printing of profile data to standard output at the conclusion of a program. Profiling data can also be printed at any time within a program by calling `PLogPrintSummary()`.

We print performance data for each routine, organized by PETSc components, followed by any user-defined events (discussed in Section 10.2). For each routine, the output data include the maximum time and flop rate over all processors. Information about parallel performance is also included, as discussed in the following section.

For simplicity, the remainder of this discussion focuses on interpreting profile data for the **SLES** component, which provides the linear solvers at the heart of the PETSc package. Recall the hierarchical organization of the PETSc library, as shown in Figure 1. Each **SLES** solver is composed of a **PC** (preconditioner) and **KSP** (Krylov subspace) component, which are in turn built on top of the **Mat** (matrix) and **Vec** (vector) modules. Thus, operations in the **SLES** module are composed of lower-level operations in these components. Note also that the nonlinear solvers component, **SNES**, is built on top of the **SLES** module, and the timestepping component, **TS**, is in turn built on top of **SNES**.

We briefly discuss interpretation of the sample output in Figure 7, which was generated by solving a linear system on one processor using restarted GMRES and ILU preconditioning. The linear solvers in **SLES** consist of two basic phases, **SLESSetUp()** and **SLESSolve()**, 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 **SLESSetUp()** include all of the operations within **PCSetUp()**, which in turn include **MatILUFactor()**, and so on.

- **SLESSetUp** - Set up linear solver
 - **PCSetUp** - Set up preconditioner
 - **MatILUFactor** - Factor preconditioning matrix
 - **MatILUFactorSymbolic** - Symbolic factorization phase
 - **MatLUFactorNumeric** - Numeric factorization phase
- **SLESSolve** - Solve linear system
 - **PCApply** - Apply preconditioner
 - **MatSolve** - Forward/backward triangular solves
 - **KSPGMRESOrthog** - Orthogonalization in GMRES
 - **VecDot** or **VecMDot** - Inner products
 - **MatMult** - Matrix-vector product
 - **MatMultAdd** - Matrix-vector product + vector addition
 - **VecScale**, **VecNorm**, **VecAXPY**, **VecCopy**, ...

The summaries printed via `-log_summary` reflect this routine hierarchy. For example, the performance summaries for a particular high-level routine such as **SLESSolve** include all of the operations accumulated in the lower-level components that make up the routine. Using the GUI utility **PETScView** (described in Chapter 14) for a small example problem can help to provide the user with an understanding of the operations within an application code, thus making this hierarchy more apparent for a particular application.

Admittedly, we do not currently present the output with `-log_summary` so that the hierarchy of PETSc operations is completely clear, primarily because we have not determined a clean and uniform way to do so throughout the library. Improvements may follow. However, for a particular problem, the user should generally have an idea of the basic operations that are required for its implementation (e.g., which operations are performed when using GMRES and ILU, as described above), so that interpreting the `-log_summary` data should be relatively straightforward.

10.1.2 Interpreting -log_summary Output: Parallel Performance

We next discuss performance summaries for parallel programs, as shown within Figures 20 and 21, which present the combined output generated by the `-log_summary` option. The program that generated this data is `$(PETSC_DIR)/src/sles/examples/ex21.c`. 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 IBM SP, using restarted GMRES and the block Jacobi preconditioner, where each block was solved with ILU.

Figure 20 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 Section 10.3) are also given. Information about the various phases of computation then follow (as shown separately here in Figure 21). Finally, a summary of memory usage and object creation and destruction is presented.

We next focus on the summaries for the various phases of the computation, as given in the table within Figure 21. The summary for each phase presents the maximum times and flop rates over all processors, as well as the ratio of maximum to minimum times and flop rates for all processors. A ratio of approximately 1 indicates that computations within a given phase are well balanced among the processors; as the ratio increases, the balance becomes increasingly poor. Also, the total computational rate (in units of MFlops/sec) is given for each phase in the final column of the phase summary table.

$$\text{Total Mflop/sec} = 10^{-6} * (\text{sum of flops over all processors}) / (\text{max time over all processors})$$

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.

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), flops/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 Section 10.3). These statistics can aid in optimizing performance, since they indicate the sections of code that could benefit from various kinds of tuning. Chapter 11 gives suggestions about achieving good performance with PETSc codes.

10.1.3 Using -log and -log_all with PETScView

The PETSc utility `$(PETSC_DIR)/bin/petscview [logfile]` can be used to examine the profile data generated by `-log` and `-log_all`. Chapter 14 provides details regarding this Tk/Tcl tool, which provides a high-level view of the interrelationships among various code modules. Also, Section 10.4 gives information on restricting event logging.

10.1.4 Using -log_mpe with Upshot/Nupshot

It is also possible to use the *Upshot* (or *Nupshot*) package [9] to visualize PETSc events. This package comes with the MPE software, which is part of the MPICH [8] implementation of MPI. The option

```
-log_mpe [logfile]
```

creates a logfile of events appropriate for viewing with *Upshot*. The user can either use the default logging file, `mpe.log`, or specify an optional name via `logfile`.

To use this logging option, the user must employ the MPICH implementation of MPI and must compile the PETSc library with the `-DHAVE_MPE` flag, which is *not* activated by default. The user can turn on MPE logging by specifying `-DHAVE_MPE` in the `PCONF` variable within `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base.site` and (re)compiling all of PETSc.

By default, not all PETSc events are logged with MPE. For example, since `MatSetValues()` may be called thousands of times in a program, by default its calls are not logged with MPE. To activate MPE logging of a particular event, one should use the command

```
PLogEventMPEActivate(int event);
```

To deactivate logging of an event for MPE, one should use

```
PLogEventMPEDeactivate(int event);
```

The **event** may be either a predefined PETSc event (as listed in the file `$(PETSC_DIR)/include/petsclog.h`) or one obtained with `PetscEventRegister()` (as described in Section 10.2). These routines may be called as many times as desired in an application program, so that one could restrict MPE event logging only to certain code segments.

To see what events are logged by default, you can use the source; see files `src/plot/src/plogmpe.c` and `include/petsclog.h`. A simple program and GUI interface to see the events that are predefined and their definition is being developed.

You can also log the MPI events. To do this, just view the PETSc application as any MPI application, and follow the instructions on logging MPI calls that are appropriate for your MPI implementation. For the MPICH implementation, this simply involves adding `-llmpi` to the library list *ahead of* `-lmpi`.

10.2 Profiling Application Codes

PETSc automatically logs object creation, times, and floating-point counts for the library routines. Users can easily supplement this information by monitoring their application codes as well. The basic steps involved in logging a user-defined portion of code, called an *event*, are shown in the code fragment below:

```
#include "petsclog.h"
int USER_EVENT;
PLogEventRegister(&USER_EVENT,"User event name","Color:");
PLogEventBegin(USER_EVENT,0,0,0,0);
    /* application code segment to monitor */
    PLogFlops(number of flops for this code segment);
PLogEventEnd(USER_EVENT,0,0,0,0);
```

One must register the event by calling `PLogEventRegister()`, which assigns a unique integer to identify the event for profiling purposes:

```
ierr = PLogEventRegister(int *e,char *string,char *color);
```

Here **string** is a user-defined event name, and **color** is an optional user-defined event color (for use with *Upshot/Nupshot* logging); one should see the man page for details. The argument returned in **e** should then be passed to the `PLogEventBegin()` and `PLogEventEnd()` routines.

Events are logged by using the pair

```
PLogEventBegin(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);
PLogEventEnd(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);
```

The four objects are the PETSc objects that are most closely associated with the event. For instance, in a matrix-vector product they would be the matrix and the two vectors. These objects can be omitted by specifying 0 for **o1** - **o4**. The code between these two routine calls will be automatically timed and logged as part of the specified event.

The user can log the number of floating-point operations for this segment of code by calling

```
PLogFlops(number of flops for this code segment);
```

between the calls to `PLogEventBegin()` and `PLogEventEnd()`. This value will automatically be added to the global flop counter for the entire program.

10.3 Profiling Multiple Sections of Code

By default, the profiling produces a single set of statistics for all code between the `PetscInitialize()` and `PetscFinalize()` calls within a program. One can independently monitor up to ten stages of code by switching among the various stages with the commands

```
PLogStagePush(int stage);
PLogStagePop();
```

where **stage** is an integer (0-9); see the man pages for details. The command

```
PLogStageRegister(int stage,char *name)
```

allows one to associate a name with a stage; these names are printed whenever summaries are generated with `-log_summary` or `PLogPrintSummary()`. The following code fragment uses three profiling stages within a program.

```
PetscInitialize(int *argc, char ***args, 0, 0);
/* [stage 0 of code here] */
PLogStageRegister(0, "Stage 0 of Code");
for (i=0; i<ntimes; i++) {
    PLogStagePush(1);
    PLogStageRegister(1, "Stage 1 of Code");
    /* [stage 1 of code here] */
    PLogStagePop()
    PLogStagePush(2);
    PLogStageRegister(1, "Stage 2 of Code");
    /* [stage 2 of code here] */
    PLogStagePop()
}
PetscFinalize();
```

Figures 20 and 21 show output generated by `-log_summary` 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, Figure 21 contains output only for stages 4 and 5 (linear solve of the second system), which represent 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 Section 10.8.

10.4 Restricting Event Logging

By default, all PETSc operations are logged. To enable or disable the PETSc logging of individual events, one uses the commands

```
PLogEventActivate(int event);
PLogEventDeactivate(int event);
```

The `event` may be either a predefined PETSc event (as listed in the file `$(PETSC_DIR)/include/petsclog.h`) or one obtained with `PetscEventRegister()` (as described in Section 10.2).

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), **SLES** (linear solvers, including **KSP** and **PC** components), and **SNES** (nonlinear solvers):

```
PLogEventDeactivateClass(MAT_COOKIE);
PLogEventDeactivateClass(SLES_COOKIE); /* includes PC and KSP */
PLogEventDeactivateClass(VEC_COOKIE);
PLogEventDeactivateClass(SNES_COOKIE);
```

and

```
PLogEventActivateClass(MAT_COOKIE);
PLogEventActivateClass(SLES_COOKIE); /* includes PC and KSP */
PLogEventActivateClass(VEC_COOKIE);
PLogEventActivateClass(SNES_COOKIE);
```

Recall that the option `-log_all` produces extensive profile data, which can be a challenge for `PETScView` to handle because of the memory limitations of `Tcl/Tk`. Thus, one should generally use `-log_all` when running programs with a relatively small number of events or when disabling some of the events that occur many times in a code (e.g., `VecSetValues()`, `MatSetValues()`).

Section 10.1.4 gives information on the restriction of events in MPE logging.

10.5 Interpreting `-log_info` Output: Informative Messages

Users can activate the printing of verbose information about algorithms, data structures, and so on to stdout by using the option `-log_info` or by calling `PLogInfoAllow(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 Section 3.1.1, `-log_info` activates the printing of information about memory allocation during matrix assembly.

Application programmers can employ this logging as well, by using the routine

```
PLogInfo(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

```
PLogInfo(snes, "Cubically determined step, lambda %g\n", lambda);
```

One can selectively turn off informative messages about any of the basic PETSc objects (e.g., `Mat`, `SNES`) with the command

```
PLogInfoDeactivateClass(int object_cookie)
```

where `object_cookie` is one of `MAT_COOKIE`, `SNES_COOKIE`, and so on. Messages can be reactivated with the command

```
PLogInfoActivateClass(int object_cookie)
```

Such deactivation can be useful when one wishes to view information about higher level PETSc components (e.g., `TS` and `SNES`) without seeing all lower level data as well (e.g., `Mat`).

10.6 Time

PETSc application programmers can access the wall clock time directly with the command

```
double time = PetscGetTime();
```

In addition, as discussed in Section 10.2, PETSc can automatically profile user-defined segments of code.

10.7 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 `-log_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.

10.8 Accurate Profiling: Overcoming the Overhead of Paging

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 Section 10.1.4.)

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 Section 10.3, we can focus easily on the problem of interest. For example, we used this technique in the program `$(PETSC_DIR)/src/sles/examples/tutorials/ex10.c` to generate the timings within Figures 20 and 21. In this case, the profiled code of interest (solving the linear system for the larger problem) occurs within event stages 4 and 5. Section 10.1.2 provides details about interpreting such profiling data.

```
mpirun ex21 -f0 medium -f1 arco6 -ksp_gmres_unmodifiedgramschmidt -log_summary -mat_mplibaij \
-matload_block_size 3 -pc_type bjacobi -optionsleft
```

```
Number of iterations = 19
Residual norm = 7.7643e-05
Number of iterations = 55
Residual norm = 6.3633e-01
```

----- PETSc Performance Summary: -----

ex21 on a rs6000 named p039 with 4 processors, by mcinnes Wed Jul 24 16:30:22 1996

	Max	Min	Avg	Total
Time (sec):	3.289e+01	1.0	3.288e+01	
Objects:	1.130e+02	1.0	1.130e+02	
Flops:	2.195e+08	1.0	2.187e+08	8.749e+08
Flops/sec:	6.673e+06	1.0		2.660e+07
MPI Messages:	2.205e+02	1.4	1.928e+02	7.710e+02
MPI Message Lengths:	7.862e+06	2.5	5.098e+06	2.039e+07
MPI Reductions:	1.850e+02	1.0		

Summary of Stages:	Time	Flops	Messages	Message-lengths	Reductions	
	Avg	%Total	Avg	%Total	counts	%Total
0: Load System 0:	1.191e+00	3.6%	3.980e+06	0.5%	3.800e+01	4.9%
1: SLESSetup 0:	6.328e-01	2.5%	1.479e+04	0.0%	0.000e+00	0.0%
2: SLESSolve 0:	2.269e-01	0.9%	1.340e+06	0.0%	1.520e+02	19.7%
3: Load System 1:	2.680e+01	87.3%	0.000e+00	0.0%	2.100e+01	2.7%
4: SLESSetup 1:	1.867e-01	0.7%	1.088e+08	2.3%	0.000e+00	0.0%
5: SLESSolve 1:	3.831e+00	15.3%	2.217e+08	97.1%	5.600e+02	72.6%
					2.333e+06	11.4%
					1.120e+02	60.5%

.... [Summary of various phases, see part II below] ...

Memory usage is given in bytes:

Object Type	Creations	Destructions	Memory	Descendants' Mem.
Viewer	5	5	0	0
Index set	10	10	127076	0
Vector	76	76	9152040	0
Vector Scatter	2	2	106220	0
Matrix	8	8	9611488	5.59773e+06
Krylov Solver	4	4	33960	7.5966e+06
Preconditioner	4	4	16	9.49114e+06
SLES	4	4	0	1.71217e+07

Figure 20: Profiling a PETSc Program: Part I - Overall Summary

```

mpirun ex21 -f0 medium -f1 arco6 -ksp_gmres_unmodifiedgramschmidt -log_summary -mat_mplibaij \
-matload_block_size 3 -pc_type bjacobi -optionsleft

```

```

----- PETSc Performance Summary: -----
.... [Overall summary, see part I] ...

Phase summary info:
  Count: number of times phase was executed
  Time and Flops/sec: Max - maximum over all processors
                    Ratio - ratio of maximum to minimum over all processors
  Mess: number of messages sent
  Avg. len: average message length
  Reduct: number of global reductions
  Global: entire computation
  Stage: optional user-defined stages of a computation. Set stages with PLogStagePush() and PLogStagePop().
        %T - percent time in this phase      %F - percent flops 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 flops over all processors)/(max time over all processors)
-----
Phase              Count      Time (sec)      Flops/sec      --- Global ---      --- Stage ---      Total
                   Max      Ratio      Max      Ratio      Mess      Avg len      Reduct %T %F %M %L %R %T %F %M %L %R Mflop/s
-----
...

--- Event Stage 4: SLESSetUp 1

MatGetReordering      1  3.491e-03  1.0  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  2  0  0  0  0  0
MatILUFctrSymbol      1  6.970e-03  1.2  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  3  0  0  0  0  0
MatLUFctrNumer        1  1.829e-01  1.1  3.2e+07  1.1  0.0e+00  0.0e+00  0.0e+00  1  2  0  0  0  90  99  0  0  0  110
SLESSetUp             2  1.989e-01  1.1  2.9e+07  1.1  0.0e+00  0.0e+00  0.0e+00  1  2  0  0  0  99  99  0  0  0  102
PCSetUp              2  1.952e-01  1.1  2.9e+07  1.1  0.0e+00  0.0e+00  0.0e+00  1  2  0  0  0  97  99  0  0  0  104
PCSetUpOnBlocks       1  1.930e-01  1.1  3.0e+07  1.1  0.0e+00  0.0e+00  0.0e+00  1  2  0  0  0  96  99  0  0  0  105

--- Event Stage 5: SLESSolve 1

MatMult              56  1.199e+00  1.1  5.3e+07  1.0  1.1e+03  4.2e+03  0.0e+00  5  28  99  23  0  30  28  99  99  0  201
MatSolve             57  1.263e+00  1.0  4.7e+07  1.0  0.0e+00  0.0e+00  0.0e+00  5  27  0  0  0  33  28  0  0  0  187
VecNorm             57  1.528e-01  1.3  2.7e+07  1.3  0.0e+00  0.0e+00  2.3e+02  1  1  0  0  31  3  1  0  0  51  81
VecScale            57  3.347e-02  1.0  4.7e+07  1.0  0.0e+00  0.0e+00  0.0e+00  0  1  0  0  0  1  1  0  0  0  184
VecCopy              2  1.703e-03  1.1  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  0  0  0  0  0  0
VecSet               3  2.098e-03  1.0  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  0  0  0  0  0  0
VecAXPY              3  3.247e-03  1.1  5.4e+07  1.1  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  0  0  0  0  0  200
VecMDot             55  5.216e-01  1.2  9.8e+07  1.2  0.0e+00  0.0e+00  2.2e+02  2  20  0  0  30  12  20  0  0  49  327
VecMAXPY            57  6.997e-01  1.1  6.9e+07  1.1  0.0e+00  0.0e+00  0.0e+00  3  21  0  0  0  18  21  0  0  0  261
VecScatterBegin      56  4.534e-02  1.8  0.0e+00  0.0  1.1e+03  4.2e+03  0.0e+00  0  0  99  23  0  1  0  99  99  0  0
VecScatterEnd        56  2.095e-01  1.2  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  1  0  0  0  0  5  0  0  0  0  0
SLESSolve            1  3.832e+00  1.0  5.6e+07  1.0  1.1e+03  4.2e+03  4.5e+02  15  97  99  23  61  99  99  99  99  99  222
KSPGMRESOrthog       55  1.177e+00  1.1  7.9e+07  1.1  0.0e+00  0.0e+00  2.2e+02  4  39  0  0  30  29  40  0  0  49  290
PCSetUpOnBlocks       1  1.180e-05  1.1  0.0e+00  0.0  0.0e+00  0.0e+00  0.0e+00  0  0  0  0  0  0  0  0  0  0  0
PCApply             57  1.267e+00  1.0  4.7e+07  1.0  0.0e+00  0.0e+00  0.0e+00  5  27  0  0  0  33  28  0  0  0  186
-----
.... [Conclusion of overall summary, see part I] ...

```

Figure 21: Profiling a PETSc Program: Part II - Phase Summaries

Chapter 11

Hints for Performance Tuning

This chapter presents some tips on achieving good performance within PETSc 2.0 codes. We urge users to read these hints before evaluating the performance of PETSc application codes.

11.1 Compiler Options

Code compiled with the `BOPT=O` option generally runs two to three times faster than that compiled with `BOPT=g`, so we recommend using one of the optimized versions of code (`BOPT=O`, `BOPT=O_c++`, or `BOPT=O_complex`) when evaluating performance.

The user can specify alternative compiler options instead of the defaults set in the PETSc distribution. One can set the compiler options for a particular architecture (`PETSC_ARCH`) and `BOPT` by editing the file `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base.$(BOPT)`. Section 13.1.2 gives details.

11.2 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 Chapter 10, the PETSc routines automatically log performance data if certain runtime options are specified. We briefly highlight usage of these features below.

- Run the code with the option `-log_summary` to print a performance summary for various phases of the code.
- Run the code with the option `-log` (or `-log_all`), and use the utility `$(PETSC_DIR)/bin/petscview` to see where the code is fast and slow and what PETSc objects are being created.
- Compile the code with `BOPT=Opg` and then run `gprof` for a good-sized problem. This will provide an idea of where most of the run time is being spent. Generally, `gprof` cannot be used when running with more than one processor because all processors attempt to generate the same profiling log file.
- Run the code with the option `-log_mpe [logfile]`, which creates a logfile of events suitable for viewing with Upshot or Nupshot (part of MPICH).

11.3 Aggregation

Performing operations on chunks of data rather than a single element at a time can significantly enhance performance.

- 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 vector elements repeatedly, employ `VecGetArray()` to allow direct manipulation of the vector elements.

- When using `MatSetValues()`, if the column indices of the values being inserted have been sorted in monotonically increasing order, call the routine `MatSetOption(mat, MAT_COLUMNS_SORTED)` before setting the values to reduce the insertion time significantly.
- When possible, use `VecMDot()` rather than a series of calls to `VecDot()`.

11.4 Efficient Memory Allocation

11.4.1 Sparse Matrix Assembly

Since dynamic memory allocation for sparse matrices is inherently very expensive, accurate preallocation of memory is crucial for efficient sparse matrix assembly. One should use the matrix creation routines for particular data structures, such as `MatCreateSeqAIJ()` and `MatCreateMPIAIJ()` for compressed, sparse row formats, instead of the generic `MatCreate()` routine. For problems with multiple degrees of freedom per node, the block, compressed, sparse row formats, created by `MatCreateSeqBAIJ()` and `MatCreateMPIBAIJ()`, can significantly enhance performance. Also, Section 3.1.1 includes extensive details and examples regarding preallocation.

11.4.2 Sparse Matrix Factorization

When symbolically factoring an AIJ matrix, PETSc has to guess how much fill there will be. Careful use of the parameter `'f'` (fill estimate) 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 `f` is to run a program with the option `-log_info`. The symbolic factorization phase will then print information such as

```
Info:MatILUFactorSymbolic_AIJ:Realloc 12 Fill ratio:given 1 needed 2.16423
```

This indicates that the user should have used a fill estimate factor of about 2.17 (instead of 1) to prevent the 12 required mallocs and copies. The command line option

```
-mat_ilu_fill 2.17
```

will cause PETSc to preallocate the correct amount of space for incomplete (ILU) factorization. The corresponding option for direct (LU) factorization is `-mat_lu_fill <fill_amount>`.

11.4.3 PetscMalloc() Calls

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. Section 11.5 gives details.

11.5 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, and so on 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 `$(PETSC_DIR)/snes/examples/tutorials/ex5.c` and `$(PETSC_DIR)/snes/examples/tutorials/ex5f.F` for examples of user-defined application contexts in C and Fortran, respectively.

11.6 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`, `-sles_view`, and so on (or the routines `SLESView()`, `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 `-log_info` to print details about the solvers' operation.
- Use the PETSc monitoring discussed in Chapter 10 to evaluate the performance of various numerical methods.

11.7 Tips for Efficient Use of Linear Solvers

As discussed in Chapter 4, the default linear solvers are

- uniprocessor: GMRES(30) with ILU(0) preconditioning
- multiprocessor: GMRES(30) with block Jacobi preconditioning, where there is 1 block per processor, 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()`. Section 4.3 gives information on setting alternative GMRES orthogonalization routines, which may provide much better parallel performance.

11.8 Finding Memory Leaks

PETSc provides a number of tools to aid in detection of problems with memory allocation. We briefly describe these below.

- The PETSc memory allocation (which collects statistics and performs error checking), is employed by default for codes compiled in a debug mode (`BOPT=g`, `BOPT=g_c++`, `BOPT=g_complex`). PETSc memory allocation can be activated for other other cases, such as `BOPT=0`, with the option `-trmalloc`, while `-notrmalloc` forces the use of conventional memory allocation for the `BOPT=g`, `BOPT=g_c++`, and `BOPT=g_complex` versions. When running timing tests, one should always use `-notrmalloc` or the `BOPT=0` version of the libraries.
- When the PETSc memory allocation routines are used, the option `-trdump` will print a list of unfreed memory at the conclusion of a program. If all memory has been freed, only a message stating the maximum allocated space will be printed. However, if some memory remains unfreed, this information will be printed. Note that the option `-trdump` merely activates a call to `TrDump()` during `PetscFinalize()`; the user can also call `TrDump()` elsewhere in a program.

- The utility `$(PETSC_DIR)/bin/petscview` can illustrate graphically what PETSc objects have never been properly destroyed during a run. First, one should check that the source has been compiled with the `-DPETSC_LOG` flag (which is the default for all versions). Then one runs the program with the `-log` option, and views the resulting log file with `petscview`, which uses colors to illustrate different levels of object activity. When PETSc objects have been destroyed, they change color (gray, by default). Any objects that have not been properly destroyed (so that their memory remains allocated) should not turn gray.

11.9 Machine-specific Optimizations

- On the IBM SP, using the `mpirun` option `-nopoll` may improve the performance of some PETSc programs.

11.10 System-related Problems

The performance of a code can be affected by a variety of factors, including the cache behavior and other users on the machine. 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 10 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 `ps` command. Also, the PETSc option `-log_summary` prints the amount of memory used by the basic PETSc objects, thus providing a lower bound on the memory used.
- **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` 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 1.e-6 seconds or less.
- **Problem too large for good cache performance:** Certain machines with low memory bandwidths (slow memory access) attempt to compensate by having a very large cache; Sun UltraSparcs and DEC Alphas are examples of such machines. 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, the problem may likely be too large for the cache size. There is really no solution to this performance problem, except not purchasing such equipment. For example, IBM RS/6000 machines do not display this problem.
- **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. Section 10.8 provides information on overcoming paging overhead when profiling a code.

Chapter 12

Other PETSc Features

12.1 Options

Allowing the user to modify parameters and options easily at runtime is very desirable for many applications. PETSc 2.0 provides a simple mechanism to enable such runtime customization. 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:

```
ierr = PetscInitialize(int *argc,char ***args,char *file_name,char *help_message);  
call PetscInitialize(character file_name,integer ierr)
```

The arguments `argc` and `args` (in the C/C++ version only) are the usual command line arguments, while the `file_name` is a name of a file that can contain additional options. By default this file is called `.petscrc` in the user's home directory. The user can also specify options via the environmental variable `PETSC_OPTIONS`. The options are processed in the following order:

- file
- environmental variable
- 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. The option values can have no intervening spaces in them either. The user can employ any naming convention. For uniformity throughout PETSc, we employ the format `-package_option` (for instance, `-ksp_type` and `-mat_view_info`).

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 one of the following symbols in the first column of a line: `#`, `%`, or `!`.

Any subroutine in a PETSc program can add entries to the database with the command

```
ierr = OptionsSetValue(char *name,char *value);
```

though this is rarely done. To locate options in the database, one should use the commands

```

ierr = OptionsHasName(char *pre,char *name,int *flg);
ierr = OptionsGetInt(char *pre,char *name,int *value,int *flg);
ierr = OptionsGetDouble(char *pre,char *name,double *value,int *flg);
ierr = OptionsGetString(char *pre,char *name,char *value,int maxlen,int *flg);
ierr = OptionsGetIntArray(char *pre,char *name,int *value,int *nmax,int *flg);
ierr = OptionsGetDoubleArray(char *pre,char *name,double *value, int *nmax,int *flg);

```

All of these routines set `flg=1` if the corresponding option was found, `flg=0` 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 `PETSC_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`.

One useful means of keeping track of user-specified runtime options is use of `-optionstable`, which prints to `stdout` during `PetscFinalize()` a table of all runtime options that the user has specified. A related option is `-optionsleft`, 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.

Since PETSc has a large number of options that can be easily forgotten, we have included a Tk/Tcl program, `$PETSC_DIR/bin/petscopts`, that provides a GUI interface to set the options. This program sets the options in the user’s `.petsrc` file. Chapter 14.3 gives details.

12.2 Viewers: Looking at PETSc Objects

PETSc employs a consistent scheme for examining, printing, and saving objects through commands of the form

```
ierr = XXXView(XXX obj,Viewer viewer);
```

Here `obj` is any PETSc object of type `XXX`, where `XXX` is `Mat`, `Vec`, `SNES`, and so forth. There are several predefined viewers:

- Passing in a zero for the viewer causes the object to be printed to `stdout`; this is most useful when viewing an object in a debugger.
- `VIEWER_STDOUT_SELF` causes the object to be printed to `stdout`. For parallel objects the parts owned by individual processors are printed in random order.
- `VIEWER_STDOUT_WORLD` is similar to `VIEWER_STDOUT_SELF`, except that parallel objects are printed in the natural order that would occur if only a single processor were in use for the entire problem; this viewer uses the communicator `MPI_COMM_WORLD`.
- `VIEWER_DRAWX_WORLD` causes the object to be drawn in a default X window.
- `VIEWER_DRAWX_SELF` causes the object to be drawn in a default X window.
- Passing in a viewer obtained by `ViewerDrawOpenX()` causes the object to be displayed graphically.
- To save an object to a file in ASCII format, the user creates the viewer object with the command `ViewerFileOpenASCII(MPIComm comm, char* file, Viewer *viewer)`. This object is analogous to `VIEWER_STDOUT_SELF` (for a communicator of `MPI_COMM_SELF`) and `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 `ViewerFileOpenBinary(MPIComm comm,char* file,ViewerBinaryType type, Viewer *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 `ViewerMatlabOpen(MPIComm comm, char *machine, int port, Viewer *viewer)`. On the Matlab side, one must first run `v = openport(int port)` and then `A = receive(v)` to obtain the matrix or vector. Once all objects have been received, the port can be closed from the Matlab end with `closeport(v)`. On the PETSc side, one should destroy the viewer object with `ViewerDestroy()`. The corresponding Matlab mex files are located in `$(PETSC_DIR)/src/viewer/impls/matlab`.

The user can control the format of ASCII printed objects with viewers created by `ViewerFileOpenASCII()` by calling

```
ierr = ViewerSetFormat(Viewer viewer, int format, char *name);
```

Possible formats include `VIEWER_FORMAT_ASCII_DEFAULT`, `VIEWER_FORMAT_ASCII_MATLAB`, and `VIEWER_FORMAT_ASCII_IMPL`. The implementation-specific format, `VIEWER_FORMAT_ASCII_IMPL`, displays the object in the most natural way for a particular implementation. For example, when viewing a block diagonal matrix that has been created with `MatCreateSeqBDiag()`, `VIEWER_FORMAT_ASCII_IMPL` prints by diagonals, while `VIEWER_FORMAT_ASCII_DEFAULT` uses the conventional row-oriented format.

The routines

```
ierr = ViewerPushFormat(Viewer viewer, int format, char *name);
ierr = ViewerPopFormat(Viewer 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 `ViewerFileOpenBinary()` and then using `MatView()`, `VecView()`, and the like. 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:

```
ierr = MatLoad(Viewer viewer, MatType outtype, Mat *newmat);
ierr = VecLoad(Viewer viewer, Vec *newvec);
```

These routines generate parallel matrices and vectors if the viewer's communicator has more than one processor. The particular matrix and vector formats are determined from the options database; see the man 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 `.petsrc` file and can (currently) contain the following:

```
-matload_block_size <bs>
-matload_bdiag_diags <s1,s2,s3,...>
```

The block size indicates the size of blocks to use if the matrix is read into a block oriented data structure (for example, `MATSEQBDIAG` or `MATMPIBAIJ`). The diagonal information `s1,s2,s3,...` indicates which (block) diagonals in the matrix have nonzero values. Section 15.5.9 gives details.

12.3 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

```
ierr = PetscPushErrorHandler(int (*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

```
ierr = PetscPopErrorHandler()
```

removes the last error handler and discards it.

PETSc provides two additional error handlers besides `PetscTraceBackErrorHandler()`:

```
PetscAbortErrorHandler()
PetscAttachErrorHandler()
```

`PetscAbortErrorHandler()` calls abort on encountering an error, while `PetscAttachErrorHandler()` attaches a debugger to the running process if an error is detected. At runtime, these error handlers can be set with the options `-on_error_abort` or `-on_error_attach_debugger` [`noxterm`, `dbx`, `xxgdb`, `xldb`] [`-display DISPLAY`].

All PETSc calls can be traced (useful for determining where a program is hanging without running in the debugger) with the option

```
-log_trace [filename]
```

where `filename` is optional. By default the traces are printed to the screen. This can also be set with the command `PLogTraceBegin(FILE*)`.

It is also possible to trap signals by using the command

```
ierr = PetscPushSignalHandler( int (*Handler)(int,void *),void *ctx);
```

The default handler `PetscDefaultSignalHandler()` calls `PetscError()` and then terminates. In general, a signal in PETSc indicates a catastrophic failure. Any error handler that the user provides should try to clean up only before exiting. By default all PETSc programs use the default signal handler, although the user can turn this off at runtime with the option `-no_signal_handler`.

There is a separate signal handler for floating-point exceptions. The option `-fp_trap` turns on the floating-point trap at runtime, and the routine

```
ierr = PetscSetFPTrap(int flag);
```

can be used in-line. A `flag` of `PETSC_FP_TRAP_ON` indicates that floating-point exceptions should be trapped, while a value of `PETSC_FP_TRAP_OFF` (the default) indicates that they should be ignored. Note that on certain machines, in particular the IBM RS/6000, trapping is very expensive.

A small set of macros is used to make the error handling lightweight. These macros are used throughout the PETSc libraries and can be employed by the application programmer as well. When an error is first detected, one should set it by calling

```
SETERRQ(int flag,int pflag,char *message);
```

The user should check the return codes for all PETSc routines (and possibly user-defined routines as well) with

```
ierr = PetscRoutine(...); CHKERRQ(ierr);
```

Likewise, all memory allocations should be checked with

```
ptr = (double *) PetscMalloc(n*sizeof(double)); CHKPTRQ(void *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. In any main programs, however, the variants `SETERRA()`, `CHKERRA()`, and `CHKPTRA()` should be used instead to cause all processes of program to abort when an error has been detected. Use of the abort variant of the error checking commands is critical in the main program, since they ensure that `MPI_Abort()` is called before the process ends; otherwise, other MPI processes that did not generate errors may remain unterminated.

Note that the macro `__FUNC__` 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()`, `SETERRA()`, `CHKERRQ()`, or `CHKERRA()`. A simple example of usage is given below.

```
#undef __FUNC__
#define __FUNC__ "MyRoutine1"
int MyRoutine1() {
    /* code here */
    return 0;
}
```

12.4 Incremental Debugging

When developing large codes, one is often in the position of having a correctly (or at least believed to be correctly) running code; making a change to the code then changes the results for some unknown reason. Often even determining the precise point at which the old and new codes diverge is a major pain. In other cases, a code generates different results when run on different numbers of processors, although in exact

arithmetic the same answer is expected. (Of course, this assumes that *exactly* the same solver and parameters are used in the two cases.)

PETSc provides some support for determining exactly where in the code the computations lead to different results. First, one should compile both programs with different names. Next, one should start running both programs as a single MPI job. This procedure is dependent on the particular MPI implementation being used. For example, when using MPICH on workstations, *procgroup* files can be used to specify the processors on which the job is to be run. Thus, to run two programs, **old** and **new**, each on two processors, one should create the *procgroup* file with the following contents:

```
local 0
workstation1 1 /home/bsmith/old
workstation2 1 /home/bsmith/new
workstation3 1 /home/bsmith/new
```

(Of course, workstation1, etc. can be the same machine.) Then, one can execute the command

```
mpirun -p4pg <procgroup_filename> old -compare <tolerance> [your_program_options]
```

Note that the same runtime options must be used for the two programs. The first time an inner product or norm detects an inconsistency larger than *<tolerance>*, PETSc will generate an error. The usual runtime options *-start_in_debugger* and *-on_error_attach_debugger* may be used. The user can also place the commands

```
PetscCompareDouble()
PetscCompareScalar()
PetscCompareInt()
```

in portions of the application code to check for consistency between the two versions.

12.5 Complex Numbers

PETSc supports the use of complex numbers in application programs written in C, C++, and Fortran. To do so, we employ C++ versions of the PETSc libraries in which the basic “scalar” datatype, given in PETSc codes by **Scalar**, is defined as **complex** (or **complex<double>** for machines using templated complex class libraries). To work with complex numbers, the user should compile the PETSc libraries (including the Fortran interface library) and the application code with **BOPT=[g_complex,0_complex,0pg_complex]** for debugging, optimized, and profiling versions, respectively. The file `$(PETSC_DIR)/Installation` provides detailed instructions for installing PETSc.

Recall that each variant of the PETSc libraries is stored in a different directory, given by `$(PETSC_DIR)/lib/lib$(BOPT)/$(PETSC_ARCH)`, according to the architecture and **BOPT** optimization variable. 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 `$(PETSC_DIR)/src/sles/examples/tutorials/ex11.c` solves a linear system with a complex coefficient matrix. Its Fortran counterpart is `$(PETSC_DIR)/src/sles/examples/tutorials/ex11f.F`.

12.6 Emacs Users

If users develop application codes on UNIX machines using Emacs (which we highly recommend), the **etags** feature can be used to search PETSc files quickly and efficiently. To use this feature, one should first check that the file `$(PETSC_DIR)/TAGS` 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 Emacs the user should issue the command “**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.

12.7 VI Users

If users develop application codes on UNIX machines using VI, the **ctags** feature can be used to browse PETSc files quickly and efficiently. To use this feature, one should first check that the file, `$(PETSC_DIR)/tags` exists. If this file is not present, it should be generated by running **make tags** from the PETSc home directory. Once the file exists, from VI, the user should issue the command “**:set tags=\$(PETSC_DIR)/tags**” or the user should add to his “`/.exrc`” file the line “**set tags=\$(PETSC_DIR)/tags**”. Then the command “**:tag FunctionName**” will cause VI to find the file and line number where a desired PETSc function is defined.

12.8 Parallel Communication

When used in a message-passing environment, all communication within PETSc is done through MPI, the Message Passing Interface standard [14]. Any file that includes **petsc.h** (or any other PETSc include file), can freely use any MPI routine.

Chapter 13

Makefiles

This chapter describes the design of the PETSc makefiles, which are the key to managing code across a wide variety of Unix systems.

13.1 Our Makefile System

To make a program named **ex1**, one may use the command

```
make BOPT=[g,0,Opt] PETSC_ARCH=arch ex1
```

which will compile a debugging, optimized, or profiling version of the example and automatically link the appropriate libraries. The architecture, **arch**, is one of **sun4**, **solaris**, **rs6000**, **IRIX**, **hpux**, **freebsd**, and so on. Note that when using command line options with make (as illustrated above), one must *not* place spaces on either side of the “=” signs. The variables **BOPT** and **PETSC_ARCH** can also be set as environmental variables. Although PETSc is written in C, it can be compiled with a C++ compiler. For many C++ users this may be the preferred route. To compile with the C++ compiler, one should use the option **BOPT=g_c++** or **BOPT=O_c++**, or **BOPT=Opt_c++**. The options **BOPT=g_complex**, **BOPT=O_complex**, and **BOPT=Opt_complex** will create versions that use complex double-precision numbers.

13.1.1 Makefile Commands

The directory `$(PETSC_DIR)/bmake` contains virtually all makefile commands and customizations to enable portability across different architectures. Most makefile commands for maintaining the PETSc system are defined in the file `$(PETSC_DIR)/bmake/common`. These commands, which process all appropriate files within the directory of execution, include

- **lib** - Updates the PETSc libraries based on the source code in the directory.
- **libfast** - Updates the libraries faster. Since **libfast** recompiles all source files in the directory at once, rather than individually, this command saves time when many files must be compiled.
- **clean** - Removes garbage files.

Most other commands are intended for PETSc developers and are generally not needed by users.

- **ci** - Uses the RCS **ci** mechanism to check in all files in the directory.
- **co** - Uses the RCS **co** mechanism to check out all files in the directory.
- **fortranstubs** - Generates the Fortran wrapper routines.
- **latexpages** - Updates the LaTeX version of the man pages.
- **wwwpages** - Updates the HTML version of the man pages.

The **tree** command enables the user to execute a particular action within a directory and all of its subdirectories. The action is specified by **ACTION=[action]**, where **action** is one of the basic commands listed above. For example, if the command

```
make BOPT=g ACTION=lib tree
```

were executed from the directory `$(PETSC_DIR)/src/ksp`, the debugging library for all Krylov subspace solvers would be built.

13.1.2 Customized Makefiles

The directory `$(PETSC_DIR)/bmake` contains a subdirectory for each architecture that contains machine-specific information, enabling the portability of our makefile system. For instance, for Sun SPARCstations running OS 4.1.3, the directory is called **sun4**. Each architecture directory contains several base makefiles:

- **base.site** - locations of all needed include and library files for a particular site. This file (discussed below) is usually the only one that the user needs to alter.
- **base** - definitions of the compilers, linkers, etc.
- **base.g** - debugging options for C version.
- **base.O** - optimization options for C version.
- **base.Opg** - optimization options for C version, with gprof profiling.
- **base.g_complex** - debugging options for complex version.
- **base.O_complex** - optimization options for complex version.
- **base.Opg_complex** - optimization options for complex version, with gprof profiling.
- **base.g_c++** - debugging options for C++ version.
- **base.O_c++** - optimization options for C++ version.
- **base.Opg_c++** - optimization options for C++ version, with gprof profiling.

Each architecture base file, denoted by `$(PETSC_DIR)/bmake/$(PETSC_ARCH)/base`, includes the file `$(PETSC_DIR)/bmake/common`, which contains the rules discussed in Section 13.1.1 that are common to all machines.

We discovered that for no apparent reason, under freeBSD the include syntax is different from that of all other makefiles. Thus, under freeBSD *gnumake* must be used.

13.2 PETSc Flags

PETSc has several flags that determine how the source code will be compiled. The default flags for particular versions are specified by the variable **PETSCFLAGS** within the base files of `$(PETSC_DIR)/bmake/$(PETSC_ARCH)`, discussed in Section 13.1.2. The flags include

- **PETSC_DEBUG** - The PETSc debugging options are activated. We recommend always using this.
- **PETSC_COMPLEX** - The version with scalars represented as complex numbers is used.
- **PETSC_LOG** - Various monitoring statistics on floating-point operations and message-passing activity are kept.

Sample Makefiles

Maintaining portable PETSc makefiles is very simple. In Figures 22, 23, and 24 we present three sample makefiles.

The first is a minimum Makefile for maintaining a single program that uses the PETSc libraires.

```
ALL: ex2

CFLAGS      = $(CPPFLAGS)
FFLAGS      =
SOURCEC     =
SOURCEF     =
SOURCEH     =
OBJSC       =
OBJSF       =
LIBBASE     = libpetscsles
DIRS        =

include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base

ex2: ex2.o chkopts
    $(CLINKER) -o ex2 ex2.o $(PETSC_LIB)
    $(RM) ex2.o
```

Figure 22: Sample PETSc Makefile for Building a Single Program

The most important line in this makefile is the line starting **include**; this line includes other makefiles that provide the needed definitions and rules.

The second controls the generation of several example programs.

```
CFLAGS      = $(CONF) $(PETSC_INCLUDE)
SOURCEC     =
SOURCEF     =
SOURCEH     =
OBJSC       =
OBJSF       =
LIBBASE     = libpetscsles
RUNEXAMPLES_1 = runex1 runex2
RUNEXAMPLES_2 = runex4
RUNEXAMPLES_3 = runex3
EXAMPLESC   = ex1.c ex2.c ex4.c
EXAMPLESF   = ex3.F
EXAMPLES_1  = ex1 ex2
EXAMPLES_2  = ex4
EXAMPLES_3  = ex3

ex1: ex1.o
    -$(CLINKER) -o ex1 ex1.o $(PETSC_LIB)
    $(RM) ex1.o
ex2: ex2.o
    -$(CLINKER) -o ex2 ex2.o $(PETSC_LIB)
    $(RM) ex2.o
ex3: ex3.o
    -$(FLINKER) -o ex3 ex3.o $(PETSC_FORTRAN_LIB) $(PETSC_LIB)
    $(RM) ex3.o
ex4: ex4.o
    -$(CLINKER) -o ex4 ex4.o $(PETSC_LIB)
    $(RM) ex4.o
```

```

runex1:
    -@$(MPIRUN) ex1
runex2:
    -@$(MPIRUN) -np 2 ex2 -mat_seqdense -optionsleft
runex3:
    -@$(MPIRUN) ex3 -v -log_summary
runex4:
    -@$(MPIRUN) -np 4 ex4 -trdump

include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base

```

Figure 23: Sample PETSc Makefile for Example Programs

The two most important lines in the makefile of Figure 23 are the location of the **PETSc** home directory, set with **PETSC_DIR**, and the **include** line that includes the files defining all of the macro variables. The variable **PETSC_DIR** is automatically defined for the examples within the PETSc library structure. In general, the user should set **PETSC_DIR** to be the location of the local PETSc library. As listed in the sample makefile, the appropriate **include** file is automatically completely specified; the user should *not* alter this statement within the makefile. Some additional variables used in the makefile are defined as follows:

- **CLINKER**, **FLINKER** - the C and Fortran linkers.
- **RM** - the remove command for deleting files.
- **PETSC_INCLUDE** - the directory locations of any PETSc (or PETSc needed) files that are included in programs.
- **CONF** - various defines that indicate what packages are available.
- **PCONF** - defined in **\$(PETSC_ARCH)/base.site** to indicate which external packages are available at a particular site.
- **COPTFLAGS** - flags defined for each system and level of optimization (C/C++ compiler).
- **FOPTFLAGS** - flags defined for each system and level of optimization (Fortran compiler).
- **CPPFLAGS** - flags defined for each system and level of optimization for C/C++ preprocessor.
- **EXAMPLES_1** - examples that will be built with **make BOPT=[g,0,0pg] examples** (see Section 13.1.1)
- **RUNEXAMPLES_1** - examples that will be run with **make runexamples** (see Section 13.1.1)
- **EXAMPLESC** - all C examples that will be checked in/out of RCS with **make ci** and **make co** (not generally needed by users).
- **EXAMPLESF** - all Fortran examples that will be checked in/out of RCS with **make ci** and **make co** (not generally needed by users).
- **PETSC_LIB** - all of the base PETSc libraries.
- **PETSC_FORTRAN_LIB** - the PETSc Fortran interface library.

Note that the PETSc example programs are divided into several categories, which currently include:

```

EXAMPLES_1 - basic C suite used in installation tests
EXAMPLES_2 - additional C suite including graphics
EXAMPLES_3 - basic Fortran .F suite
EXAMPLES_4 - subset of 1 and 2 that runs on only a single processor
EXAMPLES_5 - examples that require complex numbers
EXAMPLES_6 - C examples that do not work with complex numbers
EXAMPLES_7 - C examples that require BlockSolve
EXAMPLES_8 - Fortran .F examples that do not work with complex numbers

```

EXAMPLES_9 - uniprocessor version of 3
EXAMPLES_10 - Fortran .F examples that require complex numbers

We next list in Figure 24 a makefile that maintains a PETSc library. Although most users do not need to understand or deal with such makefiles, they are also easily used.

```
ALL: lib

CFLAGS    = $(PETSC_INCLUDE) -I../.. -I$(PETSC_DIR)/pinclude $(CONF)
SOURCEC   = splwd.c spinver.c spnd.c spqmd.c sprcm.c
SOURCEF   = degree.f fnroot.f genqmd.f qmdqt.f rcm.f fnlwd.f genlwd.f \
            genrcm.f qmdrch.f rootls.f fndsep.f gennd.f qmdmrg.f qmdupd.f
SOURCEH   =
OBJSC     = splwd.o spinver.o spnd.o spqmd.o sprcm.o
OBJSF     = degree.o fnroot.o genqmd.o qmdqt.o rcm.o fnlwd.o genlwd.o \
            genrcm.o qmdrch.o rootls.o fndsep.o gennd.o qmdmrg.o qmdupd.o
LIBBASE   = libpetscmat
MANSEC    = 2

include $(PETSC_DIR)/bmake/$(PETSC_ARCH)/base
```

Figure 24: Sample PETSc Makefile for Library Maintenance

The library's name is **libpetscmat.a**, and the source files being added to it are indicated by **SOURCEC** (for C files) and **SOURCEF** (for Fortran files). Note that the **OBJSF** and **OBJSC** are identical to **SOURCEF** and **SOURCEC**, respectively, except they use the suffix **.o** rather than **.c** or **.f**.

The variable **MANSEC** indicates that any manual pages generated from this source should be included in the second section.

13.3 Limitations

This approach to portable makefiles has some minor limitations, including the following:

- Each makefile must be called “makefile”.
- Each makefile can maintain at most one archive library.

Chapter 14

PETSc GUI Utilities

PETSc includes two GUI utilities, PETScView and PETScOpts, that facilitate library use and interpretation of computational results. We acknowledge the contributions of Matt Hille (Washington State University), who focused on the design and documentation of these tools while a participant in Argonne's Summer Student Research Participation Program, 1995.

As discussed in Chapter 10, PETSc incorporates uniform event logging throughout the library in the form of statistics regarding object creation and destruction, floating-point operations, execution time, and memory usage. The utility PETScView provides an abstract interpretation of the profile data for a high-level view of the interrelationships among various code modules. PETScView assists in debugging, analysis, and performance enhancement and is especially useful for complex simulations that employ a combination of numerical methods and modeling techniques.

An additional GUI tool is PETScOpts, which provides a simple interface to the full range of PETSc options database commands. As discussed in Section 12.1, these options enable the user to set particular solvers, data structures, profiling options, and so on at runtime, thereby facilitating the customization and comparison of various algorithms and storage schemes.

14.1 Getting Started

PETScView and PETScOpts use the Tcl and the Tk Toolkit [1]. Therefore, in order to use the PETSc utility programs, the Tcl and Tk packages must be installed on the user's local system. See the following WWW site for information about Tcl/Tk: <http://www.sunlabs.com/research/tcl/>.

In order for PETScView and PETScOptions to work properly, a slight modification of the source code is required. Using any text editor, the user should load the file `$(PETSC_DIR)/bin/petscview` (or `$(PETSC_DIR)/bin/petscopts`) and change the first line in the source code to point to the proper location of where `wish` can be found. For example, if `wish` is located at `/usr/bin/wish`, the first line for each program should be changed to

```
#!/usr/local/wish -f
```

After this modification, PETScView and PETScOpts are ready to run.

Since Tcl/Tk are constantly changing (even faster than PETSc), it is difficult to keep `PetscView` and `PetscOpts` compatible with the latest release of Tcl/Tk, while still working with earlier releases. Thus, the user may have to modify the `PetscView` and `PetscOpts` script slightly to get them working with a particular version of Tcl/Tk.

14.2 Using PETScView

Whenever a PETSc program is executed with the `-log_all` or `-log` option, a log file is produced that can then be interpreted by PETScView. PETScView generates a dynamic tree-shaped hierarchy whose nodes contain icons that uniquely identify PETSc objects (such as linear solvers, matrices, and distributed arrays). The objects are color coded to denote the various states of activity, so that PETScView illustrates the

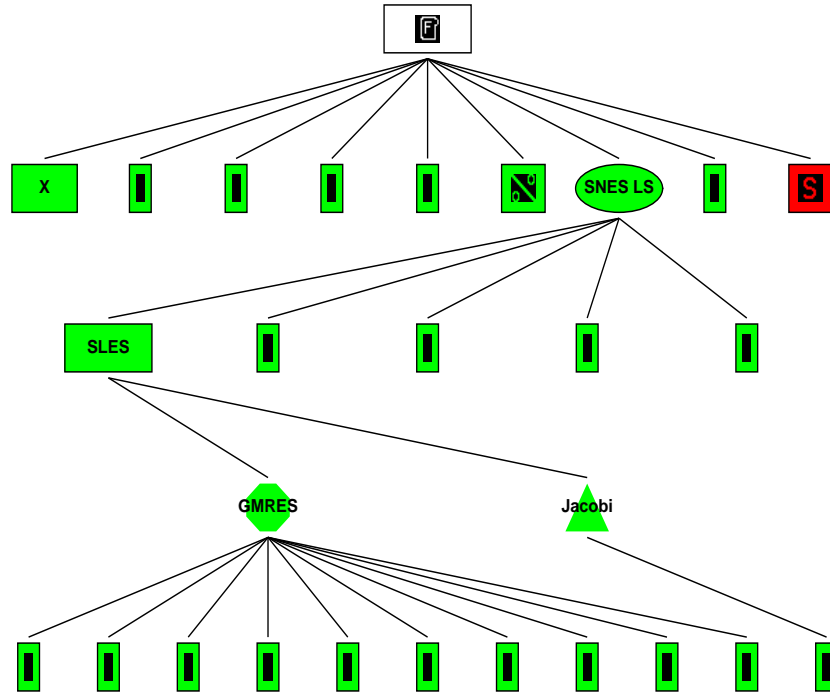


Figure 25: A Sample PETScView Object Tree

changing relationships among objects during program execution. A sample PETScView object tree is shown in Figure 25 for a parallel linear solver.

A number of built-in commands enable the application programmer to navigate easily through the simulation. In addition, PETScView can display performance statistics for particular objects and their children, thus enabling users to focus performance tuning efforts. This section provides introductory information regarding PETScView. Additional details are available via the “help” feature of the utility.

14.2.1 Running PETScView

Because of fundamental limitations of Tcl/Tk, PETScView can be run only with relatively small log files (at most a couple of thousand events). If the user generates a very large log file, Tcl/Tk will hang and often swamp the machine.

To begin PETScView, the user should type `petscview` from the UNIX shell prompt. To load a PETSc log file, one runs PETScView, giving the log file name as a command line argument:

```
petscview Log.0
```

This command invokes PETScView and automatically loads and interprets the profiling data contained in the file `Log.0`. PETScView can also be run without a log file given as a command line argument. In this case, the user must load the log file from within PETScView. To do this, the user selects the “Open File” command from the file menu. PETScView will automatically present the user with another window from which the file can be selected.

PETScView supports several additional command line arguments, as listed in Table 7. Note that the

Table 7: PETScView Command Line Options

Argument	Purpose
<code>-def_file filename</code>	The location of the definitions file
<code>-time</code>	Show the time
<code>-notime</code>	Do not show the time
<code>-stepsize N</code>	Set the stepsize to N
<code>-delay N</code>	Set the delay to N milliseconds
<code>-printer_dest DESTINATION</code>	Set the destination (File or Printer)
<code>-print_command COMMAND</code>	Set the postscript print command
<code>-printer PRINTER</code>	Specify the printer
<code>-print_orientation 1 or 0</code>	1 = landscape , 0 = portrait
<code>-print_color</code>	Specify color mode (color, gray, mono)

command line options override the default values within the user's `.petscviewrc` file, which is discussed further in the following section.

14.2.2 Loading a Configuration File

PETSc configuration files contain information that determines the graphical representation of PETSc objects within PETScView. Whenever PETScView is invoked, a configuration file is automatically read. The location of this file is specified in `.petscviewrc`, which is stored in the user's home directory. By default, `.petscviewrc` points to the configuration file, (`$PETSC_DIR/bin/petscview.cfg`); however, this can be changed to point to a configuration file created by the application programmer. Section 14.2.6 gives more information about changing the `.petscviewrc` file.

Even though PETScView loads a definitions file whenever it is initially run, a file of new definitions can be loaded from within PETScView at any time. This command is found in the file menu. Loading a new definitions file will automatically update all PETSc objects.

14.2.3 Printing a PETScView Object Tree

The “print” command of the file menu displays a dialog box from which the user can change the printing options. The default values are loaded from `.petscviewrc` when PETScView is first run. When the proper options are set for printing, one clicks on the “print” button or presses “return”. If the user is printing to a file, another dialog box will appear from which the user may specify the output filename.

Note: Currently, PETScView prints trees of size less than 1024 x 768 (measured in pixels). If the scrollbars are needed to view any parts of a tree, it is very unlikely that the whole tree will be printed. This situation presents no problem, however, when the tree is printed to a file.

14.2.4 Exiting PETScView

To exit PETScView, one selects the “Exit” option of the file menu. When this action is confirmed, PETScView will terminate and return the user to the calling shell.

14.2.5 The PETScView Simulation

Navigation

Once a file has been loaded by PETScView, the user can navigate through the simulation by using PETScView's play bar. The play bar is located at the bottom of the window and contains buttons whose appearances and functioning resemble the buttons on a tape player. From left to right, these buttons have the following functions:

- Rewind - Rewinds the simulation to the zeroth step.

- Step Backward - Steps backward through the simulation.
- Play Backward - Plays the simulation in reverse with an appropriate delay between steps.
- Stop - Brings the simulation to a halt.
- Play Forward - Plays forward through the simulation with an appropriate delay between steps.
- Step Forward - Steps forward through the simulation
- Finish - Immediately jumps to the last step of the simulation.

In addition to the play bar buttons, the scale directly below these buttons enables the user to navigate to an arbitrary position in the simulation. To use the scale, one clicks the left mouse button at the desired position on the scale. (The scale ranges from the zeroth step to the last step of the simulation.) The scale is scaled and labeled appropriately.

All of the above functions can also be accessed through the “player” menu. Selecting the player menu lists the commands in addition to their accelerator keys. The player menu also contains two additional commands that allow the user to jump to an arbitrary step in the simulation or to an arbitrary time during the simulation. When one of these commands is invoked, the user is prompted for the proper target jump value.

The View Menu

The View menu contains several additional commands that can be useful in the profiling of a PETSc program. These commands enable the user to view the raw profiling data as well as various statistics about program performance.

Changing Viewer Options

The options menu allows the user to change certain options that are set by default whenever PETScView is initially invoked. (The default values are defined in `.petscviewrc`.) These options include the step size when stepping through the simulation, the delay between events when playing through the simulation, and the colors chosen to denote the internal states of the PETSc objects.

The simulation’s step size can be changed at any time during the simulation by selecting the “step size” command of the Options menu. Whenever the user clicks on the this selection, a dialog box appears prompting the user for the desired step size. Valid step sizes range from 1 to the total number of events in the simulation.

To change the delay, one selects the “delay” command of the options menu. A cascading menu presents the user with a few built-in delays, which include none, real-time, and second delay. (The real-time delay option causes delays between events in the simulation to be proportionate to the delays in the actual execution of the PETSc program.) To specify a user-defined delay, the user clicks on the “after delay” selection. Then the user will be presented with a dialog box in which the user can specify the desired delay in milliseconds. Only values ranging from 1 to 2000 are valid.

The final selection on the Options menu presents the user with a cascading menu with entries that allow the user to change PETScView’s object color-coding scheme. Selecting any one of the entries presents the user with another window from which the color may be chosen. (The colors are taken from `/usr/lib/X11/rgb.txt`.)

14.2.6 Advanced Features

PETScView allows the user to define/redefine the graphical representation of PETSc objects. To do so, the user creates a configuration file, a Tcl script file, which includes specific definitions for group shapes, group labels, object icons, object labels, and action strings. Whenever PETScView is invoked, a configuration file is read from the location specified in the `.petscviewrc` file. By default, it is set to point to the configuration file included with the PETSc package (`$PETSC_DIR/bin/petscview.cfg`). Allowing the application programmer to create a customized configuration file enables PETScView to interpret the profiling data even when new PETSc objects have been created.

Table 8: PETSc Object Group Definitions

Group	Object_cookie
Viewers	0
Index Sets	1
Vectors	2
Vector Scattering	3
Matrices	4
Draw (simple graphics)	5
Line Graphs	6
Krylov Subspace Solvers	7
Preconditioners	8
Simplified Linear Equations Solvers	9
Grids	10
Stencils	11
Simplified Nonlinear Solvers	12
Distributed Arrays	13
Matrix Scattering	14

Group Definitions

Each PETSc object group is identified by its integer-valued object cookie. Table 8 lists the currently available object groups and their associated object cookies. For each group of objects, the PETScView requires the following definitions:

```
set GroupShape(OBJECT_COOKIE) SHAPE
set GroupDesc(OBJECT_COOKIE) "GROUP DESCRIPTION"
```

where **OBJECT_COOKIE** is the integer used to identify the group of objects and **SHAPE** is one of the predefined shapes used by PETScView. These shapes include

Square	Wide_Rectangle	Down_Triangle
Thin_RectangleV	Tall_Oval	Octagon
Thin_RectangleH	Wide_Oval	Circle
Rectangle	Up_Triangle	

GROUP DESCRIPTION is a line of text enclosed by quotes to describe the object group. For example, vectors have the following group descriptions:

```
set GroupShape(2) Thin_RectangleV
set GroupDesc(2) Vectors
```

Object Definitions

Within each group, the object is identified with the use of a second integer (**OBJECT_TYPE**). This specifies the type of object within the object group. PETScView requires both a name and an icon to be defined for every object. These definitions have the following syntax:

```
set Icon(OBJECT_COOKIE,OBJECT_TYPE) "-text TEXT" or
set Icon(OBJECT_COOKIE,OBJECT_TYPE) "-bitmap @BITMAP_LOC"
set Name(OBJECT_COOKIE,OBJECT_TYPE) "OBJECT DESCRIPTION"
```

where **TEXT** is a short description of the object (used if a bitmap is inappropriate) and **BITMAP_LOC** is the location of the bitmap graphic. The **OBJECT DESCRIPTION** is a line of text used to describe the object. As an example, we give the following definitions:

```
set Icon(2,0) "-bitmap @$env(PETSC_DIR)/bitmaps/vector.bit"
set Name(2,0) "Sequential Vector"
set Icon(2,1) "-bitmap @$env(PETSC_DIR)/bitmaps/vectorp.bit"
set Name(2,1) "Parallel Vector"
```

Notice the syntax that Tcl requires for the location of the bitmap. The bitmap location must be preceded by a @ in order for PETScView to work properly. `$env($PETSC_DIR)` is used to access the value of the environmental variable `PETSC_DIR`. To use the value of any other environmental variables in specifying a file location, one must use in the expression the following syntax:

```
$env(ENVIRONMENTAL_VARIABLE)
```

To create one's own bitmap picture to represent an object, the user creates the bitmap using a program such as `bitmap`. Once this is done, PETScView must know the location of the bitmap. The user must specify the precise location in the file system where the bitmap graphic can be found. For example, suppose that one creates a new bitmap to symbolize a parallel vector. Since the bitmap is located in the user's home directory, the following definition will *not* create an error:

```
set Icon(2,1) "-bitmap $env(HOME)/vectorp.bit"
```

More examples on defining additional PETSc objects and information about how PETSc defines the object types are given in `$PETSC_DIR/bin/petscview.cfg`.

Action Strings

When certain actions occur during the execution of a PETSc program, these actions are also recorded in the profiling data. Once again, PETSc uses an integer to specify the type of action that is being performed. PETScView interprets the actions using the definitions contained in `action()` string definitions. These definitions are also located in the configuration file. An action definition has the following syntax:

```
set Action(ACTION_ID) "ACTION"
```

where `ACTION_ID` is an integer that encodes the action and `ACTION` is a descriptive string. Currently, PETScView uses the action definitions as defined in `petscview.cfg`.

14.3 Using PETScOpts

PETScOpts is a PETSc utility program that enables the application programmer to modify his or her personal `.petscrc` file. As described in Section 12.1, the `.petscrc` file contains a list of options that will be passed to a PETSc program whenever it is executed. This file has the following format:

```
-optionname possible_value
-anotheroptionname possible_value
```

Even though this file can be manually modified by the application programmer with any text editor, PETScOpts greatly simplifies this task.

14.3.1 Running PETScOpts

The command `petscopts` will invoke PETScOpts from the UNIX shell prompt. Any entries contained in the `.petscrc` file of the user's home directory will automatically be interpreted by PETScOpts. Once inside PETScOpts, a number of entry boxes, check buttons, radio buttons, and other widgets allow the application programmer to specify the options that should be saved in the `.petscrc` file for future use.

PETScOpts can also write the PETSc command line options to a file other than the default `.petscrc` file. To do so, run PETScOpts with the file name as a command line argument:

```
petscopts file_name
```

From within PETScOpts, a different file can be loaded at any time by selecting the "Open file" option of the file menu.

14.3.2 Getting Help

Even though many of PETSc's command line options are self-explanatory, a single descriptive line of text is displayed at the bottom of the window whenever the pointer is positioned over any check button, radio button, or entry that specifies an option.

14.3.3 Exiting PETScOpts

The user can exit PETScOpts at any time by selecting the exit button from the file menu. If a `.petscrc` file was loaded when PETScOpts was initiated, the user is asked whether the current or original settings (or neither) should be saved in `.petscrc`.

Chapter 15

Design and Implementations of the Abstract Classes

PETSc 2.0 is designed using strong data encapsulation. Hence, any collection of data (for instance, a sparse matrix) is stored in a way that is completely private from the application code. The application code can manipulate the data only through a well-defined interface, as it does *not* know how the data is stored internally.

PETSc is designed around several components (e.g., **Vec** (vectors), **Mat** (matrices, both dense and sparse)). Each component has

- Its own include file `$(PETSC_DIR)/include/<component>.h`
- Its own directory, `$(PETSC_DIR)/src/<component>`
- An abstract data structure defined in the file `$(PETSC_DIR)/src/<component>/<component>impl.h`. This data structure is shared by all the different implementations of the component. For example, for matrices it is shared by dense, sparse, parallel, and sequential formats.
- An abstract interface that defines the application callable functions for the component. These are defined in the directory `$(PETSC_DIR)/src/<component>/interface`.
- One or more actual implementations of the components (for example, sparse uniprocessor and parallel matrices implemented with the AIJ storage format). These are each in a subdirectory of `$(PETSC_DIR)/src/<component>/impls`. Except in rare circumstances, data structures defined here should not be referenced from outside this directory.

Each type of object, for instance a vector, is defined in its own include file, by `typedef _Object* Object;` (for example, `typedef _Vec* Vec;`). This organization allows the compiler to perform type checking while at the same time completely removing the details of the implementation of `_Object` from the application code. The exact details of `_Object` may be changed at link time. This capability is extremely important because it allows the library internals to be changed without altering or recompiling the application code.

Polymorphism is supported through the directory `$(PETSC_DIR)/src/<component>/interface`, which contains the code that implements the abstract interface to the operations on the object. Essentially, these routines do some error checking of arguments and logging of profiling information and then call the function appropriate for the particular implementation of the object. The name of the abstract function is `ObjectOperation`, for instance, `MatMult` or `PCCreate`, while the name of a particular implementation is `ObjectOperation_Implementation`, for instance, `MatMult_SeqAIJ` or `PCCreate_ILU`. These naming conventions are used to simplify code maintenance.

Each object structure (named `_Object`) consists of three parts: a common PETSc header (defined in `include/phead.h`), a pointer to a list of operations for the object, and any additional information that is appropriate for the particular abstract object. The PETSc header includes an integer cookie, an integer type, an MPI communicator, a function pointer that indicates a destroy routine, and a function pointer that

indicates a viewer routine. The header can also contain additional records. Several routines are provided for manipulating data within the header, including

```
int PetscObjectGetComm(PetscObject object, MPI_Comm *comm)
```

which returns in `comm` the MPI communicator associated with the specified object.

After the header, each PETSc object contains a pointer to a list of operations for the object. For example, the vector operations include norms, assembly routines, scatters, and gathers. Finally, information that is appropriate for the particular abstract object is included. Generally, this information includes a pointer to data used by a particular implementation. For example, the `_Vec` structure is given by

```
struct _Vec {
    PETSCHEADER
    struct _VecOps ops;
    void          *data;
};
```

15.1 Names

Consistency of names for variables, functions, and the like is extremely important in making the package both usable and maintainable. We use several conventions:

- All function names and enum types consist of words, each of which is capitalized, for example, `SLESSolve()` and `MatGetReordering()`.
- All enum elements and macro variables are capitalized. When they consist of several complete words, there is an underscore between each word.
- Functions that are private to PETSc (not callable by the application code) either
 - have an appended `_Private` (for example, `StashValues_Private`) or
 - have an appended `_ObjectSubtype` (for example, `MatMult_SeqAIJ`).

In addition, functions that are not intended for use outside of a particular file are declared static.

- Function names in structures are the same as the base application function name without the object prefix, and all are in small letters. For example, `MatMultTrans()` has a structure name of `multtrans()`.
- Each application usable function begins with the name of the object, for example, `ISInvertPermutation` or `MatMult`.

15.2 Coding Conventions and Style Guide

Within the PETSc source code, we adhere to the following guidelines so that the code is uniform and easily maintainable:

- All PETSc function bodies are indented two characters.
- Each additional level of loops, if statements, and so on is indented two more characters.
- Wrapping lines should be avoided whenever possible.
- Source code lines should not be more than 120 characters wide.
- The macros `SETERRQ()` and `CHKERRQ()` should be on the same line as the routine to be checked unless this violates the 120-character-width rule. Try to make error messages short, but informative.
- The local variable declarations should be aligned. For example, use the style

```
int    i,j;
Scalar a;
```

instead of

```
int i,j;
Scalar a;
```

- All local variables of a particular type (e.g., `int`) should be listed on the same line if possible; otherwise, they should be listed on adjacent lines.
- Equal signs should be aligned in regions where possible.
- For routines of more than a few lines, there should be a blank line between the local variable declarations and the body of the function.
- Indentation for `if` statements should be done either as

```
if ( ) {
    ....
}
else {
    ....
}
```

or as

```
if ( ) {
    ....
} else {
    ....
}
```

- No tabs are allowed in *any* of the source code.

15.3 Option Names

Since consistency simplifies usage and code maintenance, the names of PETSc routines, flags, options, and so on have been selected with great care. The default option names are of the form `-package_subpackage_name`. For example, the option name for the basic convergence tolerance for the KSP package is `-ksp_atol`. In addition, operations in different packages of a similar nature have a similar name. For example, the option name for the basic convergence tolerance for the SNES package is `-snes_atol`.

15.4 Implementation of Profiling

This section provides details about the implementation of event logging and profiling within PETSc. Chapter 10 gives information about using the profiling in application codes.

The interface for profiling in PETSc is contained in the file `$(PETSC_DIR)/include/petsclog.h`. It includes

```
PLogObjectCreate(PetscObject h);
```

which logs the creation of any PETSc object. This should be included in any PETSc source code that uses `PetscHeaderCreate()`.

Just before an object is destroyed, it is logged with

```
PLogObjectDestroy(PetscObject h);
```

If an object has a clearly defined parent object (for instance, when a work vector is generated for use in a Krylov solver), this information is logged with the command,

```
PLogObjectParent(PetscObject parent,PetscObject child);
```

It is also useful to log information about the state of an object, as can be done with the command

```
#if defined(PETSC_LOG)
PLogObjectState(PetscObject h,char *format,...);
#endif
```

For example, for sparse matrices we usually log the matrix dimensions and number of nonzeros.

As discussed in the preceding section, events are logged using the pair

```
PLogEventBegin(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);
PLogEventEnd(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);
```

This logging is usually done in the abstract interface file for the operations, for example, `src/mat/src/matrix.c`.

Several routines that will be used rarely by the application programmer are

```
PLogBegin();
PLogAllBegin();
PLogDump(char *filename);
PLogPrintSummary(FILE *fd);
```

These routines are normally called by the `PetscInitialize()` and `PetscFinalize()` routines when the option `-log`, `-log_summary`, or `-log_all` is given.

15.5 The Various Matrix Classes

PETSc provides a variety of matrix implementations, since no single matrix format is appropriate for all problems. The following sections briefly describe the ever-expanding assortment of matrix types within PETSc.

15.5.1 Sequential AIJ Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the Yale sparse matrix format or compressed sparse row format, CSR). Section 3.1.1 describes this matrix type.

15.5.2 Parallel AIJ Sparse Matrices

15.5.3 Sequential Block AIJ Sparse Matrices

The sequential and parallel block AIJ formats, which are extensions of the AIJ formats described above, are intended especially for use with multicomponent PDEs. The block variants store matrix elements by fixed-sized dense $\text{nb} \times \text{nb}$ blocks, where currently `nb` ranges from two through five. These formats are fully compatible with standard Fortran 77 storage. That is, the stored row and column indices can begin at either one (as in Fortran) or zero.

The routine for creating a sequential block AIJ matrix with `m` rows, `n` columns, and a block size of `nb` is

```
ierr = MatCreateSeqBAIJ(MPI_Comm comm,int nb,int m,int n,int nz,int *nnz, Mat *A)
```

The arguments `nz` and `nnz` can be used to preallocate matrix memory by indicating the number of *block* nonzeros per row. For good performance during matrix assembly, preallocation is crucial; however, the user can set `nz=0` and `nnz=PETSC_NULL` for PETSc to dynamically allocate matrix memory as needed. Section 3.1.1 discusses preallocation for the AIJ format; extension to the block AIJ format is straightforward.

15.5.4 Parallel Block AIJ Sparse Matrices

Parallel block AIJ matrices with block size `nb` can be created with the command

```
ierr = MatCreateMPIBAIJ(MPI_Comm comm,int nb,int m,int n,int M,int N,int d_nz,
                        int *d_nnz, int o_nz,int *o_nnz,Mat *A);
```

`A` is the newly created matrix, while 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 processor, given by `m`, or determined by PETSc if `m` is `PETSC_DECIDE`.

If one does not use `PETSC_DECIDE` for `m` and `n`, one must ensure that they are chosen to be compatible with the vectors. To do this, one first considers the product $y = Ax$. The `m` that one uses in `MatCreateMPIBAIJ()` must match the local size used in the `VecCreateMPI()` for `y`. 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=PETSC_NULL`, and `o_nnz=PETSC_NULL` for PETSc to control dynamic allocation of matrix memory space. Analogous to `nz` and `nnz` for the routine `MatCreateSeqBAIJ()`, these arguments optionally specify block 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 processor's diagonal portion to be its local rows and the corresponding columns (a square submatrix); each processor's off-diagonal portion encompasses the remainder of the local matrix (a rectangular submatrix). Section 3.1.1 gives an example of preallocation for the parallel AIJ matrix format; extension to the block parallel AIJ case is straightforward.

15.5.5 Sequential Dense Matrices

PETSc provides both sequential and parallel dense matrix formats, where each processor stores its entries in a column-major array in the usual Fortran 77 style. Section 3.1.2 provides details on creating these matrices.

15.5.6 Parallel Dense Matrices

The parallel dense matrices are partitioned by rows across the processors, so that each local rectangular submatrix is stored in the dense format described above.

15.5.7 Parallel Cyclic Block Dense Matrices

Not yet implemented.

15.5.8 Parallel BlockSolve Sparse Matrices

PETSc provides a parallel, sparse, row-based matrix format that is intended for use in conjunction with the ILU and ICC preconditioners in `BlockSolve95`. Section 4.4.1 gives details on this matrix type.

15.5.9 Block Diagonal Sparse Matrices

Storage by block diagonals is available in both uniprocessor and parallel versions, although currently only a subset of matrix operations is supported. Each element of a block diagonal is defined to be a square dense block of size `nb × nb`, where conventional diagonal storage results for `nb=1`. Such storage is particularly useful for multicomponent PDEs discretized on regular grids.

The routine for creating a uniprocessor block diagonal matrix with `m` rows, `n` columns, and a block size of `nb` is

```
ierr = MatCreateSeqBDBdiag(MPI_COMM_SELF,int m,int n,int nd,int nb,int *diag,
                             Scalar **diagv,Mat *A);
```

The argument `nd` is the number of block diagonals, and `diag` is an array of block diagonal numbers. For the matrix element A_{ij} , where i and j respectively denote the row and column number of the element, the block diagonal number is computed using integer division by

$$\text{diag} = i/nb - j/nb.$$

If matrix storage space is allocated by the user, the argument `diagv` is a pointer to the actual diagonals (in the same order as the `diag` array). For PETSc to control memory allocation, the user should merely set `diagv=PETSC_NULL`.

A simple example of this storage format is illustrated below for block size `nb=1`. Here `nd = 4` and `diag = [2, 1, 0, -3]`. The diagonals need not be listed in any particular order, so that `diag = [-3, 0, 1, 2]` or `diag = [0, 2, -3, 1]` would also be valid values for the `diag` array.

a00	0	0	a03	0	0
a10	a11	0	0	a14	0
a20	a21	a22	0	0	a25
0	a31	a32	a33	0	0
0	0	a42	a43	a44	0
0	0	0	a53	a54	a55

15.5.10 Parallel Block Diagonal Sparse Matrices

The parallel block diagonal matrices are partitioned by rows across the processors, so that each local rectangular submatrix is stored by block diagonals as described above. The routine for creating a parallel block diagonal matrix with `m` local rows, `M` global rows, `n` global columns, and a block size of `nb` is

```
ierr = MatCreateMPIBDDiag(MPI_COMM_SELF,int m,int M,int N,int nd,int nb,int *diag,  
                          Scalar **diagv,Mat *A);
```

Either the `m` or `M` can be set to `PETSC_DECIDE` for PETSc to determine the corresponding quantity.

15.6 Other Libraries and Packages

It is not the intention of PETSc to provide all of the software pieces to solve a user's application problem. The intention is that other libraries can provide the pieces that are not in PETSc. Unfortunately, not many robust, general-purpose, easy-to-use software packages are available, and some of those that do exist have a very low-level user interface.

To simplify the use of certain packages, we have integrated into PETSc an interface to those packages. At the moment, these include parts of LAPACK, BLAS, BlockSolve95, and the reordering routines in SPARSPAK. We invite users to let us know (by writing to petsc-maint@mcs.anl.gov) whether they need particular libraries that are related to PETSc for an application; we may try to provide easy-to-use PETSc interfaces for them.

To allow the portable use of BLAS and LAPACK from C, we constructed a new include file, `$(PETSC_DIR)/include/pinclude/plapack.h`, that uses defines to make a uniform naming scheme for BLAS (all routines begin with BL) and LAPACK (all routines begin with LA). This file incorporates function prototypes for the BLAS and LAPACK routines used. In addition, the include file deals with different Fortran 77 compilers' treatment of function names (for instance, Sun compilers put an underscore at the end of all functions created in Fortran 77). The include file also deals with the fact that on Crays, C double precision is the same as Fortran 77 single precision. Lastly, it also calls the appropriate complex LAPACK and BLAS routines when needed.

Appendix A

PETSc Function Reference List

Routine Prefixes

Vec	Vector	Mat	Matrix routines
SLES	Linear equation solvers	KSP	Krylov subspace methods
PC	Preconditioners	IS	Index sets
SNES	Nonlinear equation solvers	DA	Distributed arrays
TS	timesteppers	Options	Options database
Petsc	Miscellaneous system	Draw	Drawing graphics routines
AO	Application	PLog	Profiling functions

A.1 Vector Routines

Data Structures:

- Vec - a vector of any type
- VecScatter - an object used in scatter routines, which allows reuse of communication information between different (but identical) scatters
- AO - application orderings, which provide mappings between user orderings and PETSc orderings

Norms:

- NORM_1
- NORM_2
- NORM_INFINITY or NORM_MAX

InsertMode:

- INSERT_VALUES
- ADD_VALUE

Runtime Options:

- -vec_mpi
- -vec_view
- -vec_view_draw
- -vec_view_draw_lg
- -vec_view_matlab

#include “vec.h”

int DrawTensorContour(Draw win,int m,int n,double *x,double *y,Vec V)
Draws a contour plot for a two-dimensional array that is stored as a PETSc vector.
int VecAXPBY(Scalar *alpha,Scalar *beta,Vec x,Vec y)
Computes $y = \alpha x + \beta y$.
int VecAXPY(Scalar *alpha,Vec x,Vec y)
Computes $y = \alpha x + y$.
int VecAYPX(Scalar *alpha,Vec x,Vec y)
Computes $y = x + \alpha y$.
int VecAbs(Vec v)
Replaces every element in a vector with its absolute value.
int VecAssemblyBegin(Vec vec)
Begins assembling the vector. This routine should be called after completing all calls to VecSetValues().
int VecAssemblyEnd(Vec vec)
Completes assembling the vector. This routine should be called after VecAssemblyBegin().
int VecCopy(Vec x,Vec y)
Copies a vector.
int VecCreateMPI(MPI_Comm comm,int n,int N,Vec *vv)
Creates a parallel vector.
int VecCreateSeq(MPI_Comm comm,int n,Vec *V)
Creates a standard, sequential array-style vector.
int VecCreate(MPI_Comm comm,int n,Vec *V)
Creates a vector, where the vector type is determined from the options database. Generates a parallel MPI vector if the communicator has more than one processor.
int VecDestroyVecs(Vec *vv,int m)
Frees a block of vectors obtained with VecDuplicateVecs().
int VecDestroy(Vec v)
Destroys a vector.
int VecDot(Vec x,Vec y,Scalar *val)
Computes the vector dot product.
int VecDuplicateVecs(Vec v,int m,Vec **V)
Creates several vectors of the same type as an existing vector.
int VecDuplicate(Vec v,Vec *newv)
Creates a new vector of the same type as an existing vector.
int VecEqual(Vec vec1,Vec vec2,PetscTruth *flag)
Compares two vectors.
int VecGetArrays(Vec *x,int n,Scalar ***a)
Returns a pointer to the arrays in a set of vectors that were created by a call to VecDuplicateVecs(). You MUST call VecRestoreArrays() when you no longer need access to the array.
int VecGetArray(Vec x,Scalar **a)
Returns a pointer to vector data. For default PETSc vectors, VecGetArray() returns a pointer to the local data array. Otherwise, this routine is implementation dependent. You MUST call VecRestoreArray() when you no longer need access to the array.
int VecGetLocalSize(Vec x,int *size)
Returns the number of elements of the vector stored in local memory. This routine may be implementation dependent, so use with care.
int VecGetOwnershipRange(Vec x,int *low,int *high)
Returns the range of indices owned by this processor, assuming that the vectors are laid out with the first n1 elements on the first processor, next n2 elements on the second, etc. For certain parallel layouts this range may not be well defined.
int VecGetSize(Vec x,int *size)
Returns the global number of elements of the vector.
int VecGetType(Vec vec,VecType *type,char **name)
Gets the vector type and name (as a string) from the vector.

int VecLoad(Viewer viewer,Vec *newvec)
Loads a vector that has been stored in binary format with VecView().
int VecMAXPY(int nv,Scalar *alpha,Vec x,Vec *y)
Computes $y[j] = \alpha[j] x + y[j]$.
int VecMDot(int nv,Vec x,Vec *y,Scalar *val)
Computes vector multiple dot products.
int VecMTDot(int nv,Vec x,Vec *y,Scalar *val)
Computes indefinite vector multiple dot products. That is, it does NOT use the complex conjugate.
int VecMax(Vec x,int *p,double *val)
Determines the maximum vector component and its location.
int VecMin(Vec x,int *p,double *val)
Determines the minimum vector component and its location.
int VecNorm(Vec x,NormType type,double *val)
Computes the vector norm.
int VecPlaceArray(Vec vec,Scalar *array)
Allows one to replace the array in a vector with a user provided one. This is useful to avoid copying an array into a vector. EXPERTS ONLY.
int VecPointwiseDivide(Vec x,Vec y,Vec w)
Computes the componentwise division $w = x/y$.
int VecPointwiseMult(Vec x,Vec y,Vec w)
Computes the componentwise multiplication $w = x*y$.
int VecReciprocal(Vec v)
Replaces each component of a vector by its reciprocal.
int VecRestoreArrays(Vec *x,int n,Scalar ***a)
Restores a group of vectors after VecGetArrays() has been called.
int VecRestoreArray(Vec x,Scalar **a)
Restores a vector after VecGetArray() has been called.
int VecScale(Scalar *alpha,Vec x)
Scales a vector.
int VecScatterBegin(Vec x,Vec y,InsertMode addv,ScatterMode mode,VecScatter inctx)
Begins a generalized scatter from one vector to another. Complete the scattering phase with VecScatterEnd().
int VecScatterCopy(VecScatter sctx,VecScatter *ctx)
Makes a copy of a scatter context.
int VecScatterCreate(Vec xin,IS ix,Vec yin,IS iy,VecScatter *newctx)
Creates a vector scatter context.
int VecScatterDestroy(VecScatter ctx)
Destroys a scatter context created by VecScatterCreate().
int VecScatterEnd(Vec x,Vec y,InsertMode addv,ScatterMode mode, VecScatter ctx)
Ends a generalized scatter from one vector to another. Call after first calling VecScatterBegin().
int VecScatterPostRecvsv(Vec x,Vec y,InsertMode addv,ScatterMode mode,VecScatter inctx)
Posts the receives required for the ready-receiver version of the VecScatter routines.
int VecScatterRemap(VecScatter scat,int *rto,int *rfrom)
Remaps the "from" and "to" indices in a vector scatter context. FOR EXPERTS ONLY!
int VecScatterView(VecScatter ctx, Viewer viewer)
Views a vector scatter context.
int VecSetLocalToGlobalMapping(Vec x, int n,int *indices)
Sets a local numbering to global numbering used by the routine VecSetValuesLocal() to allow users to insert vector entries using a local (per-processor) numbering.
int VecSetOption(Vec x,VecOption op)
Allows one to set options for a vectors behavior.
int VecSetRandom(PetscRandom rctx,Vec x)
Sets all components of a vector to random numbers.

int VecSetValuesLocal(Vec x,int ni,int *ix,Scalar *y,InsertMode iora) Inserts or adds values into certain locations of a vector, using a local ordering of the nodes.
int VecSetValues(Vec x,int ni,int *ix,Scalar *y,InsertMode iora) Inserts or adds values into certain locations of a vector.
void VecSetValue(Vec v,int row,Scalar value, InsertMode mode); Set a single entry into a vector.
int VecSet(Scalar *alpha,Vec x) Sets all components of a vector to a scalar.
int VecShift(Scalar *shift,Vec v) Shifts all of the components of a vector by computing $x[i] = x[i] + \text{shift}$.
int VecSum(Vec v,Scalar *sum) Computes the sum of all the components of a vector.
int VecSwap(Vec x,Vec y) Swaps the vectors x and y.
int VecTDot(Vec x,Vec y,Scalar *val) Computes an indefinite vector dot product. That is, this routine does NOT use the complex conjugate.
int VecValid(Vec v,PetscTruth *flag) Checks whether a vector object is valid.
int VecView(Vec v,Viewer viewer) Views a vector object.
int VecWAXPY(Scalar *alpha,Vec x,Vec y,Vec w) Computes $w = \alpha x + y$.

A.2 Matrix Routines

Data Structures:

- Mat - a matrix of any type (including matrix-free)

Matrix options:

- MAT_ROW_ORIENTED - inserts are done with row-oriented blocks
- MAT_COLUMN_ORIENTED - inserts are done with column-oriented blocks
- MAT_ROWS_SORTED - row indices in the inserted block are sorted
- MAT_COLUMNS_SORTED - column indices in the inserted block are sorted
- MAT_NO_NEW_NONZERO_LOCATIONS - additional inserts will not be allowed if they generate a new non-zero.
- MAT_YES_NEW_NONZERO_LOCATIONS - additional inserts will be allowed
- MAT_SYMMETRIC_MATRIX - matrix is symmetric
- MAT_STRUCTURALLY_SYMMETRIC_MATRIX - matrix is symmetric in nonzero structure

Orderings:

- ORDER_NATURAL - Natural
- ORDER_ND - Nested Dissection
- ORDER_1WD - One-Way Dissection
- ORDER_RCM - Reverse Cuthill-McKee
- ORDER_QMD - Quotient Minimum Degree

Norms:

- NORM_1
- NORM_2
- NORM_FROBENIUS
- NORM_INFINITY or NORM_MAX

Options to MatAssemblyXXX():

- MAT_FLUSH_ASSEMBLY - intermediate matrix assembly
- MAT_FINAL_ASSEMBLY - final matrix assembly

Options to MatGetInfo():

- MAT_LOCAL
- MAT_GLOBAL_MAX
- MAT_GLOBAL_SUM

MatType:

- MATSAME - same format as current (e.g., used for **MatConvert()**)
- MATSEQAIJ - sequential sparse row (AIJ) format
- MATMPIAIJ - parallel sparse row (AIJ) format
- MATSEQDENSE - sequential dense format
- MATMPIDENSE - parallel dense format
- MATMPIROWBS - parallel row-based format compatible with BlockSolve95
- MATSEQBAIJ - sequential sparse block row (BAIJ) format
- MATMPIBAIJ - parallel sparse block row (BAIJ) format
- MATSEQBDIAG - sequential block diagonal format
- MATMPIBDIAG - parallel block diagonal format

MatGetSubMatrixCall:

- MAT_INITIAL_MATRIX
- MAT_REUSE_MATRIX

Runtime Options:

- -mat_aij
- -mat_aij_dxml
- -mat_aij_essl
- -mat_aij_inode_limit [limit]
- -mat_aij_no_inode
- -mat_aij_oneindex
- -mat_aij_superlu

- -mat_baij
- -mat_bdiag
- -mat_bdiag_diags [diag_number_1,diag_number_2,...]
- -mat_bdiag_ndiag [number_of_diagonals]
- -mat_block_size [size]
- -mat_coloring
- -mat_dense
- -mat_ilu_fill [fill]
- -mat_lu_fill [fill]
- -mat_lu_pivotthreshold [threshold]
- -mat_mpiailj
- -mat_mpibaij
- -mat_mpibdiag
- -mat_mpidense
- -mat_mpirowbs
- -mat_no_unroll
- -mat_order [order]
- -mat_rowbs_no_inode
- -mat_seqailj
- -mat_seqbaij
- -mat_seqbdiag
- -mat_seqdense
- -mat_view
- -mat_view_draw
- -mat_view_info
- -mat_view_info_detailed
- -mat_view_matlab
- -matload_bdiag_diags
- -matload_block_size [size]
- -matload_ignore_info

#include "mat.h"

int MatAXPY(Scalar *a,Mat X,Mat Y) Computes $Y = a * X + Y$.
int MatAssemblyBegin(Mat mat,MatAssemblyType type) Begins assembling the matrix. This routine should be called after completing all calls to MatSetValues().
int MatAssemblyEnd(Mat mat,MatAssemblyType type) Completes assembling the matrix. This routine should be called after MatAssemblyBegin().
int MatBDiagGetData(Mat mat,int *nd,int *bs,int **diag,int **bdlen,Scalar ***diagv) Gets the data for the block diagonal matrix format. For the parallel case, this returns information for the local submatrix.
int MatCholeskyFactorNumeric(Mat mat,Mat *fact) Performs numeric Cholesky factorization of a symmetric matrix. Call this routine after first calling MatCholeskyFactorSymbolic().
int MatCholeskyFactorSymbolic(Mat mat,IS perm,double f,Mat *fact) Performs symbolic Cholesky factorization of a symmetric matrix.
int MatCholeskyFactor(Mat mat,IS perm,double f) Performs in-place Cholesky factorization of a symmetric matrix.
int MatColoringGetName(MatColoring meth,char **name) Gets the name associated with a coloring.
int MatColoringPatch(Mat mat,int n,int *colorarray,ISColoring *iscoloring) EXPERTS ONLY, used inside matrix coloring routines that use matGetRowIJ() and/or MatGetColumnIJ().
int MatColoringRegisterAll() Registers all of the matrix coloring routines in PETSc.
int MatColoringRegisterDestroy() Frees the list of coloring routines.
int MatColoringRegister(MatColoring *name,char *sname,int (*color)(Mat,MatColoring,ISColoring*)) Adds a new sparse matrix coloring to the matrix package.
int MatCompress(Mat mat) Tries to store the matrix in as little space as possible. May fail if memory is already fully used, since it tries to allocate new space.
int MatConvert(Mat mat,MatType newtype,Mat *M) Converts a matrix to another matrix, either of the same or different type.
int MatCopy(Mat A,Mat B) Copys a matrix to another matrix.
int MatCreateMPIAIJ(MPI_Comm comm,int m,int n,int M,int N, int d_nz,int *d_nnz,int o_nz,int *o_nnz,Mat *A) Creates a sparse parallel matrix in AIJ format (the default parallel PETSc format). For good matrix assembly performance the user should preallocate the matrix storage by setting the parameters d_nz (or d_nnz) and o_nz (or o_nnz). By setting these parameters accurately, performance can be increased by more than a factor of 50.
int MatCreateMPIBAIJ(MPI_Comm comm,int bs,int m,int n,int M,int N, int d_nz,int *d_nnz,int o_nz,int *o_nnz,Mat *A) Creates a sparse parallel matrix in block AIJ format (block compressed row). For good matrix assembly performance the user should preallocate the matrix storage by setting the parameters d_nz (or d_nnz) and o_nz (or o_nnz). By setting these parameters accurately, performance can be increased by more than a factor of 50.
int MatCreateMPIBDiag(MPI_Comm comm,int m,int M,int N,int nd,int bs, int *diag,Scalar **diagv,Mat *A) Creates a sparse parallel matrix in MPIBDiag format.
int MatCreateMPIDense(MPI_Comm comm,int m,int n,int M,int N,Scalar *data,Mat *A) Creates a sparse parallel matrix in dense format.
int MatCreateMPIRowbs(MPI_Comm comm,int m,int M,int nz,int *nnz,void *procinfo,Mat *newA) Creates a sparse parallel matrix in the MATMPIROWBS format. This format is intended primarily as an interface for BlockSolve95.

int MatCreateSeqAIJ(MPI_Comm comm,int m,int n,int nz,int *nnz, Mat *A)
Creates a sparse matrix in AIJ (compressed row) format (the default parallel PETSc format). For good matrix assembly performance the user should preallocate the matrix storage by setting the parameter nz (or the array nzz). By setting these parameters accurately, performance during matrix assembly can be increased by more than a factor of 50.
int MatCreateSeqBAIJ(MPI_Comm comm,int bs,int m,int n,int nz,int *nnz, Mat *A)
Creates a sparse matrix in block AIJ (block compressed row) format. For good matrix assembly performance the user should preallocate the matrix storage by setting the parameter nz (or the array nzz). By setting these parameters accurately, performance during matrix assembly can be increased by more than a factor of 50.
int MatCreateSeqBDiag(MPI_Comm comm,int m,int n,int nd,int bs,int *diag, Scalar **diagv,Mat *A)
Creates a sequential block diagonal matrix.
int MatCreateSeqDense(MPI_Comm comm,int m,int n,Scalar *data,Mat *A)
Creates a sequential dense matrix that is stored in column major order (the usual Fortran 77 manner). Many of the matrix operations use the BLAS and LAPACK routines.
int MatCreateShell(MPI_Comm comm,int m,int n,int M,int N,void *ctx,Mat *A)
Creates a new matrix class for use with a user-defined private data storage format.
int MatCreate(MPI_Comm comm,int m,int n,Mat *A)
Creates a matrix where the type is determined from the options database. Generates a parallel MPI matrix if the communicator has more than one processor. The default matrix type is AIJ, using the routines MatCreateSeqAIJ() and MatCreateMPIAIJ().
int MatDestroyMatrices(int n,Mat **mat)
Destroys a set of matrices obtained with MatGetSubMatrices().
int MatDestroy(Mat mat)
Frees space taken by a matrix.
int MatDiagonalScale(Mat mat,Vec l,Vec r)
Scales a matrix on the left and right by diagonal matrices that are stored as vectors. Either of the two scaling matrices can be PETSC_NULL.
int MatDiagonalShift(Mat Y,Vec D)
Computes $Y = Y + D$, where D is a diagonal matrix that is represented as a vector.
int MatEqual(Mat A,Mat B,PetscTruth *flg)
Compares two matrices.
int MatFDColoringApply(Mat J,MatFDColoring coloring,Vec x1,Vec w1,Vec w2,Vec w3, int (*f)(void *,Vec,Vec,void*),void *sctx,void *fctx)
Given a matrix for which a MatFDColoring has been created, computes the Jacobian for a function via finite differences.
int MatFDColoringCreate(Mat mat,ISColoring iscoloring,MatFDColoring *color)
Creates a matrix coloring context for finite difference computation of Jacobians.
int MatFDColoringDestroy(MatFDColoring c)
Destroys a matrix coloring context that was created via MatFDColoringCreate().
int MatFDColoringPrintHelp(MatFDColoring fd)
Prints help message for matrix finite difference calculations using coloring.
int MatFDColoringSetFromOptions(MatFDColoring matfd)
Set coloring finite difference parameters from the options database.
int MatFDColoringSetParameters(MatFDColoring matfd,double error,double umin)
Sets the parameters for the approximation of Jacobian using finite differences.
int MatFDColoringView(MatFDColoring color,Viewer viewer)
Views a finite difference coloring context.
int MatGetArray(Mat mat,Scalar **v)
Returns a pointer to the element values in the matrix. This routine is implementation dependent, and may not even work for certain matrix types. You MUST call MatRestoreArray() when you no longer need to access the array.
int MatGetBlockSize(Mat mat,int *bs)
Returns the matrix block size; useful especially for the block row and block diagonal formats.

int MatGetColoringTypeFromOptions(char *prefix,MatColoring *type) Gets matrix coloring method from the options database.
int MatGetColoring(Mat mat,MatColoring type,ISColoring *iscoloring) Gets a coloring for a matrix to reduce fill or to improve numerical stability of LU factorization.
int MatGetColumnIJ(Mat mat,int shift,PetscTruth symmetric,int *n,int **ia,int** ja,PetscTruth *done) Returns the compress Column storage i and j indices for sequential matrices. EXPERTS ONLY.
int MatGetDiagonal(Mat mat,Vec v) Gets the diagonal of a matrix.
int MatGetInfo(Mat mat,MatInfoType flag,MatInfo *info) Returns information about matrix storage (number of nonzeros, memory, etc.).
int MatGetLocalSize(Mat mat,int *m,int* n) Returns the number of rows and columns in a matrix stored locally. This information may be implementation dependent, so use with care.
int MatGetOwnershipRange(Mat mat,int *m,int* n) Returns the range of matrix rows owned by this processor, assuming that the matrix is laid out with the first n1 rows on the first processor, the next n2 rows on the second, etc. For certain parallel layouts this range may not be well defined.
int MatGetReorderingTypeFromOptions(char *prefix,MatReordering *type) Gets matrix reordering method from the options database.
int MatGetReordering(Mat mat,MatReordering type,IS *rperm,IS *cperm) Gets a reordering for a matrix to reduce fill or to improve numerical stability of LU factorization.
int MatGetRowIJ(Mat mat,int shift,PetscTruth symmetric,int *n,int **ia,int** ja,PetscTruth *done) Returns the compress row storage i and j indices for sequential matrices. EXPERTS ONLY.
int MatGetRow(Mat mat,int row,int *ncols,int **cols,Scalar **vals) Gets a row of a matrix. You MUST call MatRestoreRow() for each row that you get to ensure that your application does not bleed memory.
int MatGetSize(Mat mat,int *m,int* n) Returns the numbers of rows and columns in a matrix.
int MatGetSubMatrices(Mat mat,int n,IS *irow,IS *icol,MatGetSubMatrixCall scall, Mat **submat) Extracts several submatrices from a matrix. If submat points to an array of valid matrices, it may be reused.
int MatGetTypeFromOptions(MPI_Comm comm,char *pre,MatType *type,int *set) Determines from the options database what matrix format the user has specified.
int MatGetType(Mat mat,MatType *type,char **name) Gets the matrix type and name (as a string) from the matrix.
int MatGetValues(Mat mat,int m,int *idxm,int n,int *idxn,Scalar *v) Gets a block of values from a matrix.
int MatHasOperation(Mat mat,MatOperation op,PetscTruth *has) Determines if the given matrix supports the particular operation.
int MatILUDTFactor(Mat mat,double dt,int maxnz,IS row,IS col,Mat *fact) Performs a drop tolerance ILU factorization.
int MatILUFactorSymbolic(Mat mat,IS row,IS col,double f,int fill,Mat *fact) Performs symbolic ILU factorization of a matrix. Uses levels of fill only, not drop tolerance. Use MatLUFactorNumeric() to complete the factorization.
int MatILUFactor(Mat mat,IS row,IS col,double f,int level) Performs in-place ILU factorization of matrix.
int MatIncompleteCholeskyFactorSymbolic(Mat mat,IS perm,double f,int fill, Mat *fact) Performs symbolic incomplete Cholesky factorization for a symmetric matrix. Use MatCholeskyFactorNumeric() to complete the factorization.
int MatIncreaseOverlap(Mat mat,int n, IS *is,int ov) Given a set of submatrices indicated by index sets, replaces the index by larger ones that represent submatrices with more overlap.

int MatLUFactorNumeric(Mat mat,Mat *fact)
Performs numeric LU factorization of a matrix. Call this routine after first calling MatLUFactorSymbolic().
int MatLUFactorSymbolic(Mat mat,IS row,IS col,double f,Mat *fact)
Performs symbolic LU factorization of matrix. Call this routine before calling MatLUFactorNumeric().
int MatLUFactor(Mat mat,IS row,IS col,double f)
Performs in-place LU factorization of matrix.
int MatLoad(Viewer viewer,MatType outtype,Mat *newmat)
Loads a matrix that has been stored in binary format with MatView(). The matrix format is determined from the options database. Generates a parallel MPI matrix if the communicator has more than one processor. The default matrix type is AIJ.
int MatMultAdd(Mat mat,Vec v1,Vec v2,Vec v3)
Computes $v3 = v2 + A * v1$.
int MatMultTransAdd(Mat mat,Vec v1,Vec v2,Vec v3)
Computes $v3 = v2 + A' * v1$.
int MatMultTrans(Mat mat,Vec x,Vec y)
Computes matrix transpose times a vector.
int MatMult(Mat mat,Vec x,Vec y)
Computes the matrix-vector product, $y = Ax$.
int MatNorm(Mat mat,NormType type,double *norm)
Calculates various norms of a matrix.
int MatPermute(Mat mat,IS row,IS col,Mat *B)
Creates a new matrix with rows and columns permuted from the original.
int MatPrintHelp(Mat mat)
Prints all the options for the matrix.
int MatRelax(Mat mat,Vec b,double omega,MatSORType flag,double shift, int its,Vec x)
Computes one relaxation sweep.
int MatReorderForNonzeroDiagonal(Mat mat,double atol,IS ris,IS cis)
Changes matrix ordering to remove zeros from diagonal. This may help in the LU factorization to prevent a zero pivot.
int MatReorderingGetName(MatReordering meth,char **name)
Gets the name associated with a reordering.
int MatReorderingRegisterAll()
Registers all of the matrix reordering routines in PETSc.
int MatReorderingRegisterDestroy()
Frees the list of ordering routines.
int MatReorderingRegister(MatReordering *name,char *sname,int (*order)(Mat,MatReordering,IS*,IS*))
Adds a new sparse matrix reordering to the matrix package.
int MatRestoreArray(Mat mat,Scalar **v)
Restores the matrix after MatGetArray has been called.
int MatRestoreColumnIJ(Mat mat,int shift,PetscTruth symmetric,int *n,int **ia,int** ja,PetscTruth *done)
Call after you are completed with the ia,ja indices obtained with MatGetColumnIJ(). EXPERTS ONLY.
int MatRestoreRowIJ(Mat mat,int shift,PetscTruth symmetric,int *n,int **ia,int** ja,PetscTruth *done)
Call after you are completed with the ia,ja indices obtained with MatGetRowIJ(). EXPERTS ONLY.
int MatRestoreRow(Mat mat,int row,int *ncols,int **cols,Scalar **vals)
Frees any temporary space allocated by MatGetRow().
int MatScale(Scalar *a,Mat mat)
Scales all elements of a matrix by a given number.
int MatSetLocalToGlobalMapping(Mat x, int n,int *indices)
Sets a local numbering to global numbering used by the routine MatSetValuesLocal() to allow users to insert matrices entries using a local (per-processor) numbering.

int MatSetOption(Mat mat, MatOption op)
Sets a parameter option for a matrix. Some options may be specific to certain storage formats. Some options determine how values will be inserted (or added). Sorted, row-oriented input will generally assemble the fastest. The default is row-oriented, nonsorted input.
int MatSetUnfactored(Mat mat)
Resets a factored matrix to be treated as unfactored.
int MatSetValuesLocal(Mat x, int nrow, int *irow, int ncol, int *icol, Scalar *y, InsertMode iora)
Inserts or adds values into certain locations of a matrix, using a local ordering of the nodes.
int MatSetValues(Mat mat, int m, int *idxm, int n, int *idxn, Scalar *v, InsertMode addv)
Inserts or adds a block of values into a matrix. These values may be cached, so MatAssemblyBegin() and MatAssemblyEnd() MUST be called after all calls to MatSetValues() have been completed.
void MatSetValue(Mat m, int row, int col, Scalar value, InsertMode mode);
Set a single entry into a matrix.
int MatShellGetContext(Mat mat, void **ctx)
Returns the user-provided context associated with a shell matrix.
int MatShellSetOperation(Mat mat, MatOperation op, void *f)
Allows user to set a matrix operation for a shell matrix.
int MatShift(Scalar *a, Mat Y)
Computes $Y = Y + a I$, where a is a scalar and I is the identity matrix.
int MatSolveAdd(Mat mat, Vec b, Vec y, Vec x)
Computes $x = y + \text{inv}(A)*b$, given a factored matrix.
int MatSolveTransAdd(Mat mat, Vec b, Vec y, Vec x)
Computes $x = y + \text{inv}(\text{trans}(A)) b$, given a factored matrix.
int MatSolveTrans(Mat mat, Vec b, Vec x)
Solves $A' x = b$, given a factored matrix.
int MatSolve(Mat mat, Vec b, Vec x)
Solves $A x = b$, given a factored matrix.
int MatTranspose(Mat mat, Mat *B)
Computes an in-place or out-of-place transpose of a matrix.
int MatValid(Mat m, PetscTruth *flag)
Checks whether a matrix object is valid.
int MatView(Mat mat, Viewer viewer)
Visualizes a matrix object.
int MatZeroEntries(Mat mat)
Zeros all entries of a matrix. For sparse matrices this routine retains the old nonzero structure.
int MatZeroRowsLocal(Mat mat, IS is, Scalar *diag)
Zeros all entries (except possibly the main diagonal) of a set of rows of a matrix; using local numbering of rows.
int MatZeroRows(Mat mat, IS is, Scalar *diag)
Zeros all entries (except possibly the main diagonal) of a set of rows of a matrix.

A.3 Simplified Linear Solvers

Data Structures:

- SLES - linear equation solver context

Runtime Options:

- -sles_view

#include "sles.h"

int SLESAppendOptionsPrefix(SLES sles,char *prefix)
Appends to the prefix used for searching for all SLES options in the database. You must include the - at the beginning of the prefix name.
int SLESCreate(MPI_Comm comm,SLES *outsles)
Creates a linear equation solver context.
int SLESDestroy(SLES sles)
Destroys the SLES context.
int SLESGetKSP(SLES sles,KSP *ksp)
Returns the KSP context for a SLES solver.
int SLESGetOptionsPrefix(SLES sles,char **prefix)
Gets the prefix used for searching for all SLES options in the database.
int SLESGetPC(SLES sles,PC *pc)
Returns the preconditioner (PC) context for a SLES solver.
int SLESPrintHelp(SLES sles)
Prints SLES options.
int SLESSetFromOptions(SLES sles)
Sets various SLES parameters from user options. Also takes all KSP and PC options.
int SLESSetOperators(SLES sles,Mat Amat,Mat Pmat,MatStructure flag)
Sets the matrix associated with the linear system and a (possibly) different one associated with the preconditioner.
int SLESSetOptionsPrefix(SLES sles,char *prefix)
Sets the prefix used for searching for all SLES options in the database. You must include the - at the beginning of the prefix name.
int SLESSetUpOnBlocks(SLES sles)
Sets up the preconditioner for each block in the block Jacobi, block Gauss-Seidel, and overlapping Schwarz methods.
int SLESSetUp(SLES sles,Vec b,Vec x)
Performs set up required for solving a linear system.
int SLESSolve(SLES sles,Vec b,Vec x,int *its)
Solves a linear system.
int SLESView(SLES sles,Viewer viewer)
Prints the SLES data structure.

A.4 Preconditioners

Data Structures:

- PC - preconditioner context

Available Methods:

- PCNONE - null preconditioner
- PCJACOBI - Jacobi
- PCSOR - SOR/SSOR and other variants
- PCEISENSTAT - SOR using the Eisenstat trick
- PCBJACOBI - block Jacobi
- PCASM - additive overlapping Schwarz
- PCILU - incomplete LU
- PCICC - incomplete Cholesky
- PCBGS - block Gauss-Seidel

- PCMG - multigrid
- PCSHELL - user-defined shell preconditioner
- PCLU - LU (direct solver)

MGType:

- MGMULTIPLICATIVE
- MGADDITIVE
- MGFULL
- MGKASKADE

Runtime Options:

- -pc_type [jacobi,bjacobi,sor,eisenstat,ilu,icc,asm,bgs,mg,lu,shell,none]
- -pc_asm_blocks [blocks]
- -pc_asm_overlap [overlap]
- -pc_bgs_blocks [blocks]
- -pc_bgs_symmetric
- -pc_bgs_truelocal
- -pc_bjacobi_blocks [blocks]
- -pc_bjacobi_truelocal
- -pc_eisenstat_diagonal_scaling
- -pc_eisenstat_omega [omega]
- -pc_icc_factorpointwise
- -pc_ilu_factorpointwise
- -pc_ilu_in_place
- -pc_ilu_levels [levels]
- -pc_ilu_preserve_row_sums
- -pc_ilu_reuse_fill
- -pc_ilu_reuse_reordering
- -pc_ilu_use_drop_tolerance
- -pc_lu_in_place
- -pc_mg_cycles [cycles]
- -pc_mg_levels [levels]
- -pc_mg_method [method]
- -pc_mg_smoothdown
- -pc_mg_smoothup
- -pc_sor_backward

- -pc_sor_its [iterations]
- -pc_sor_local_backward
- -pc_sor_local_forward
- -pc_sor_local_symmetric
- -pc_sor_omega [omega]
- -pc_sor_symmetric

#include "pc.h"

int MGCheck(PC pc)
Checks that all components of the MG structure have been set.
int MGDefaultResidual(Mat mat,Vec b,Vec x,Vec r)
Default routine to calculate the residual.
int MGGetCoarseSolve(PC pc,SLES *sles)
Gets the solver context to be used on the coarse grid.
int MGGetLevels(PC pc,int *levels)
Gets the number of levels to use with MG.
int MGGetSmootherDown(PC pc,int l,SLES *sles)
Gets the SLES context to be used as smoother before coarse grid correction (pre-smoother).
int MGGetSmootherUp(PC pc,int l,SLES *sles)
Gets the SLES context to be used as smoother after coarse grid correction (post-smoother).
int MGGetSmoother(PC pc,int l,SLES *sles)
Gets the SLES context to be used as smoother for both pre- and post-smoothing. Call both MGGetSmootherUp() and MGGetSmootherDown() to use different functions for pre- and post-smoothing.
int MGSetCyclesOnLevel(PC pc,int l,int c)
Sets the number of cycles to run on this level.
int MGSetCycles(PC pc,int n)
Sets the number of cycles to use. 1 denotes a V-cycle; 2 denotes a W-cycle. Use MGSetCyclesOnLevel() for more complicated cycling.
int MGSetInterpolate(PC pc,int l,Mat mat)
Sets the function to be used to calculate the interpolation on the lth level.
int MGSetLevels(PC pc,int levels)
Sets the number of levels to use with MG. Must be called before any other MG routine.
int MGSetNumberSmoothDown(PC pc,int n)
Sets the number of pre-smoothing steps to use on all levels. Use MGGetSmootherDown() to set different pre-smoothing steps on different levels.
int MGSetNumberSmoothUp(PC pc,int n)
Sets the number of post-smoothing steps to use on all levels. Use MGGetSmootherUp() to set different numbers of post-smoothing steps on different levels.
int MGSetResidual(PC pc,int l,int (*residual)(Mat,Vec,Vec,Vec),Mat mat)
Sets the function to be used to calculate the residual on the lth level.
int MGSetRestriction(PC pc,int l,Mat mat)
Sets the function to be used to restrict vector from level l to l-1.
int MGSetRhs(PC pc,int l,Vec c)
Sets the vector space to be used to store the right-hand side on a particular level. The user should free this space at the conclusion of multigrid use.
int MGSetR(PC pc,int l,Vec c)
Sets the vector space to be used to store the residual on a particular level. The user should free this space at the conclusion of multigrid use.
int MGSetType(PC pc,MGType form)
Determines the form of multigrid to use: multiplicative, additive, full, or the Kaskade algorithm.
int MGSetX(PC pc,int l,Vec c)
Sets the vector space to be used to store the solution on a particular level. The user should free this space at the conclusion of multigrid use.

int PCASMCreateSubdomains2D(int m,int n,int M,int N,int dof,int overlap,int *Nsub,IS **is)
Creates the index sets for the overlapping Schwarz preconditioner for a two-dimensional problem on a regular grid.
int PCASMGetSubSLES(PC pc,int *n_local,int *first_local,SLES **sles)
Gets the local SLES contexts for all blocks on this processor.
int PCASMSetLocalSubdomains(PC pc, int n, IS *is)
Sets the local subdomains (for this processor only) for the additive Schwarz preconditioner. Either all or no processors in the PC communicator must call this routine.
int PCASMSetOverlap(PC pc, int ovl)
Sets the overlap between a pair of subdomains for the additive Schwarz preconditioner. Either all or no processors in the PC communicator must call this routine.
int PCASMSetTotalSubdomains(PC pc, int N, IS *is)
Sets the subdomains for all processor for the additive Schwarz preconditioner. Either all or no processors in the PC communicator must call this routine, with the same index sets.
int PCAppendOptionsPrefix(PC pc,char *prefix)
Appends to the prefix used for searching for all PC options in the database. You must NOT include the - at the beginning of the prefix name.
int PCApplyBAorABTrans(PC pc,PCSide side,Vec x,Vec y,Vec work)
Applies the transpose of the preconditioner and operator to a vector.
int PCApplyBAorAB(PC pc, PCSide side,Vec x,Vec y,Vec work)
Applies the preconditioner and operator to a vector.
int PCApplyRichardsonExists(PC pc, PetscTruth *exists)
Determines whether a particular preconditioner has a built-in fast application of Richardson's method.
int PCApplyRichardson(PC pc,Vec x,Vec y,Vec w,int its)
Applies several steps of Richardson iteration with the particular preconditioner. This routine is usually used by the Krylov solvers and not the application code directly.
int PCApplySymmetricLeft(PC pc,Vec x,Vec y)
Applies the left part of a symmetric preconditioner to a vector.
int PCApplySymmetricRight(PC pc,Vec x,Vec y)
Applies the right part of a symmetric preconditioner to a vector.
int PCApplyTrans(PC pc,Vec x,Vec y)
Applies the transpose of preconditioner to a vector.
int PCApply(PC pc,Vec x,Vec y)
Applies the preconditioner to a vector.
int PCBGSGetSubSLES(PC pc,int *n_local,int *first_local,SLES **sles)
Gets the local SLES contexts for all blocks on this processor.
int PCBGSSetLocalBlocks(PC pc, int blocks,int *lens)
Sets the local number of blocks for the block Gauss-Seidel (BGS) preconditioner.
int PCBGSSetSymmetric(PC pc, PCBGSType flag)
Sets the BGS preconditioner to use symmetric, backward, or forward relaxation. By default, forward relaxation is used.
int PCBGSSetTotalBlocks(PC pc, int blocks,int *lens)
Sets the global number of blocks for the block Gauss-Seidel (BGS) preconditioner.
int PCBGSSetUseTrueLocal(PC pc)
Sets a flag to indicate that the block problem is associated with the linear system matrix instead of the default (where it is associated with the preconditioning matrix). That is, if the local system is solved iteratively then it iterates on the block from the matrix using the block from the preconditioner as the preconditioner for the local block.
int PCBJacobiGetSubSLES(PC pc,int *n_local,int *first_local,SLES **sles)
Gets the local SLES contexts for all blocks on this processor.
int PCBJacobiSetLocalBlocks(PC pc, int blocks,int *lens)
Sets the local number of blocks for the block Jacobi preconditioner.
int PCBJacobiSetTotalBlocks(PC pc, int blocks,int *lens)
Sets the global number of blocks for the block Jacobi preconditioner.

int PCBJacobiSetUseTrueLocal(PC pc)
Sets a flag to indicate that the block problem is associated with the linear system matrix instead of the default (where it is associated with the preconditioning matrix). That is, if the local system is solved iteratively then it iterates on the block from the matrix using the block from the preconditioner as the preconditioner for the local block.
int PCCreate(MPI_Comm comm,PC *newpc)
Creates a preconditioner context.
int PCDestroy(PC pc)
Destroys PC context that was created with PCCreate().
int PCEisenstatSetOmega(PC pc,double omega)
Sets the SSOR relaxation coefficient, omega, to use with Eisenstat's trick (where omega = 1.0 by default).
int PCEisenstatUseDiagonalScaling(PC pc)
Causes the Eisenstat preconditioner to do an additional diagonal preconditioning. For matrices with very different values along the diagonal, this may improve convergence.
int PCGetFactoredMatrix(PC pc,Mat *mat)
Gets the factored matrix from the preconditioner context. This routine is valid only for the LU, incomplete LU, Cholesky, and incomplete Cholesky methods.
int PCGetOperators(PC pc,Mat *mat,Mat *pmat,MatStructure *flag)
Gets the matrix associated with the linear system and possibly a different one associated with the preconditioner.
int PCGetOptionsPrefix(PC pc,char **prefix)
Gets the prefix used for searching for all PC options in the database.
int PCGetType(PC pc,PCType *meth,char **name)
Gets the PC method type and name (as a string) from the PC context.
int PCILUSetLevels(PC pc,int levels)
Sets the number of levels of fill to use.
int PCILUSetReuseFill(PC pc,PetscTruth flag)
When matrices with same nonzero structure are ILUDT factored, this causes later ones to use the fill computed in the initial factorization.
int PCILUSetReuseReordering(PC pc,PetscTruth flag)
When similar matrices are factored, this causes the ordering computed in the first factor to be used for all following factors; applies to both fill and drop tolerance ILUs.
int PCILUSetUseDropTolerance(PC pc,double dt,int dtcount)
The preconditioner will use an ILU based on a drop tolerance.
int PCILUSetUseInPlace(PC pc)
Tells the system to do an in-place incomplete factorization.
int PCLUSetUseInPlace(PC pc)
Tells the system to do an in-place factorization. For some implementations, for instance, dense matrices, this enables the solution of much larger problems.
int PCModifySubMatrices(PC pc,int nsub,IS *row,IS *col,Mat *submat,void *ctx)
Calls an optional user-defined routine within certain preconditioners if one has been set with PCSetModifySubMatrices().
int PCNullSpaceCreate(MPI_Comm comm, int has_cnst, int n, Vec *vecs,PCNullSpace *SP)
Creates a data-structure used to project vectors out of null spaces.
int PCNullSpaceDestroy(PCNullSpace sp)
Destroys a data-structure used to project vectors out of null spaces.
int PCNullSpaceRemove(PCNullSpace sp,Vec vec)
Removes all the components of a null space from a vector.
int PCPostSolve(PC pc,KSP ksp)
Optional post-solve phase, intended for any preconditioner-specific actions that must be performed after the iterative solve itself.
int PCPreSolve(PC pc,KSP ksp)
Optional pre-solve phase, intended for any preconditioner-specific actions that must be performed before the iterative solve itself.
int PCPrintHelp(PC pc)
Prints all the options for the PC component.

int PCRegisterAll()
Registers all of the preconditioners in the PC package.
int PCRegisterDestroy()
Frees the list of preconditioners that were registered by PCRegister().
int PCRegister(PCType name, char *sname, int (*create)(PC))
Adds the preconditioner to the preconditioner package, given a preconditioner name (PCType) and a function pointer.
int PCSORSetIterations(PC pc, int its)
Sets the number of inner iterations to be used by the SOR preconditioner. The default is 1.
int PCSORSetOmega(PC pc, double omega)
Sets the SOR relaxation coefficient, omega (where omega = 1.0 by default).
int PCSORSetSymmetric(PC pc, MatSORType flag)
Sets the SOR preconditioner to use symmetric (SSOR), backward, or forward relaxation. The local variants perform SOR on each processor. By default forward relaxation is used.
int PCSetFromOptions(PC pc)
Sets PC options from the options database. This routine must be called before PCSetUp() if the user is to be allowed to set the preconditioner method.
int PCSetModifySubMatrices(PC pc, int (*func)(PC, int, IS*, IS*, Mat*, void*), void *ctx)
Sets a user-defined routine for modifying the submatrices that arise within certain subdomain-based preconditioners. The basic submatrices are extracted from the preconditioner matrix as usual; the user can then alter these (for example, to set different boundary conditions for each submatrix) before they are used for the local solves.
int PCSetOperators(PC pc, Mat Amat, Mat Pmat, MatStructure flag)
Sets the matrix associated with the linear system and a (possibly) different one associated with the preconditioner.
int PCSetOptionsPrefix(PC pc, char *prefix)
Sets the prefix used for searching for all PC options in the database. You must NOT include the - at the beginning of the prefix name.
int PCSetType(PC ctx, PCType type)
Builds PC for a particular preconditioner.
int PCSetUpOnBlocks(PC pc)
Sets up the preconditioner for each block in the block Jacobi, block Gauss-Seidel, and overlapping Schwarz methods.
int PCSetUp(PC pc)
Prepares for the use of a preconditioner.
int PCSetVector(PC pc, Vec vec)
Sets a vector associated with the preconditioner.
int PCShellGetName(PC pc, char **name)
Gets an optional name that the user has set for a shell preconditioner.
int PCShellSetApplyRichardson(PC pc, int (*apply)(void*, Vec, Vec, Vec, int), void *ptr)
Sets routine to use as preconditioner in Richardson iteration.
int PCShellSetApply(PC pc, int (*apply)(void*, Vec, Vec), void *ptr)
Sets routine to use as preconditioner.
int PCShellSetName(PC pc, char *name)
Sets an optional name to associate with a shell preconditioner.
int PCView(PC pc, Viewer viewer)
Prints the PC data structure.

A.5 Krylov Subspace Methods

Data Structures:

- KSP - Krylov space solver context

Available Methods:

- KSPCG - Conjugate Gradient
- KSPGMRES - Generalized Minimal Residual (GMRES)
- KSPBCGS - BiCGSTAB
- KSPCGS - Conjugate Gradient Squared
- KSPTFQMR - Transpose-Free Quasi-Minimal Residual (var. 1)
- KSPTCQMR - Transpose-Free Quasi-Minimal Residual (var. 2)
- KSPCR - Conjugate Residual
- KSPLSQR - Least Squares Method
- KSPRICHARDSON - Richardson's method
- KSPCHEBYCHEV - Chebyshev method
- KSPPREONLY - shell for no KSP method

Runtime Options:

- -ksp_type [cg,cgs,bcgs,gmres,tcqmr,tfqmr,cr,richardson,chebyshev,lsqr,qcg,preonly]
- -ksp_atol [absolute_tolerance]
- -ksp_bsmonitor
- -ksp_cg_Hermitian
- -ksp_cg_symmetric
- -ksp_compute_eigenvalues
- -ksp_compute_eigenvalues_explicitly
- -ksp_divtol [divergence_tolerance]
- -ksp_eigen
- -ksp_gmres_irorthog
- -ksp_gmres_preallocate
- -ksp_gmres_restart [restart_number]
- -ksp_gmres_unmodifiedgramschmidt
- -ksp_left_pc
- -ksp_max_it [maximum_iterations]
- -ksp_monitor
- -ksp_plot_eigenvalues
- -ksp_plot_eigenvalues_explicitly
- -ksp_preres
- -ksp_richardson_scale
- -ksp_right_pc
- -ksp_rtol [relative_tolerance]

- -ksp_singmonitor
- -ksp_smonitor
- -ksp_symmetric_pc
- -ksp_truemonitor
- -ksp_type
- -ksp_xmonitor
- -ksp_xtruemonitor

#include "ksp.h"

int KSPAddOptionsChecker(int (*kspcheck)(KSP))
Adds an additional function to check for KSP options.
int KSPAppendOptionsPrefix(KSP ksp,char *prefix)
Appends to the prefix used for searching for all KSP options in the database. You must NOT include the - at the beginning of the prefix name.
int KSPBuildResidual(KSP ctx, Vec t, Vec v, Vec *V)
Builds the residual in a vector provided.
int KSPBuildSolution(KSP ctx, Vec v, Vec *V)
Builds the approximate solution in a vector provided. This routine is NOT commonly needed (see SLESSolve()).
int KSPCGSetType(KSP ksp,KSPCGType type)
Sets the variant of the conjugate gradient method to use for solving a linear system with a complex coefficient matrix. This option is irrelevant when solving a real system.
int KSPChebychevSetEigenvalues(KSP ksp,double emax,double emin)
Sets estimates for the extreme eigenvalues of the preconditioned problem.
int KSPComputeEigenvaluesExplicitly(KSP ksp,int nmax,double *r,double *c)
Computes all of the eigenvalues of the preconditioned operator using LAPACK. This is very slow but will generally provide accurate eigenvalue estimates. It will only run for small problems, say $n \leq 500$. It explicitly forms a dense matrix representing the preconditioned operator.
int KSPComputeEigenvalues(KSP ksp,int n,double *r,double *c)
Computes the extreme eigenvalue for the preconditioned operator. Called after or during KSPSolve() (SLESSolve()). This does not usually provide accurate estimates; it is only for helping people understand the convergence of iterative methods, not for eigenanalysis.
int KSPComputeExplicitOperator(KSP ksp, Mat *mat)
Computes as a dense matrix the explicit preconditioned operator. This is done by applying the operators to columns of the identity matrix.
int KSPComputeExtremeSingularValues(KSP ksp,double *emax,double *emin)
Computes the extreme singular values for the preconditioned operator. Called after or during KSPSolve() (SLESSolve()).
int KSPCreate(MPI_Comm comm,KSP *ksp)
Creates the default KSP context.
int KSPDefaultConverged(KSP ksp,int n,double rnorm,void *dummy)
Determines convergence of the iterative solvers (default code).
int KSPDefaultMonitor(KSP ksp,int n,double rnorm,void *dummy)
Print the residual norm at each iteration of an iterative solver.
int KSPDestroy(KSP ksp)
Destroys KSP context.
int KSPGMRESSetOrthogonalization(KSP ksp,int (*fcn)(KSP,int))
Sets the orthogonalization routine used by GMRES.
int KSPGMRESSetPreAllocateVectors(KSP ksp)
Causes GMRES to preallocate all its needed work vectors at initial setup rather than the default, which is to allocate them in chunks when needed.
int KSPGMRESSetRestart(KSP ksp,int max_k)
Sets the number of search directions for GMRES before restart.

int KSPGetConvergenceContext(KSP ksp, void **ctx) Gets the convergence context set with KSPSetConvergenceTest().
int KSPGetMonitorContext(KSP ksp, void **ctx) Gets the monitoring context, as set by KSPSetMonitor().
int KSPGetOptionsPrefix(KSP ksp, char **prefix) Gets the prefix used for searching for all KSP options in the database.
int KSPGetPC(KSP ksp, PC *B) Returns a pointer to the preconditioner context set with KSPSetPC().
int KSPGetPreconditionerSide(KSP ksp, PCSide *side) Gets the preconditioning side.
int KSPGetRhs(KSP ksp, Vec *r) Gets the right-hand-side vector for the linear system to be solved.
int KSPGetSolution(KSP ksp, Vec *v) Gets the location of the solution for the linear system to be solved. Note that this may not be where the solution is stored during the iterative process; see KSPBuildSolution().
int KSPGetTolerances(KSP ksp, double *rtol, double *atol, double *dtol, int *maxits) Gets the relative, absolute, divergence, and maximum iteration tolerances used by the default KSP convergence tests.
int KSPGetType(KSP ksp, KSPType *type, char **name) Gets the KSP type and method name (as a string) from the method type.
int KSPLGMonitorCreate(char *host, char *label, int x, int y, int m, int n, DrawLG *draw) Creates a line graph context for use with KSP to monitor convergence of preconditioned residual norms.
int KSPLGMonitorDestroy(DrawLG drawlg) Destroys a line graph context that was created with KSPLGMonitorCreate().
int KSPLGTrueMonitorCreate(MPI_Comm comm, char *host, char *label, int x, int y, int m, int n, DrawLG *draw) Creates a line graph context for use with KSP to monitor convergence of true residual norms (as opposed to preconditioned residual norms).
int KSPLGTrueMonitorDestroy(DrawLG drawlg) Destroys a line graph context that was created with KSPLGTrueMonitorCreate().
int KSPPrintHelp(KSP ksp) Prints all options for the KSP component.
int KSPRegisterAll() Registers all of the Krylov subspace methods in the KSP package.
int KSPRegisterDestroy() Frees the list of KSP methods that were registered by KSPRegister().
int KSPRegister(KSPType name, char *sname, int (*create)(KSP)) Adds the iterative method to the KSP package, given an iterative name (KSPType) and a function pointer.
int KSPResidual(KSP ksp, Vec vsoln, Vec vt1, Vec vt2, Vec vres, Vec vbinvf, Vec vb) Computes the residual.
int KSPRichardsonSetScale(KSP ksp, double scale) Call after KSPCreate(KSPRICHARDSON) to set the damping factor; if this routine is not called, the factor defaults to 1.0.
int KSPSetComputeEigenvalues(KSP ksp) Sets a flag so that the extreme eigenvalues values will be calculated via a Lanczos or Arnoldi process as the linear system is solved.
int KSPSetComputeResidual(KSP ksp, PetscTruth flag) Sets a flag to indicate whether the two norm of the residual is calculated at each iteration.
int KSPSetComputeSingularValues(KSP ksp) Sets a flag so that the extreme singular values will be calculated via a Lanczos or Arnoldi process as the linear system is solved.
int KSPSetConvergenceTest(KSP ksp, int (*converge)(KSP, int, double, void*), void *cctx) Sets the function to be used to determine convergence.

int KSPSetFromOptions(KSP ksp)
Sets KSP options from the options database. This routine must be called before KSPSetUp() if the user is to be allowed to set the Krylov type.
int KSPSetInitialGuessNonzero(KSP ksp)
Tells the iterative solver that the initial guess is nonzero; otherwise KSP assumes the initial guess is to be zero (and thus zeros it out before solving).
int KSPSetMonitor(KSP ksp, int (*monitor)(KSP,int,double,void*), void *mctx)
Sets the function to be called at every iteration to monitor the residual/error etc.
int KSPSetOptionsPrefix(KSP ksp,char *prefix)
Sets the prefix used for searching for all KSP options in the database. You must not include the - at the beginning of the prefix name.
int KSPSetPC(KSP ksp,PC B)
Sets the preconditioner to be used to calculate the application of the preconditioner on a vector.
int KSPSetPreconditionerSide(KSP ksp,PCSide side)
Sets the preconditioning side.
int KSPSetResidualHistory(KSP ksp, double *a, int na)
Sets the array used to hold the residual history. If set, this array will contain the residual norms computed at each iteration of the solver.
int KSPSetRhs(KSP ksp,Vec b)
Sets the right-hand-side vector for the linear system to be solved.
int KSPSetSolution(KSP ksp, Vec x)
Sets the location of the solution for the linear system to be solved.
int KSPSetTolerances(KSP ksp,double rtol,double atol,double dtol,int maxits)
Sets the relative, absolute, divergence, and maximum iteration tolerances used by the default KSP convergence testers.
int KSPSetType(KSP ksp,KSPType itmethod)
Builds KSP for a particular solver.
int KSPSetUp(KSP ksp)
Sets up the internal data structures for the later use of an iterative solver.
int KSPSetUsePreconditionedResidual(KSP ksp)
Sets a flag so that the two norm of the preconditioned residual is used rather than the true residual, in the default convergence tests.
int KSPSingularValueMonitor(KSP ksp,int n,double rnorm,void *dummy)
Prints the two norm of the true residual and estimation of the extreme eigenvalues of the preconditioned problem at each iteration.
int KSPSolve(KSP ksp, int *its)
Solves linear system; call it after calling KSPCreate(), KSPSetup(), and KSPSet*().
int KSPTrueMonitor(KSP ksp,int n,double rnorm,void *dummy)
Prints the true residual norm as well as the preconditioned residual norm at each iteration of an iterative solver.
int KSPUnwindPreconditioner(KSP ksp,Vec vsoln,Vec vt1)
Unwinds the preconditioning in the solution.
int KSPView(KSP ksp,Viewer viewer)
Prints the KSP data structure.

A.6 Nonlinear Solvers

Data Structures:

- SNES - nonlinear solver context

Available Methods:

- SNES_EQ_LS - line search for systems of nonlinear equations
- SNES_EQ_TR - trust region for systems of nonlinear equations

- SNES_UM_LS - line search for unconstrained minimization
- SNES_UM_TR - trust region for unconstrained minimization

Runtime Options:

- -snes_type [ls,tr,umtr,uums,test]
- -snes_atol [absolute_tolerance]
- -snes_eq_ls
- -snes_eq_ls_alpha [alpha]
- -snes_eq_ls_maxstep [maxstep]
- -snes_eq_ls_steptol [steptol]
- -snes_eq_tr_delta0 [delta0]
- -snes_eq_tr_delta1 [delta1]
- -snes_eq_tr_delta2 [delta2]
- -snes_eq_tr_delta3 [delta3]
- -snes_eq_tr_eta [eta]
- -snes_eq_tr_mu [mu]
- -snes_eq_tr_sigma [sigma]
- -snes_fd
- -snes_fmin
- -snes_ksp_ew_alpha [alpha]
- -snes_ksp_ew_alpha2 [alpha2]
- -snes_ksp_ew_conv [conv]
- -snes_ksp_ew_gamma [gamma]
- -snes_ksp_ew_rtol0 [rtol0]
- -snes_ksp_ew_rtolmax [rtolmax]
- -snes_ksp_ew_threshold [threshold]
- -snes_ksp_ew_version [version]
- -snes_max_funcs [maximum_function_evaluations]
- -snes_max_it [maximum_iterations]
- -snes_mf
- -snes_mf_err [err]
- -snes_mf_operator
- -snes_mf_umin [minimum_value]
- -snes_monitor
- -snes_rtol [relative_tolerance]

- -snes_smonitor
- -snes_stol [step_tolerance]
- -snes_test_display
- -snes_trtol [trust_region_tolerance]
- -snes_um_delta0
- -snes_um_eta1
- -snes_um_eta2
- -snes_um_eta3
- -snes_um_eta4
- -snes_um_factor1
- -snes_um_ls_ftol
- -snes_um_ls_gamma_factor
- -snes_um_ls_gtol
- -snes_um_ls_maxfev
- -snes_um_ls_rtol
- -snes_um_ls_stepmax
- -snes_um_ls_stepmin
- -snes_view
- -snes_xmonitor

#include "snes.h"

int SNESAddOptionsChecker(int (*snescheck)(SNES))
Adds an additional function to check for SNES options.
int SNESAppendOptionsPrefix(SNES snes,char *prefix)
Appends to the prefix used for searching for all SNES options in the database. You must NOT include the - at the beginning of the prefix name.
int SNESComputeFunction(SNES snes,Vec x, Vec y)
Computes the function that has been set with SNESSetFunction().
int SNESComputeGradient(SNES snes,Vec x, Vec y)
Computes the gradient that has been set with SNESSetGradient().
int SNESComputeHessian(SNES snes,Vec x,Mat *A,Mat *B,MatStructure *flag)
Computes the Hessian matrix that has been set with SNESSetHessian().
int SNESComputeJacobian(SNES snes,Vec X,Mat *A,Mat *B,MatStructure *flg)
Computes the Jacobian matrix that has been set with SNESSetJacobian().
int SNESComputeMinimizationFunction(SNES snes,Vec x,double *y)
Computes the function that has been set with SNESSetMinimizationFunction().
int SNESConverged_EQ_LS(SNES snes,double xnorm,double pnorm,double fnorm,void *dummy)
Monitors the convergence of the solvers for systems of nonlinear equations (default).
int SNESConverged_EQ_TR(SNES snes,double xnorm,double pnorm,double fnorm,void *dummy)
Monitors the convergence of the trust region method SNES_EQ_TR for solving systems of nonlinear equations (default).
int SNESConverged_UM_LS(SNES snes,double xnorm,double gnorm,double f, void *dummy)
Monitors the convergence of the SNESolve_UM_LS() routine (default).

int SNESConverged_UM_TR(SNES snes,double xnorm,double gnorm,double f, void *dummy) Monitors the convergence of the SNESsolve_UM_TR() routine (default).
int SNESCreate(MPI_Comm comm,SNESProblemType type,SNES *outsnes) Creates a nonlinear solver context.
int SNESCubicLineSearch(SNES snes,Vec x,Vec f,Vec g,Vec y,Vec w, double fnorm,double *ynorm,double *gnorm,int *flag) Performs a cubic line search (default line search method).
int SNESDefaultComputeHessian(SNES snes,Vec x1,Mat *J,Mat *B,MatStructure *flag,void *ctx) Computes the Hessian using finite differences.
int SNESDefaultComputeJacobianWithColoring(SNES snes,Vec x1,Mat *JJ,Mat *B,MatStructure *flag,void *ctx) nput Parameters: . snes - nonlinear solver object . x1 - location at which to evaluate Jacobian . ctx - MatFDColoring context
int SNESDefaultComputeJacobian(SNES snes,Vec x1,Mat *J,Mat *B,MatStructure *flag,void *ctx) Computes the Jacobian using finite differences.
int SNESDefaultMatrixFreeMatAddNullSpace(Mat J,int has_cnst,int n,Vec *vecs) Provides a null space that an operator is supposed to have. Since roundoff will create a small component in the null space, if you know the null space you may have it automatically removed.
int SNESDefaultMatrixFreeMatCreate(SNES snes,Vec x, Mat *J) Creates a matrix-free matrix context for use with a SNES solver. This matrix can be used as the Jacobian argument for the routine SNESSetJacobian().
int SNESDefaultMonitor(SNES snes,int its,double fgnorm,void *dummy) Monitoring progress of the SNES solvers (default).
int SNESDestroy(SNES snes) Destroys the nonlinear solver context that was created with SNESCreate().
int SNESGetApplicationContext(SNES snes, void **usrP) Gets the user-defined context for the nonlinear solvers.
int SNESGetFunctionNorm(SNES snes,Scalar *fnorm) Gets the norm of the current function that was set with SNESSetFunction().
int SNESGetFunction(SNES snes,Vec *r) Returns the vector where the function is stored.
int SNESGetGradientNorm(SNES snes,Scalar *gnorm) Gets the norm of the current gradient that was set with SNESSetGradient().
int SNESGetGradient(SNES snes,Vec *r) Returns the vector where the gradient is stored.
int SNESGetHessian(SNES snes,Mat *A,Mat *B, void **ctx) Returns the Hessian matrix and optionally the user provided context for evaluating the Hessian.
int SNESGetIterationNumber(SNES snes,int* iter) Gets the current iteration number of the nonlinear solver.
int SNESGetJacobian(SNES snes,Mat *A,Mat *B, void **ctx) Returns the Jacobian matrix and optionally the user provided context for evaluating the Jacobian.
int SNESGetMinimizationFunction(SNES snes,double *r) Returns the scalar function value for unconstrained minimization problems.
int SNESGetNumberLinearIterations(SNES snes,int* lits) Gets the total number of linear iterations used by the nonlinear solver.
int SNESGetNumberUnsuccessfulSteps(SNES snes,int* nfails) Gets the number of unsuccessful steps attempted by the nonlinear solver.
int SNESGetOptionsPrefix(SNES snes,char **prefix) Sets the prefix used for searching for all SNES options in the database.
int SNESGetSLES(SNES snes,SLES *sles) Returns the SLES context for a SNES solver.
int SNESGetSolutionUpdate(SNES snes,Vec *x) Returns the vector where the solution update is stored.

int SNESGetSolution(SNES snes,Vec *x)
Returns the vector where the approximate solution is stored.
int SNESGetTolerances(SNES snes,double *atol,double *rtol,double *stol,int *maxit,int *maxf)
Gets various parameters used in convergence tests.
int SNESGetType(SNES snes, SNESType *method,char **name)
Gets the SNES method type and name (as a string).
int SNESNoLineSearch(SNES snes, Vec x, Vec f, Vec g, Vec y, Vec w, double fnorm, double *ynorm, double *gnorm,int *flag)
This routine is not a line search at all; it simply uses the full Newton step. Thus, this routine is intended to serve as a template and is not recommended for general use.
int SNESPrintHelp(SNES snes)
Prints all options for the SNES component.
int SNESQuadraticLineSearch(SNES snes, Vec x, Vec f, Vec g, Vec y, Vec w, double fnorm, double *ynorm, double *gnorm,int *flag)
Performs a quadratic line search.
int SNESRegisterAll()
Registers all of the nonlinear solvers in the SNES package.
int SNESRegisterDestroy()
Frees the list of nonlinear solvers that were registered by SNESRegister().
int SNESRegister(int name, char *sname, int (*create)(SNES))
Adds the method to the nonlinear solver package, given a function pointer and a nonlinear solver name of the type SNESType.
int SNESSetApplicationContext(SNES snes,void *usrP)
Sets the optional user-defined context for the nonlinear solvers.
int SNESSetConvergenceHistory(SNES snes, double *a, int na)
Sets the array used to hold the convergence history.
int SNESSetConvergenceTest(SNES snes,int (*func)(SNES,double,double,double,void*),void *cctx)
Sets the function that is to be used to test for convergence of the nonlinear iterative solution.
int SNESSetFromOptions(SNES snes)
Sets various SNES and SLES parameters from user options.
int SNESSetFunction(SNES snes, Vec r, int (*func)(SNES,Vec,Vec,void*),void *ctx)
Sets the function evaluation routine and function vector for use by the SNES routines in solving systems of nonlinear equations.
int SNESSetGradient(SNES snes,Vec r,int (*func)(SNES,Vec,Vec,void*),void *ctx)
Sets the gradient evaluation routine and gradient vector for use by the SNES routines.
int SNESSetHessian(SNES snes,Mat A,Mat B,int (*func)(SNES,Vec,Mat*,Mat*, MatStructure*,void*),void *ctx)
Sets the function to compute Hessian as well as the location to store the matrix.
int SNESSetJacobian(SNES snes,Mat A,Mat B,int (*func)(SNES,Vec,Mat*,Mat*, MatStructure*,void*),void *ctx)
Sets the function to compute Jacobian as well as the location to store the matrix.
int SNESSetLineSearch(SNES snes,int (*func)(SNES,Vec,Vec,Vec,Vec,Vec, double,double*,double*,int*))
Sets the line search routine to be used by the method SNES_EQ_LS.
int SNESSetMatrixFreeParameters(SNES snes,double error,double umin)
Sets the parameters for the approximation of matrix-vector products using finite differences.
int SNESSetMinimizationFunctionTolerance(SNES snes,double ftol)
Sets the minimum allowable function tolerance for unconstrained minimization solvers.
int SNESSetMinimizationFunction(SNES snes,int (*func)(SNES,Vec,double*,void*), void *ctx)
Sets the function evaluation routine for unconstrained minimization.
int SNESSetMonitor(SNES snes, int (*func)(SNES,int,double,void*),void *mctx)
Sets the function that is to be used at every iteration of the nonlinear solver to display the iteration's progress.
int SNESSetOptionsPrefix(SNES snes,char *prefix)
Sets the prefix used for searching for all SNES options in the database. You must NOT include the - at the beginning of the prefix name.

int SNESSetTolerances(SNES snes,double atol,double rtol,double stol,int maxit,int maxf) Sets various parameters used in convergence tests.
int SNESSetTrustRegionTolerance(SNES snes,double tol) Sets the trust region parameter tolerance.
int SNESSetType(SNES snes,SNESType method) Sets the method for the nonlinear solver.
int SNESSetUp(SNES snes,Vec x) Sets up the internal data structures for the later use of a nonlinear solver.
int SNESsolve(SNES snes,Vec x,int *its) Solves a nonlinear system. Call SNESsolve after calling SNESCreate() and optional routines of the form SNESSetXXX().
int SNESView(SNES snes,Viewer viewer) Prints the SNES data structure.
int SNES_KSP_SetConvergenceTestEW(SNES snes) Sets alternative convergence test for the linear solvers within an inexact Newton method.
int SNES_KSP_SetParametersEW(SNES snes,int version,double rtol_0, double rtol_max,double gamma2,double alpha, double alpha2,double threshold) Sets parameters for Eisenstat-Walker convergence criteria for the linear solvers within an inexact Newton method.

A.7 Timestepping, ODE Solvers

Available Methods:

- TS_EULER - Euler method
- TS_BEULER - backward Euler method
- TS_PSEUDO_POSITION_INDEPENDENT_TIMESTEP - pseudo-transient timestep variant 1
- TS_PSEUDO_POSITION_DEPENDENT_TIMESTEP - pseudo-transient timestep variant 2 TSType:

Data Structures:

- TS - timestepping context

TSProblemType:

- TS_LINEAR
- TS_NONLINEAR

Runtime Options:

- -ts_max_steps [steps]
- -ts_monitor
- -ts_pseudo_increment [increment]
- -ts_type
- -ts_view

#include "ts.h"

int TSCreate(MPI_Comm comm,TSProblemType problemtype,TS *outts) Creates a timestepper context.
int TSDestroy(TS ts) Destroys the timestepper context that was created with TSCreate().

int TSGetApplicationContext(TS ts, void **usrP)
Gets the user-defined context for the timestepper.
int TSGetSLES(TS ts,SLES *sles)
Returns the SLES (linear solver) associated with a TS (timestepper) context.
int TSGetSNES(TS ts,SNES *snes)
Returns the SNES (nonlinear solver) associated with a TS (timestepper) context. Valid only for nonlinear problems.
int TSGetSolution(TS ts,Vec *v)
Returns the solution at the present timestep. It is valid to call this routine inside the function that you are evaluating in order to move to the new timestep. This vector not changed until the solution at the next timestep has been calculated.
int TSGetTimeStepNumber(TS ts,int* iter)
Gets the current number of timesteps.
int TSGetTimeStep(TS ts,double* dt)
Gets the current timestep size.
int TSGetType(TS ts, TSType *method,char **name)
Gets the TS method type and name (as a string).
int TSPrintHelp(TS ts)
Prints all options for the TS (timestepping) component.
int TSPseudoComputeTimeStep(TS ts,double *dt)
Computes the next timestep for a currently running pseudo-timestepping process.
int TSPseudoDefaultTimeStep(TS ts,double* newdt,void* dtctx)
Default code to compute pseudo-timestepping. Use with TSPseudoSetTimeStep().
int TSPseudoDefaultVerifyTimeStep(TS ts,Vec update,void *dtctx,double *newdt,int *flag)
Default code to verify the quality of the last timestep.
int TSPseudoIncrementDtFromInitialDt(TS ts)
Indicates that a new timestep is computed via $\text{initial_dt} \cdot \text{initial_fnorm} / \text{current_fnorm}$ rather than the default $\text{current_dt} \cdot \text{previous_fnorm} / \text{current_fnorm}$.
int TSPseudoSetTimeStepIncrement(TS ts,double inc)
Sets the scaling increment applied to dt when using the TSPseudoDefaultTimeStep() routine.
int TSPseudoSetTimeStep(TS ts,int (*dt)(TS,double*,void*),void* ctx)
Sets the user-defined routine to be called at each pseudo-timestep to update the timestep.
int TSPseudoSetVerifyTimeStep(TS ts,int (*dt)(TS,Vec,void*,double*,int*),void* ctx)
Sets a user-defined routine to verify the quality of the last timestep.
int TSPseudoVerifyTimeStep(TS ts,Vec update,double *dt,int *flag)
Verifies whether the last timestep was acceptable.
int TSRegisterAll()
Registers all of the timesteppers in the TS package.
int TSRegisterDestroy()
Frees the list of nonlinear solvers that were registered by TSRegister().
int TSRegister(int name, char *sname, int (*create)(TS))
Adds the method to the nonlinear solver package, given a function pointer and a nonlinear solver name of the type TSType.
int TSSetApplicationContext(TS ts,void *usrP)
Sets an optional user-defined context for the timesteppers.
int TSSetDuration(TS ts,int maxsteps,double maxtime)
Sets the maximum number of timesteps to use and maximum time for iteration.
int TSSetFromOptions(TS ts)
Sets various TS parameters from user options.
int TSSetInitialTimeStep(TS ts,double initial_time,double time_step)
Sets the initial timestep to be used, as well as the initial time.
int TSSetMonitor(TS ts, int (*monitor)(TS,int,double,Vec,void*), void *mctx)
Sets the function that is to be used at every timestep to display the iteration's progress.
int TSSetRHSFunction(TS ts,int (*f)(TS,double,Vec,Vec,void*),void *ctx)
Sets the routine for evaluating the function, $F(t,u)$, where $U_t = F(t,u)$.

int TSSetRHSJacobian(TS ts,Mat A, Mat B,int (*f)(TS,double,Vec,Mat*,Mat*, MatStructure*,void*),void *ctx) Sets the function to compute the Jacobian of F, where $U_t = F(U,t)$, as well as the location to store the matrix.
int TSSetRHSMatrix(TS ts,Mat A, Mat B,int (*f)(TS,double,Mat*,Mat*, MatStructure*,void*),void *ctx) Sets the function to compute the matrix A, where $U_t = A(t)u$. Also sets the location to store A.
int TSSetSolution(TS ts,Vec x) Sets the initial solution vector for use by the TS routines.
int TSSetTimeStep(TS ts,double time_step) Allows one to reset the timestep at any time, useful for simple pseudo-timestepping codes.
int TSSetType(TS ts,TSType method) Sets the method for the timestepping solver.
int TSSetUp(TS ts) Sets up the internal data structures for the later use of a timestepper. Call TSSetUp() after calling TSCreate() and optional routines of the form TSSetXXX(), but before calling TSStep().
int TSStep(TS ts,int *steps,double *time) Steps the requested number of timesteps.
int TSView(TS ts,Viewer viewer) Prints the TS data structure.

A.8 Index Sets, Distributed Arrays, and Application Orderings

Data Structures:

- IS - an index set
- DA - a distributed array
- AO - an application ordering

DAPeriodicType:

- DA_NONPERIODIC
- DA_XPERIODIC
- DA_YPERIODIC
- DA_XYPERIODIC
- DA_XYZPERIODIC
- DA_XZPERIODIC
- DA_YZPERIODIC
- DA_ZPERIODIC

DAS stencilType:

- DA_STENCIL_STAR
- DA_STENCIL_BOX

Runtime Options:

- -ao_view
- -da_partition_blockcomm

- -da_partition_nodes_at_end

- -da_view

```
#include "is.h"
#include "da.h"
```

int AOApplicationToPetscIS(AO ao,IS is) Maps an index set in the application-defined ordering to the PETSc ordering.
int AOApplicationToPetsc(AO ao,int n,int *ia) Maps a set of integers in the application-defined ordering to the PETSc ordering.
int AODestroy(AO ao) Destroys an application ordering set.
int AOPetscToApplicationIS(AO ao,IS is) Maps an index set in the PETSc ordering to the application-defined ordering.
int AOPetscToApplication(AO ao,int n,int *ia) Maps a set of integers in the PETSc ordering to the application-defined ordering.
int AOView(AO ao, Viewer viewer) Displays an application ordering.
int DACreate1d(MPI_Comm comm,DAPeriodicType wrap,int M,int w,int s,DA *inra) Creates a one-dimensional regular array that is distributed across some processors.
int DACreate2d(MPI_Comm comm,DAPeriodicType wrap,DASTencilType stencil_type, int M,int N,int m,int n,int w,int s,DA *inra) Creates a two-dimensional regular array that is distributed across some processors.
int DACreate3d(MPI_Comm comm,DAPeriodicType wrap,DASTencilType stencil_type, int M,int N,int P,int m,int n,int p,int w,int s,DA *inra) Creates a three-dimensional regular array that is distributed across some processors.
int DADestroy(DA da) Destroys a distributed array.
int DAGetAO(DA da, AO *ao) Gets the application ordering context for a distributed array.
int DAGetBilinearInterpolation2dBox(DA dac,DA daf,Mat *A) Gets the matrix representing bilinear interpolation from a DA grid to the next refinement.
int DAGetColoring2dBox(DA da,ISColoring *coloring,Mat *J) Gets the coloring required for computing the Jacobian via finite differences on a function defined using the nine point stencil on a two dimensional grid.
int DAGetCorners(DA da,int *x,int *y,int *z,int *m, int *n, int *p) Returns the global (x,y,z) indices of the lower left corner of the local region, excluding ghost points.
int DAGetDistributedVector(DA da,Vec* g) Gets a distributed vector for a distributed array. Additional vectors of the same type can be created with VecDuplicate().
int DAGetGhostCorners(DA da,int *x,int *y,int *z,int *m, int *n, int *p) Returns the global (x,y,z) indices of the lower left corner of the local region, including ghost points.
int DAGetGlobalIndices(DA da, int *n,int **idx) Returns the global node number of all local nodes, including ghost nodes.
int DAGetInfo(DA da,int *dim,int *M,int *N,int *P,int *m,int *n,int *p,int *w,int *s) Gets information about a given distributed array.
int DAGetLocalVector(DA da,Vec* l) Gets a local vector (including ghost points) for a distributed array. Additional vectors of the same type can be created with VecDuplicate().
int DAGetProcessorSubset(DA da,DADirection dir,int gp,MPI_Comm *comm) Returns a communicator consisting only of the processors in a DA that own a particular global x, y, or z grid point (corresponding to a logical plane in a 3D grid or a line in a 2D grid).
int DAGetScatter(DA da, VecScatter *ltog,VecScatter *gtol,VecScatter *ltol) Gets the local-to-global, local-to-global, and local-to-local vector scatter contexts for a distributed array.

int DAGlobalToLocalBegin(DA da,Vec g, InsertMode mode,Vec l)
Maps values from the global vector to the local patch; the ghost points are included. Must be followed by DAGlobalToLocalEnd() to complete the exchange.
int DAGlobalToLocalEnd(DA da,Vec g, InsertMode mode,Vec l)
Maps values from the global vector to the local patch; the ghost points are included. Must be preceded by DAGlobalToLocalBegin().
int DALocalToGlobal(DA da,Vec l, InsertMode mode,Vec g)
Maps values from the local patch back to the global vector. The ghost points are discarded.
int DALocalToLocalBegin(DA da,Vec g, InsertMode mode,Vec l)
Maps from a local vector (including ghost points that contain irrelevant values) to another local vector where the ghost points in the second are set correctly. Must be followed by DALocalToLocalEnd().
int DALocalToLocalEnd(DA da,Vec g, InsertMode mode,Vec l)
Maps from a local vector (including ghost points that contain irrelevant values) to another local vector where the ghost points in the second are set correctly. Must be preceded by DALocalToLocalBegin().
int DAPrintHelp(DA da)
Prints command line options for DA.
int DAREfine(DA da, DA *daref)
Creates a new distributed array that is a refinement of a given distributed array.
int DAView(DA da, Viewer v)
Visualizes a distributed array object.
int ISBlockGetBlockSize(IS is,int *size)
Returns the number of elements in a block.
int ISBlockGetIndices(IS in,int **idx)
Gets the indices associated with each block.
int ISBlockGetSize(IS is,int *size)
Returns the number of blocks in the index set.
int ISBlockRestoreIndices(IS is,int **idx)
Restores the indices associated with each block.
int ISBlock(IS is,PetscTruth *flag)
Checks if an index set is blocked.
int ISColoringCreate(MPI_Comm comm,int n,int *colors,ISColoring *iscoloring)
From lists (provided by each processor) of colors for each node, generate a ISColoring
int ISColoringDestroy(ISColoring iscoloring)
Destroy's a coloring context.
int ISColoringView(ISColoring iscoloring,Viewer viewer)
View's a coloring context.
int ISCreateBlock(MPI_Comm comm,int bs,int n,int *idx,IS *is)
Creates a data structure for an index set containing a list of integers. The indices are relative to entries, not blocks.
int ISCreateGeneral(MPI_Comm comm,int n,int *idx,IS *is)
Creates a data structure for an index set containing a list of integers.
int ISCreateStride(MPI_Comm comm,int n,int first,int step,IS *is)
Creates a data structure for an index set containing a list of evenly spaced integers.
int ISDestroy(IS is)
Destroys an index set.
int ISEqual(IS is1, IS is2, PetscTruth *flag)
Compares if two index sets have the same set of indices.
int ISGetIndices(IS is,int **ptr)
Returns a pointer to the indices. The user should call ISRestoreIndices() after having looked at the indices. The user should NOT change the indices.
int ISGetSize(IS is,int *size)
Returns the global length of an index set.
int ISIdentity(IS is,PetscTruth *ident)
Determines whether index set is the identity mapping.

int ISInvertPermutation(IS is,IS *isout) Creates a new permutation that is the inverse of a given permutation.
int ISLocalToGlobalMappingApplyIS(ISLocalToGlobalMapping mapping, IS is, IS *newis) Creates a new IS using the global numbering defined in an ISLocalToGlobalMapping from an IS in the local numbering.
void ISLocalToGlobalMappingApply(ISLocalToGlobalMapping mapping,int N,int *in,int *out); Takes a list of integers in local numbering and converts them to global numbering.
int ISLocalToGlobalMappingCreate(int n, int *indices,ISLocalToGlobalMapping *mapping) Creates a mapping between a local (0 to n) ordering and a global parallel ordering.
int ISLocalToGlobalMappingDestroy(ISLocalToGlobalMapping mapping) Destroys a mapping between a local (0 to n) ordering and a global parallel ordering.
int ISPermutation(IS is,PetscTruth *perm) PETSC_TRUE or PETSC_FALSE depending on whether the index set has been declared to be a permutation.
int ISRestoreIndices(IS is,int **ptr) Restores an index set to a usable state after a call to ISGetIndices().
int ISSetIdentity(IS is) Informs the index set that it is an identity.
int ISSetPermutation(IS is) Informs the index set that it is a permutation.
int ISSorted(IS is, PetscTruth *flag) Checks the indices to determine whether they have been sorted.
int ISSort(IS is) Sorts the indices of an index set.
int ISStrideGetInfo(IS is,int *first,int *step) Returns the first index in a stride index set and the stride width.
int ISStride(IS is,PetscTruth *flag) Determines if an IS is based on a stride.
int ISView(IS is, Viewer viewer) Displays an index set.

A.9 Utility and System Routines

Runtime Options:

- -debugger_nodes [nodes]
- -debugger_pause [seconds]
- -fp_trap
- -help (or -h)
- -log_history
- -mpidump
- -no_signal_handler
- -on_error_abort
- -on_error_attach_debugger
- -on_error_stop
- -optionsleft
- -optionstable

- -trdebug
- -trdump
- -trinfo
- -trmalloc
- -trmalloc_off
- -version (or -v)

```
#include "sys.h"
#include "options.h"
```

int OptionsAllUsed()
Returns a count of the number of options in the database that have never been selected.
int OptionsGetDoubleArray(char* pre,char *name,double *dvalue, int *nmax,int *flag)
Gets an array of double precision values for a particular option in the database. The values must be separated with commas with no intervening spaces.
int OptionsGetDouble(char* pre,char *name,double *dvalue,int *flag)
Gets the double precision value for a particular option in the database.
int OptionsGetIntArray(char* pre,char *name,int *dvalue,int *nmax,int *flag)
Gets an array of integer values for a particular option in the database. The values must be separated with commas with no intervening spaces.
int OptionsGetInt(char*pre,char *name,int *ivalue,int *flag)
Gets the integer value for a particular option in the database.
int OptionsGetScalar(char* pre,char *name,Scalar *dvalue,int *flag)
Gets the scalar value for a particular option in the database. At the moment can get only a Scalar with 0 imaginary part.
int OptionsGetString(char *pre,char *name,char *string,int len, int *flag)
Gets the string value for a particular option in the database.
int OptionsHasName(char* pre,char *name,int *flag)
Determines whether a certain option is given in the database.
int OptionsPrint(FILE *fd)
Prints the options that have been loaded. This is useful for debugging purposes.
int OptionsSetValue(char *name,char *value)
Sets an option name-value pair in the options database, overriding whatever is already present.
int PetscAbortErrorHandler(int line,char *function,char *file,char* dir,int number,int p, char *message,void *ctx)
Error handler that calls abort on error. This routine is very useful when running in the debugger, because the user can look directly at the stack frames and the variables.
int PetscAttachDebuggerErrorHandler(int line,char* fun,char *file,char* dir,int num,int p, char* mess,void *ctx)
Error handler that attaches a debugger to a running process when an error is detected. This routine is useful for examining variables, etc.
int PetscAttachDebugger()
Attaches the debugger to the running process.
int PetscBinaryRead(int fd,void *p,int n,PetscBinaryType type)
Reads from a binary file.
int PetscBinaryWrite(int fd,void *p,int n,PetscBinaryType type,int istemp)
Writes to a binary file.
int PetscCObjectToFortranObject(void *cobj,int *fobj)
Converts a PETSc object represented in C to one appropriate to pass to a Fortran routine.
int PetscCompareDouble(double d)
Compares doubles while running with PETSc's incremental debugger; triggered with the -compare ;tol; flag.

int PetscCompareInt(int d)
Compares intss while running with PETSc's incremental debugger; triggered with the -compare option.
int PetscCompareScalar(Scalar d)
Compares scalars while running with PETSc's incremental debugger; triggered with the -compare ;tol; flag.
int PetscDefaultSignalHandler(int sig, void *ptr)
Default signal handler.
int PetscDoubleView(int N,double* idx,Viewer viewer)
Prints an array of double, useful for debugging.
int PetscError(int line,char *function,char* file,char *dir,int number,int p,char *message)
Routine that is called when an error has been detected, usually called through the macro SETERRQ().
int PetscFClose(MPI_Comm comm,FILE *fd)
Has the first processor in the communicator close a file; all others do nothing.
FILE *PetscFOpen(MPI_Comm comm,char *name,char *mode)
Has the first process in the communicator open a file; all others do nothing.
int PetscFPrintf(MPI_Comm comm,FILE* fd,char *format,...)
Prints to a file, only from the first processor in the communicator.
int PetscFinalize()
Checks for options to be called at the conclusion of the program and calls MPI_Finalize().
int PetscFortranObjectToCObject(int fobj,void *cobj)
Converts a PETSc object represented in Fortran to one appropriate for C.
int PetscGetArchType(char *str,int slen)
Returns a standardized architecture type for the machine that is executing this routine.
int PetscGetFileFromPath(char *path,char *defname,char *name,char *fname, char mode)
Finds a file from a name and a path string. A default can be provided.
int PetscGetFullPath(char *path, char *fullpath, int flen)
Given a filename, returns the fully qualified file name.
int PetscGetHomeDirectory(int maxlen,char *dir)
Returns user's home directory name.
int PetscGetHostName(char *name, int nlen)
Returns the name of the host. This attempts to return the entire Internet name. It may not return the same name as MPI_Get_processor_name().
int PetscGetRealPath(char * path, char *rpath)
Get the path without symbolic links etc. and in absolute form.
int PetscGetRelativePath(char *fullpath, char *path, int flen)
Given a filename, returns the relative path (removes all directory specifiers).
int PetscGetUserName(char *name, int nlen)
Returns the name of the user.
int PetscGetWorkingDirectory(char *path,int len)
Gets the current working directory.
void PetscInitializeFortran()
Routine that should be called from C after the call to PetscInitialize() if one is using a C main program that calls Fortran routines that call PETSc routines.
int PetscInitializeLargeInts(int *p,int n)
Intializes an array of integers with very large values.
int PetscInitializeNans(Scalar *p,int n)
Intialize certain memory locations with NaNs. This routine is used to mark an array as uninitialized so that if values are used for computation without first having been set, a floating point exception is generated.
int PetscInitialize(int *argc,char ***args,char *file,char *help)
Initializes the PETSc database and MPI. PetscInitialize calls MPI_Init() if that has yet to be called, so this routine should always be called near the beginning of your program – usually the very first line!
int PetscIntView(int N,int* idx,Viewer viewer)
Prints an array of integers, useful for debugging.
int PetscMPIDump(FILE *fd)
Dumps a listing of incomplete MPI operations, such as sends that have never been received, etc.

int PetscMemcmp(void * str1, void *str2, int len)
Compares two byte streams in memory.
void PetscMemcpy(void *a,void *b,int n)
Copies n bytes, beginning at location b, to the space beginning at location a.
void PetscMemzero(void *a,int n)
Zeros the specified memory.
int PetscObjectDestroy(PetscObject obj)
Destroys any PetscObject, regardless of the type. This routine should seldom be needed.
int PetscObjectExists(PetscObject obj,int *exists)
Determines whether a PETSc object has been destroyed.
int PetscObjectGetChild(PetscObject obj,void **child)
Gets the child of any PetscObject.
int PetscObjectGetComm(PetscObject obj,MPI_Comm *comm)
Gets the MPI communicator for any PetscObject, regardless of the type.
int PetscObjectGetCookie(PetscObject obj,int *cookie)
Gets the cookie for any PetscObject,
int PetscObjectGetName(PetscObject obj,char **name)
Gets a string name associated with a PETSc object.
int PetscObjectGetNewTag(PetscObject obj,int *tag)
Gets a unique new tag from a PETSc object. All processors that share the object MUST call this routine EXACTLY the same number of times. This tag should only be used with the current object's communicator; do NOT use it with any other MPI communicator.
int PetscObjectGetType(PetscObject obj,int *type)
Gets the object type of any PetscObject.
int PetscObjectInherit(PetscObject obj,void *ptr, int (*copy)(void *,void **), int (*destroy)(void*))
Associate another object with a given PETSc object. This is to provide a limited support for inheritance.
int PetscObjectReference(PetscObject obj)
Indicate to any PetscObject that it is being referenced in another PetscObject. This increases the reference count for that object by one.
int PetscObjectRestoreNewTag(PetscObject obj,int *tag)
Restores a new tag from a PETSc object. All processors that share the object MUST call this routine EXACTLY the same number of times.
int PetscObjectSetName(PetscObject obj,char *name)
Sets a string name associated with a PETSc object.
int PetscPopErrorHandler()
Removes the latest error handler that was pushed with PetscPushErrorHandler().
int PetscPrintf(MPI_Comm comm,char *format,...)
Prints to standard out, only from the first processor in the communicator.
int PetscPushErrorHandler(int (*handler)(int,char *,char*,char*,int,int,char*,void*),void *ctx)
Sets a routine to be called on detection of errors.
int PetscPushSignalHandler(int (*routine)(int, void*),void* ctx)
Catches the usual fatal errors and calls a user-provided routine.
int PetscRandomCreate(MPI_Comm comm,PetscRandomType type,PetscRandom *r)
Creates a context for generating random numbers, and initializes the random-number generator.
int PetscRandomDestroy(PetscRandom r)
Destroys a context that has been formed by PetscRandomCreate().
int PetscRandomGetValue(PetscRandom r,Scalar *val)
Generates a random number. Call this after first calling PetscRandomCreate().
int PetscRandomSetInterval(PetscRandom r,Scalar low,Scalar high)
Sets the interval over which the random numbers will be randomly distributed. By default, this interval is [0,1).

int PetscRegisterCookie(int *cookie)
Registers a new cookie for use with a newly created PETSc object class. The user should pass in a variable initialized to zero; then it will be assigned a cookie. Repeated calls to this routine with the same variable will not change the cookie.
int PetscRemoveHomeDirectory(char *path)
Given a complete true path, removes the user's home directory.
int PetscSequentialPhaseBegin(MPI_Comm comm,int ng)
Begins a sequential section of code.
int PetscSequentialPhaseEnd(MPI_Comm comm,int ng)
Ends a sequential section of code.
int PetscSetCommWorld(MPI_Comm comm)
Sets a communicator to be PETSc's world communicator (default is MPI_COMM_WORLD). Must call BEFORE PetscInitialize().
int PetscSetDebugger(char *debugger, int xterm,char *display)
Sets options associated with the debugger.
int PetscSetFPTrap(int flag)
Enables traps/exceptions on common floating point errors. This option may not work on certain machines.
int PetscSetMalloc(void *(*imalloc)(unsigned int,int,char*,char*,char*), int (*ifree)(void*,int,char*,char*,char*))
Sets the routines used to do mallocs and frees. This routine MUST be called before PetscInitialize() and may be called only once.
void PetscSleep(int s)
Sleeps some number of seconds.
int PetscSortDoubleWithPermutation(int n, double *i, int *idx)
Computes the permutation of values that gives a sorted sequence.
int PetscSortDouble(int n,double *v)
Sorts an array of doubles in place in increasing order.
int PetscSortIntWithPermutation(int n, int *i, int *idx)
Computes the permutation of values that gives a sorted sequence.
int PetscSortInt(int n, int *i)
Sorts an array of integers in place in increasing order.
int PetscStopErrorHandler(int line,char *fun,char *file,char *dir,int number,int p, char *message,void *ctx)
Calls MPLabort() and exists.
int PetscSynchronizedFlush(MPI_Comm comm)
Flushes to the screen output from all processors involved in previous PetscSynchronizedPrintf() calls.
int PetscSynchronizedPrintf(MPI_Comm comm,char *format,...)
Prints output from several processors that is synchronized so that printed by first processor is followed by second etc.
int PetscTrDump(FILE *fp)
Dumps the allocated memory blocks to a file. The information printed is: size of space (in bytes), address of space, id of space, file in which space was allocated, and line number at which it was allocated.
int PetscTrLogDump(FILE *fp)
Dumps the log of all calls to malloc.
int PetscTrLog()
Indicates that you wish all calls to malloc to be logged.
int PetscTrSpace(double *space, double *fr, double *maxs)
Returns space statistics.
int PetscTraceBackErrorHandler(int line,char *fun,char* file,char *dir,int number,int p, char *message,void *ctx)
Default error handler routine that generates a traceback on error detection.

A.10 Viewers

Default Viewers:

- VIEWER_STDOUT_WORLD
- VIEWER_STDOUT_SELF
- VIEWER_DRAWX_WORLD
- VIEWER_DRAWX_SELF
- VIEWER_MATLAB_WORLD

Format options:

- ASCII_FORMAT_DEFAULT - default
- ASCII_FORMAT_MATLAB - Matlab format
- ASCII_FORMAT_IMPL - implementation-specific format (which is, in many cases, the same as the default)
- ASCII_FORMAT_INFO - basic information about object
- ASCII_FORMAT_INFO_DETAILED - more detailed info about object
- ASCII_FORMAT_COMMON - identical output format for all objects of a particular type
- BINARY_FORMAT_NATIVE - store the object to disk in the format it is in. This currently works only for dense matrices.

int ViewerASCIIGetPointer(Viewer viewer, FILE **fd) Extracts the file pointer from an ASCII viewer.
int ViewerBinaryGetDescriptor(Viewer viewer,int *fdes) Extracts the file descriptor from a viewer.
int ViewerBinaryGetInfoPointer(Viewer viewer,FILE **file) Extracts the file pointer for the ASCII info file associated with a binary file.
int ViewerDestroy(Viewer v) Destroys a viewer.
int ViewerFileOpenASCII(MPI_Comm comm,char *name,Viewer *lab) Opens an ASCII file as a viewer.
int ViewerFileOpenBinary(MPI_Comm comm,char *name,ViewerBinaryType type,Viewer *binv) Opens a file for binary input/output.
int ViewerFlush(Viewer v) Flushes a viewer (i.e. tries to dump all the data that has been printed through a viewer).
int ViewerGetType(Viewer v,ViewerType *type) Returns the type of a viewer.
int ViewerMatlabOpen(MPI_Comm comm,char *machine,int port,Viewer *lab) Opens a connection to a Matlab server.
int ViewerPopFormat(Viewer v) Resets the format for file viewers.
int ViewerPushFormat(Viewer v,int format,char *name) Sets the format for file viewers.
int ViewerSetFormat(Viewer v,int format,char *name) Sets the format for file viewers.
int ViewerStringOpen(MPI_Comm comm,char *string,int len, Viewer *lab) Opens a string as a viewer. This is a very simple viewer; information on the object is simply stored into the string in a fairly nice way.
int ViewerStringPrintf(Viewer v,char *format,...) Prints information to a viewer string.

A.11 Profiling

Runtime Options:

- -log [filename]
- -log_summary
- -log_all [filename]
- -log_mpe [filename]

int PLogAllBegin()	Turns on extensive logging of objects and events. Logs all events. This creates large log files and slows the program down.
int PLogBegin()	Turns on logging of objects and events. This logs flop rates and object creation and should not slow programs down too much. This routine may be called more than once.
int PLogDestroy()	Destroys the object and event logging data and resets the global counters.
int PLogDump(char* sname)	Dumps logs of objects to a file. This file is intended to be read by petsc/bin/petscview.
int PLogEventActivateClass(int cookie)	Activates event logging for a PETSc object class.
int PLogEventActivate(int event)	Indicates that a particular event should be logged. Note: the event may be either a pre-defined PETSc event (found in include/petsclog.h) or an event number obtained with PLogEventRegister().
void PLogEventBegin(int e,PetscObject o1,PetscObject o2,PetscObject o3, PetscObject o4)	Logs the beginning of a user event.
int PLogEventDeactivateClass(int cookie)	Deactivates event logging for a PETSc object class.
int PLogEventDeactivate(int event)	Indicates that a particular event should not be logged. Note: the event may be either a pre-defined PETSc event (found in include/petsclog.h) or an event number obtained with PLogEventRegister().
void PLogEventEnd(int e,PetscObject o1,PetscObject o2,PetscObject o3, PetscObject o4)	Log the end of a user event.
int PLogEventMPEActivate(int event)	Indicates that a particular event should be logged using MPE. Note: the event may be either a pre-defined PETSc event (found in include/petsclog.h) or an event number obtained with PLogEventRegister().
int PLogEventMPEDeactivate(int event)	Indicates that a particular event should not be logged using MPE. Note: the event may be either a pre-defined PETSc event (found in include/petsclog.h) or an event number obtained with PLogEventRegister().
int PLogEventRegister(int *e,char *string,char *color)	Registers an event name for logging operations in an application code.
void PLogFlops(int f)	Adds floating point operations to the global counter.
int PLogInfoActivateClass(int objclass)	Activates PlogInfo() messages for a PETSc object class.
int PLogInfoAllow(PetscTruth flag)	Causes PLogInfo() messages to be printed to standard output.
int PLogInfoDeactivateClass(int objclass)	Deactivates PlogInfo() messages for a PETSc object class.
int PLogInfo(void *vobj,char *message,...)	Logs informative data, which is printed to standard output when the option -log_info is specified.
int PLogMPEBegin()	Turns on MPE logging of events. This creates large log files and slows the program down.

int PLogMPEDump(char* sname)
Dumps the MPE logging info to file for later use with Upshot.
int PLogPrintSummary(MPI_Comm comm,FILE *fd)
Prints a summary of the logging.
int PLogSet(int (*b)(int,int,PetscObject,PetscObject,PetscObject,PetscObject), int (*e)(int,int,PetscObject,PetscObject,PetscObject,PetscObject))
Sets the logging functions called at the beginning and ending of every event.
int PLogStagePop()
Users can log up to 10 stages within a code by using -log_summary in conjunction with PLogStagePush() and PLogStagePop().
int PLogStagePush(int stage)
Users can log up to 10 stages within a code by using -log_summary in conjunction with PLogStagePush() and PLogStagePop().
int PLogStageRegister(int stage, char *sname)
Attaches a character string name to a logging stage.
int PLogTraceBegin(FILE *file)
Activates trace logging. Every time a PETSc event begins or ends, the event name is printed.
void PetscBarrier(PetscObject obj)
Blocks Until this routine is executed by all processors owning the object A.
double PetscGetFlops()
Returns the number of flops used on this processor since the program began.
double PetscGetTime()
Returns the current time of day in seconds. This returns wall-clock time.

A.12 Graphics Routines

Data Structures:

- Draw - a drawing surface, probably a window.
- DrawAxis - a two-dimensional line graph axis.
- DrawLG - a two-dimensional line graph.

#include "draw.h"

int DrawAppendTitle(Draw draw,char *title)
Appends to the title of a Draw context.
int DrawAxisCreate(Draw win,DrawAxis *ctx)
Generate the axis data structure.
int DrawAxisDestroy(DrawAxis ad)
Frees the space used by an axis structure.
int DrawAxisDraw(DrawAxis ad)
Draws an axis.
int DrawAxisSetColors(DrawAxis ad,int ac,int tc,int cc)
Sets the colors to be used for the axis, tickmarks, and text.
int DrawAxisSetLabels(DrawAxis ad,char* top,char *xlabel,char *ylabel)
Sets the x and y axis labels.
int DrawAxisSetLimits(DrawAxis ad,double xmin,double xmax,double ymin,double ymax)
Sets the limits (in user coords) of the axis
int DrawBOP(Draw draw)
Begins a new page or frame on the selected graphical device.
int DrawCheckResizedWindow(Draw draw)
Checks if the user has resized the window.
int DrawClear(Draw draw)
Clears graphical output.

int DrawCreatePopUp(Draw draw,Draw *popup) Creates a popup window associated with a Draw window.
int DrawDestroy(Draw draw) Deletes a draw context.
int DrawEOP(Draw draw) Ends a page or frame on the selected graphical device.
int DrawFlush(Draw draw) Flushs graphical output.
int DrawGetCoordinates(Draw draw,double *xl,double *yl,double *xr,double *yr) Gets the application coordinates of the corners of the window (or page).
int DrawGetMouseButton(Draw draw,DrawButton *button,double* x_user,double *y_user, double *x_phys,double *y_phys) Returns location of mouse and which button was pressed. Waits for button to be pressed.
int DrawGetPause(Draw draw,int *pause) Gets the amount of time that program pauses after a DrawPause() is called.
int DrawGetTitle(Draw draw,char **title) Gets pointer to title of a Draw context.
int DrawIsNull(Draw draw,PetscTruth *yes) Returns PETSC_TRUE if draw is a null draw object.
int DrawLGAddPoints(DrawLG lg,int n,double **xx,double **yy) Adds several points to each of the line graphs. The new points must have an X coordinate larger than the old points.
int DrawLGAddPoint(DrawLG lg,double *x,double *y) Adds another point to each of the line graphs. The new point must have an X coordinate larger than the old points.
int DrawLGCreate(Draw win,int dim,DrawLG *outctx) Creates a line graph data structure.
int DrawLGDestroy(DrawLG lg) Frees all space taken up by line graph data structure.
int DrawLGDraw(DrawLG lg) Redraws a line graph.
int DrawLGGetAxis(DrawLG lg,DrawAxis *axis) Gets the axis context associated with a line graph. This is useful if one wants to change some axis property, such as labels, color, etc. The axis context should not be destroyed by the application code.
int DrawLGGetDraw(DrawLG lg,Draw *win) Gets the draw context associated with a line graph.
int DrawLGIndicateDataPoints(DrawLG lg) Causes LG to draw a big dot for each data-point.
int DrawLGReset(DrawLG lg) Clears line graph to allow for reuse with new data.
int DrawLGSetDimension(DrawLG lg,int dim) Change the number of lines that are to be drawn.
int DrawLGSetLimits(DrawLG lg,double x_min,double x_max,double y_min, double y_max) Sets the axis limits for a line graph. If more points are added after this call, the limits will be adjusted to include those additional points.
int DrawLineGetWidth(Draw draw,double *width) Gets the line width for future draws. The width is relative to the user coordinates of the window; 0.0 denotes the natural width; 1.0 denotes the interior viewport.
int DrawLineSetWidth(Draw draw,double width) Sets the line width for future draws. The width is relative to the user coordinates of the window; 0.0 denotes the natural width; 1.0 denotes the entire viewport.
int DrawLine(Draw draw,double xl,double yl,double xr,double yr,int cl) Draws a line onto a drawable.
int DrawOpenVRML(MPIComm comm,char* fname,char *title, Draw* inctx) Opens an VRML viewer for use with the Draw routines.

int DrawOpenX(MPI_Comm comm,char* display,char *title,int x,int y,int w,int h, Draw* inctx) Opens an X-window for use with the Draw routines.
int DrawPause(Draw draw) Waits n seconds or until user input, depending on input to DrawSetPause().
int DrawPointSetSize(Draw draw,double width) Sets the point size for future draws. The size is relative to the user coordinates of the window; 0.0 denotes the natural width, 1.0 denotes the entire viewport.
int DrawPoint(Draw draw,double xl,double yl,int cl) Draws a point onto a drawable.
int DrawRectangle(Draw draw,double xl,double yl,double xr,double yr, int c1, int c2,int c3,int c4) Draws a rectangle onto a drawable.
int DrawSPAddPoints(DrawSP sp,int n,double **xx,double **yy) Adds several points to each of the scatter plots.
int DrawSPAddPoint(DrawSP sp,double *x,double *y) Adds another point to each of the scatter plots.
int DrawSPCreate(Draw win,int dim,DrawSP *outctx) Creates a scatter plot data structure.
int DrawSPDestroy(DrawSP sp) Frees all space taken up by scatter plot data structure.
int DrawSPDraw(DrawSP sp) Redraws a scatter plot.
int DrawSPGetAxis(DrawSP sp,DrawAxis *axis) Gets the axis context associated with a line graph. This is useful if one wants to change some axis property, such as labels, color, etc. The axis context should not be destroyed by the application code.
int DrawSPGetDraw(DrawSP sp,Draw *win) Gets the draw context associated with a line graph.
int DrawSPReset(DrawSP sp) Clears line graph to allow for reuse with new data.
int DrawSPSetDimension(DrawSP sp,int dim) Change the number of sets of points that are to be drawn.
int DrawSPSetLimits(DrawSP sp,double x_min,double x_max,double y_min, double y_max) Sets the axis limits for a line graph. If more points are added after this call, the limits will be adjusted to include those additional points.
int DrawSetCoordinates(Draw draw,double xl,double yl,double xr, double yr) Sets the application coordinates of the corners of the window (or page).
int DrawSetDoubleBuffer(Draw draw) Sets a window to be double buffered.
int DrawSetPause(Draw draw,int pause) Sets the amount of time that program pauses after a DrawPause() is called.
int DrawSetTitle(Draw draw,char *title) Sets the title of a Draw context.
int DrawSetViewport(Draw draw,double xl,double yl,double xr,double yr) Sets the portion of the window (page) to which draw routines will write.
int DrawSyncClear(Draw draw) Clears graphical output. All processors must call this routine. Does not return until the drawable is clear.
int DrawSyncFlush(Draw draw) Flushes graphical output. This waits until all processors have arrived and flushed, then does a global flush. This is usually done to change the frame for double buffered graphics.
int DrawTextGetSize(Draw draw,double *width,double *height) Gets the size for character text. The width is relative to the user coordinates of the window; 0.0 denotes the natural width; 1.0 denotes the entire viewport.
int DrawTextSetSize(Draw draw,double width,double height) Sets the size for character text. The width is relative to the user coordinates of the window; 0.0 denotes the natural width; 1.0 denotes the entire viewport.

int DrawTextVertical(Draw draw,double xl,double yl,int cl,char *text) Draws text onto a drawable.
int DrawText(Draw draw,double xl,double yl,int cl,char *text) Draws text onto a drawable.
int DrawTriangle(Draw draw,double x1,double y1,double x2,double y2, double x3,double y3,int c1, int c2,int c3) Draws a triangle onto a drawable.
Viewer VIEWER_DRAWX_(MPI_Comm comm) Creates a window viewer shared by all processors in a communicator.
int ViewerDrawGetDrawLG(Viewer v, DrawLG *drawlg) Returns DrawLG object from Viewer object. This DrawLG object may then be used to perform graphics using DrawLGXXX() commands.
int ViewerDrawGetDraw(Viewer v, Draw *draw) Returns Draw object from Viewer object. This Draw object may then be used to perform graphics using DrawXXX() commands.
int ViewerDrawOpenVRML(MPI_Comm comm,char* fname,char *title,Viewer *viewer) Opens an VRML file for use as a viewer. If you want to do graphics in this window, you must call ViewerDrawGetDraw() and perform the graphics on the Draw object.
int ViewerDrawOpenX(MPI_Comm comm,char* display,char *title,int x,int y, int w,int h,Viewer *viewer) Opens an X window for use as a viewer. If you want to do graphics in this window, you must call ViewerDrawGetDraw() and perform the graphics on the Draw object.

Acknowledgments

We thank Victor Eijkhout for his valuable comments on this manual as well as on the source code for PETSc 2.0. We also thank David Keyes for his insightful suggestions about increased functionality. In addition, we thank all our users of PETSc for their suggestions, bug reports, support, and encouragement.

Some of the source code and utility routines in PETSc were written by Cameron Cooper, Matt Hille, Peter Mell, and Wing-Lok Wan while visiting Argonne National Laboratory as summer research students.

PETSc uses routines from BLAS, LAPACK, MINPACK, SPARSPAK, and BlockSolve95 to provide a small subset of its low-level functionality.

Bibliography

- [1] Tcl/Tk World Wide Web page. <http://www.sunlabs.com/research/tcl/>, August 1996.
- [2] Peter N. Brown and Youcef Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.
- [3] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
- [4] S. Eisenstat. Efficient implementation of a class of CG methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.
- [5] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact Newton method. Preprint Utah State University Math. Stat. Dept. Res. Report 6/94/75, Logan, UT, 1994. (to appear in *SIAM J. Sci. Comput.*).
- [6] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica. Cambridge University Press, 1992.
- [7] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Stat. Comput.*, 14:470–482, 1993.
- [8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. MPICH home page. <http://www.mcs.anl.gov/mpi/mpich/index.html>, December 1996.
- [9] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, August 1991.
- [10] 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.
- [11] Mark T. Jones and Paul E. Plassmann. BlockSolve v1.1: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-92/46, Argonne National Laboratory, 1992.
- [12] 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*, pages 88–111, 1984.
- [13] 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.
- [14] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.
- [15] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.
- [16] Barry F. Smith, Petter Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

- [17] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.
- [18] Trond Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM J. Numer. Anal.*, 20:626–637, 1983.
- [19] 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.

Index

- aggregation, 114
- AIJ matrix format, 41
- alias, 118
- Arnoldi, 52
- array, distributed, 27
- ASM, 56
- assembly, 21
- axis, drawing, 85

- backward Euler, 77
- block diagonal matrix storage, 140
- block Gauss-Seidel, 56
- block Jacobi, 56, 119
- BlockSolve95, 54
- boundary conditions, 46

- C++, 124
- Cai, Xiao-Chuan, 56
- CG, 49
- Cholesky, 80
- coarse grid solve, 57
- collective operations, 11
- command line arguments, 6
- command line options, 118
- communicator, 6, 52, 137
- compiler options, 114
- complex numbers, 10, 122, 124
- convergence tests, 51, 66
- coordinates, 84
- CSR, compressed sparse row format, 41
- ctags, in VI, 123

- debugging, 6, 120
- direct solver, 55
- distributed array, 27
- double buffer, 85

- eigenvalues, 52
- Eisenstat trick, 55
- Emacs, 122
- errors, 120
- etags, in Emacs, 122
- Euler, 77

- factorization, 80
- floating-point exceptions, 121
- flushing, graphics, 85

- gather, 24
- ghost points, 26, 27
- global representation, 26
- GMRES, 49
- gradient, 60
- Gram-Schmidt, 50
- graphics, 84
- GUI utilities, 129

- Hessian, 60
- Hessian, debugging, 68

- IEEE floating point, 121
- incremental debugging, 122
- index sets, 23
- inexact Newton methods, 68
- installing PETSc, 5

- Jacobi, 56
- Jacobian, 60
- Jacobian, debugging, 68
- Jacobian, testing, 68

- Krylov subspace methods, 48, 49

- Lanczo, 52
- line graphs, 85
- line search, 59, 66
- linear system solvers, 48
- lines, drawing, 85
- local linear solves, 56
- local representation, 26
- local to global mapping, 26
- logging, 105, 114
- LU, 80

- man pages, 5
- matrices, 10, 40
- matrix reordering, 81
- matrix-free Jacobians, 69
- matrix-free methods, 46, 48
- MPI, 123
- multigrid, 57

- nested dissection, 55, 80
- Newton-like methods, 59
- nonlinear equation solvers, 59
- Nupshot, 107

- ODE solvers, 76
- one-way dissection, 55, 80
- options, 6, 118
- orderings, 25, 26, 54, 55
- overlapping Schwarz, 56

- performance tuning, 114
- petsc-maint, 141
- preconditioners, 53
- preconditioning, 48, 50
- profiling, 105, 114

- quotient minimum degree, 55, 80

- relaxation, 55, 57
- reorder, 80
- reordering, 81
- restart, 50
- reverse Cuthill-McKee, 55, 80
- Richardson's method, 83
- running PETSc programs, 5

- scatter, 24
- signals, 121
- smoothing, 57
- SOR, 55
- SPARSKIT, 42
- SPARSPAK, 141
- spectrum, 52
- SSOR, 55
- stride, 23
- symbolic factorization, 81

- text, drawing, 85
- time, 110
- timing, 105, 114
- trust region, 59, 66

- Upshot, 107

- V-cycle, 57
- vector values, getting, 25
- vector values, setting, 22
- vectors, 10, 21
- VI, 123
- visualizing program activity, 129

- W-cycle, 57
- wall clock time, 110

- X windows, 84

Index

- compare, 122
- draw_pause, 85
- draw_x_private_colormap, 84
- fp_trap, 6, 121
- h, 6
- help, 6
- ksp_atol, 51
- ksp_bsmonitor, 54
- ksp_cancelmonitors, 52
- ksp_compute_eigenvalues, 52
- ksp_compute_eigenvalues_explicitly, 52
- ksp_divtol, 51
- ksp_gmres_iorthog, 50
- ksp_gmres_restart, 50
- ksp_max_it, 51
- ksp_monitor, 51, 52
- ksp_plot_eigenvalues, 52
- ksp_plot_eigenvalues_explicitly, 52
- ksp_richardson_scale, 50
- ksp_right_pc, 50
- ksp_rtol, 51
- ksp_singmonitor, 52
- ksp_truemonitor, 52
- ksp_type, 49
- ksp_xmonitor, 51, 52, 87
- log, 105, 107, 114, 129
- log_all, 105, 107, 114, 129
- log_history, 110
- log_info, 42, 44, 105, 110
- log_mpe, 107, 114
- log_summary, 105–107, 114
- log_trace, 105, 121
- mat_ajj_oneindex, 42
- mat_coloring, 75
- mat_fd_coloring_err, 75
- mat_fd_coloring_umin, 75
- mat_order, 80
- no_signal_handler, 121
- nox, 86
- optionsleft, 119
- optionstable, 119
- pc_asm_type, 56
- pc_bgs_blocks, 56
- pc_bjacobi_blocks, 56
- pc_eisenstat_diagonal_scaling, 55
- pc_eisenstat_omega, 55
- pc_ilu_in_place, 54
- pc_ilu_levels, 54
- pc_ilu_nonzeros_along_diagonal, 54, 81
- pc_ilu_reuse_fill, 54
- pc_ilu_reuse_reordering, 54
- pc_ilu_use_drop_tolerance, 54
- pc_lu_in_place, 55
- pc_lu_nonzeros_along_diagonal, 56, 81
- pc_mg_cycles, 57
- pc_mg_method, 57
- pc_mg_smoothdown, 57
- pc_mg_smoothup, 57
- pc_sor_backward, 55
- pc_sor_its, 55
- pc_sor_local_backward, 55
- pc_sor_local_forward, 55
- pc_sor_local_symmetric, 55
- pc_sor_omega, 55
- pc_sor_symmetric, 55
- pc_type, 53
- snes_atol, 67
- snes_cancelmonitors, 67
- snes_fmin, 67
- snes_ksp_ew_conv, 68
- snes_line_search, 66
- snes_line_search_alpha, 66
- snes_line_search_maxstep, 66
- snes_line_search_steptol, 66
- snes_max_funcs, 67
- snes_max_it, 67
- snes_mf, 69
- snes_mf_err, 69
- snes_mf_operator, 69
- snes_mf_umin, 69
- snes_monitor, 67
- snes_rtol, 67
- snes_stol, 67
- snes_test_display, 68
- snes_trtol, 67
- snes_type, 64
- snes_xmonitor, 67, 87
- sub_ksp_type, 56
- sub_pc_type, 56
- trdump, 6
- ts_pseudo_increment_dt_from_initial_dt, 79
- ts_type, 77

- v, 6
- version, 6
- .petschistory, 110
- .petsrc, 118
- .petsviewrc, 131
- ADD_VALUES, 22, 24
- Additive, 57
- alias, 118
- AO, 25, 26
- AOApplicationToPetsc(), 26
- AOApplicationToPetscIS(), 26
- AOCreatDebug, 25
- AOCreatDebugIS, 26
- AODestroy(), 26
- AOPetscToApplication(), 26
- AOPetscToApplicationIS(), 26
- AOView, 26
- CHKERRA(), 121
- CHKERRQ(), 121
- CHKPTRA(), 121
- CHKPTRQ(), 121
- COLORING_ID, 75
- COLORING_LF, 75
- COLORING_SL, 75
- DA_NONPERIODIC, 27
- DA_STENCIL_BOX, 27
- DA_STENCIL_STAR, 27
- DA_XPERIODIC, 27
- DA_XYPERIODIC, 27
- DA_XYZPERIODIC, 28
- DA_XZPERIODIC, 28
- DA_YPERIODIC, 27
- DA_YZPERIODIC, 28
- DA_ZPERIODIC, 28
- DACreate1d(), 27
- DACreate2d, 27
- DACreate3d(), 27
- DAGetAO, 29
- DAGetCorners(), 28
- DAGetDistributedVector, 28
- DAGetDistributedVector(), 28
- DAGetGhostCorners(), 28
- DAGetGlobalIndices(), 28, 89
- DAGetLocalVector(), 28
- DAGetScatter(), 28
- DAGlobalToLocalBegin(), 28
- DAGlobalToLocalEnd(), 28
- DALocalToGlobal(), 28
- DALocalToLocalBegin(), 28
- DALocalToLocalEnd(), 28
- DFVecDrawTensorContoursX, 39
- DFVecDrawTensorSurfaceContoursVRML, 39
- DFVecView, 39
- DrawAxis*(), 52
- DrawAxisSetColors(), 85
- DrawAxisSetLabels(), 85
- DrawFlush(), 85
- DrawLG*(), 52
- DrawLGAddPoint(), 85
- DrawLGAddPoints(), 85
- DrawLGCreate(), 85
- DrawLGDestroy(), 85
- DrawLGDraw(), 85
- DrawLGGetAxis(), 85
- DrawLGReset(), 85
- DrawLGSetLimits(), 85
- DrawLine(), 85
- DrawOpenX(), 84
- DrawSetCoordinates(), 84
- DrawSetDoubleBuffer(), 85
- DrawSetViewPort(), 84
- DrawSP*(), 52
- DrawSyncFlush(), 85
- DrawText(), 85
- DrawTextGetSize(), 85
- DrawTextSetSize(), 85
- DrawTextVertical(), 85
- Full Multigrid, 57
- HAVE_FORTRAN_CAPS, 91
- HAVE_FORTRAN_UNDERSCORE, 91
- Hermitian matrix, 50
- inplace solvers, 55
- INSERT_VALUES, 21, 24
- ISBlock(), 24
- ISBlockGetBlockSize(), 24
- ISBlockGetIndices(), 24
- ISBlockGetSize(), 24
- ISColoringDestroy(), 75
- ISCreateBlock, 24
- ISCreateGeneral(), 23
- ISDestroy(), 24
- ISGetIndices(), 24, 89
- ISGetSize(), 24
- ISLocalToGlobalMapping, 26
- ISLocalToGlobalMappingApply(), 26
- ISLocalToGlobalMappingApplyIS(), 26
- ISLocalToGlobalMappingCreate(), 26
- ISLocalToGlobalMappingDestroy(), 26
- ISRestoreIndices(), 24
- ISStrideGetInfo(), 24
- Kaskade, 57
- KSP_CG_SYMMETRIC, 50
- KSPBCGS, 49
- KSPBuildResidual(), 53
- KSPBuildSolution(), 53
- KSPCG, 49
- KSPCGSetType(), 50
- KSPCGType, 50
- KSPCHEBYCHEV, 49
- KSPChebychevSetEigenvalues(), 49

KSPComputeEigenvalues(), 52
 KSPCR, 49
 KSPCreate(), 82
 KSPDefaultMonitor(), 52
 KSPDestroy(), 82
 KSPGetRhs(), 52
 KSPGetSolution(), 52
 KSPGMRES, 49
 KSPGMRESIROrthog, 50
 KSPGMRESSetOrthogonalization(), 50
 KSPGMRESSetRestart(), 49
 KSPGMRESUnmodifiedGramSchmidtOrthogonalization, 50
 KSPLGMonitor(), 87
 KSPLGMonitorCreate(), 52, 87
 KSPLGMonitorDestroy(), 52
 KSPPREONLY, 49
 KSPRICHARDSON, 49
 KSPRichardsonSetScale(), 49
 KSPSetComputeEigenvalues(), 52
 KSPSetConvergenceTest(), 51
 KSPSetInitialGuessNonzero(), 50
 KSPSetMonitor(), 51
 KSPSetPC(), 82
 KSPSetRhs(), 53
 KSPSetSolution(), 53
 KSPSetTolerances(), 51
 KSPSetType(), 49
 KSPSetUp(), 82
 KSPSingularValueMonitor(), 52
 KSPSolve(), 82
 KSPTCQMR, 49
 KSPTFQMR, 49
 KSPTrueMonitor(), 52
 MAT_COLUMNS_SORTED, 40
 MAT_FINAL_ASSEMBLY, 41
 MAT_FLUSH_ASSEMBLY, 41
 MAT_ROWS_SORTED, 40
 MatAssemblyBegin(), 10, 41
 MatAssemblyEnd(), 10, 41
 MatCholeskyFactor(), 81
 MatCholeskyFactorNumeric(), 81
 MatCholeskyFactorSymbolic(), 81
 MatConvert(), 46
 MatCreate(), 10, 40
 MatCreateMPIAIJ(), 43
 MatCreateMPIBAIJ(), 140
 MatCreateMPIRowbs(), 54
 MatCreateSeqAIJ(), 41
 MatCreateSeqBAIJ, 139
 MatCreateSeqBDiag, 141
 MatCreateSeqDense(), 44
 MatCreateShell(), 46, 48
 MatFDColoringCreate(), 75
 MatFDColoringSetFromOptions(), 75
 MatFDColoringSetParameters(), 75
 MatGetArray(), 89
 MatGetColoring(), 75
 MatGetOwnershipRange(), 41
 MatGetReordering(), 80
 MatGetRow(), 47
 MatLoad(), 120
 MatLUFactor(), 81
 MatLUFactorNumeric(), 81
 MatLUFactorSymbolic(), 81
 MatMult(), 45
 MatMultAdd(), 45
 MatMultTrans(), 45
 MatMultTransAdd(), 45
 MatReorderForNonzeroDiagonal, 81
 MatReorderingRegister(), 81
 MatRestoreRow(), 47
 MatSetOption(), 40
 MatSetValues(), 10, 40
 MATSHELL, 69
 MatShellGetContext(), 46
 MatShellSetOperation(), 46
 MatSolve(), 81
 MatSolveAdd(), 82
 MatSolveTrans(), 82
 MatSolveTransAdd(), 82
 MatView(), 45
 MatZeroEntries(), 46
 MatZeroRows(), 46
 MG_W_CYCLE, 57
 MGADDITIVE, 57
 MGDefaultResidual(), 58
 MGFULL, 57
 MGGetCoarseSolve(), 57
 MGGetSmoother(), 57
 MGKASKADE, 57
 MGMULTIPLICATIVE, 57
 MGSetCycles(), 57
 MGSetLevels(), 57
 MGSetNumberSmoothDown(), 57
 MGSetNumberSmoothUp(), 57
 MGSetR(), 58
 MGSetResidual(), 57
 MGSetRhs(), 58
 MGSetSmoother(), 57
 MGSetType(), 57
 MGSetX(), 58
 MPI_Finalize(), 7
 MPI_Init(), 6
 mpirun, 5
 Multiplicative, 57
 OptionsGetDouble(), 119
 OptionsGetDoubleArray(), 119
 OptionsGetInt(), 119
 OptionsGetIntArray(), 119

OptionsGetString(), 119
OptionsHasName(), 119
OptionsSetValue(), 118
ORDER_1WD, 55, 80
ORDER_NATURAL, 55, 80
ORDER_ND, 55, 80
ORDER_NEW, 81
ORDER_QMD, 55, 80
ORDER_RCM, 55, 80
PC_ASM_BASIC, 56
PC_ASM_INTERPOLATE, 56
PC_ASM_NONE, 56
PC_ASM_RESTRICT, 56
PCApply(), 83
PCApplyBAorAB(), 83
PCApplyBAorABTrans(), 83
PCApplyRichardson(), 83
PCApplyTrans(), 83
PCASM, 53
PCASMSetOverlap, 57
PCASMSetTotalSubdomains(), 56
PCASMSetType(), 56
PCBGs, 53
PCBGsSetTotalBlocks(), 56
PCBJACOBI, 53
PCBJacobiGetSubSLES(), 56
PCBJacobiSetTotalBlocks(), 56
PCCreate(), 82
PCDestroy(), 83
PCEISENSTAT, 55
PCEisenstatSetOmega(), 55
PCEisenstatUseDiagonalScaling(), 55
PCGetOperators(), 82
PCICC, 53
PCILU, 53
PCILUSetLevels(), 54
PCILUSetReuseFill(), 54
PCILUSetReuseReordering, 54
PCILUSetUseDropTolerance(), 54
PCILUSetUseInPlace(), 54
PCJACOBI, 53
PCLU, 53
PCLUSetUseInPlace(), 49, 55
PCNONE, 53
PCSetOperators(), 82
PCSetType(), 53, 82
PCSetVector(), 82
PCSHELL, 53, 69
PCShellSetApply(), 57
PCSide, 50
PCSOR, 53
PCSORSetIterations(), 55
PCSORSetOmega(), 55
PCSORSetSymmetric(), 55
PETSC_COMM_WORLD, 6
PETSC_COMPLEX, 125
PETSC_DEBUG, 125
PETSC_DECIDE, 21, 43, 44, 140
PETSC_DEFAULT, 51
PETSC_DIR, 5, 127
PETSC_FORTRAN_LIB, 92, 127
PETSC_FP_TRAP_OFF, 121
PETSC_FP_TRAP_ON, 121
PETSC_INCLUDE, 127
PETSC_LIB, 92, 127
PETSC_LOG, 105, 125
PETSC_NULL, 91
PETSC_NULL_CHARACTER, 91
PETSC_OPTIONS, 6, 118
PetscAbortErrorHandler(), 120
PetscAttachErrorHandler(), 120
PetscCObjectToFortranObject(), 90
PetscCompareDouble(), 122
PetscCompareInt(), 122
PetscCompareScalar(), 122
PetscDefaultSignalHandler(), 121
PetscError(), 120
PetscFinalize(), 7
PetscFortranObjectToCObject(), 90
PetscFPrintf(), 110
PetscGetTime(), 110
PetscInitialize(), 6
petscman, 5
PetscObjectGetComm(), 52, 137
PetscObts, 119
PetscPopErrorHandler(), 120
PetscPrintf(), 110
PetscPushErrorHandler(), 120
PetscPushSignalHandler(), 121
PetscSetCommWorld, 6
PetscSetFPTrap(), 121
PetscTraceBackErrorHandler(), 120
petscview, 107, 129
PLogAllBegin(), 139
PLogBegin(), 139
PLogDump(), 139
PLogEventBegin(), 108, 139
PLogEventEnd(), 108
PLogEventRegister(), 108
PLogFlops(), 108
PLogInfo(), 110
PLogInfoActivateClass(), 110
PLogInfoAllow(), 110
PLogInfoDeactivateClass(), 110
PLogObjectCreate(), 138
PLogObjectDestroy(), 138
PLogObjectParent(), 138
PLogObjectState(), 139
PLogPrintSummary(), 139
PLogStagePop(), 109

PLogStagePush(), 109
 PLogStageRegister(), 109
 PLogTraceBegin(), 121
 SAME_NONZERO_PATTERN, 48, 65
 SAME_PRECONDITIONER, 48
 Scalar, 10
 SCATTER_FORWARD, 24
 SCATTER_REVERSE, 25
 SETERRA(), 121
 SETERRQ(), 121
 SLESCreate(), 10, 48
 SLESDestroy(), 10, 49
 SLESGetKSP(), 49
 SLESGetPC(), 49
 SLESSetFromOptions(), 10, 49
 SLESSetOperators(), 10, 48
 SLESSetUp(), 49, 56
 SLESSolve(), 10, 49
 SNESDefaultMatrixFreeMatCreate(), 69
 SNESDefaultMonitor(), 67
 SNESetFromOptions(), 64
 SNESGetFunction, 67
 SNESGetSolution(), 67
 SNESGetTolerances(), 67
 SNESNoLineSearch(), 66
 SNESSetConvergenceTest(), 67
 SNESSetFunction(), 64
 SNESSetGradient(), 65
 SNESSetHessian(), 65
 SNESSetJacobian(), 64, 77, 78
 SNESSetLineSearch(), 66
 SNESSetMatrixFreeParameters(), 69
 SNESSetMinimizationFunction(), 65
 SNESSetMonitor(), 67
 SNESSetTolerances(), 67
 SNESsetType(), 64
 SNESsolve, 64
 SOR_BACKWARD_SWEEP, 55
 SOR_FORWARD_SWEEP, 55
 SOR_LOCAL_BACKWARD_SWEEP, 55
 SOR_LOCAL_FORWARD_SWEEP, 55
 SOR_LOCAL_SYMMETRIC_SWEEP, 55
 SOR_SYMMETRIC_SWEEP, 55
 TS, 76
 TS_BEULER, 77
 TS_EULER, 77
 TS_PSEUDO, 77
 TSCreate(), 77
 TSDestroy(), 77
 TSGetTimeStep(), 77
 TSProblemType, 77
 TSPseudoIncrementDtFromInitialDt(), 79
 TSPseudoSetTimeStepIncrement(), 79
 TSSetDuration(), 77
 TSSetInitialTimeStep, 77
 TSSetRHSFunction, 78
 TSSetRHSJacobian, 78
 TSSetRHSMatrix(), 77
 TSSetSolution(), 77
 TSSetTimeStep(), 77
 TSSetType(), 77
 TSSetUp(), 77
 TSView(), 77
 Vec, 21
 VecAssemblyBegin(), 21
 VecAssemblyEnd(), 21
 VecCreate(), 10, 21
 VecCreateMPI(), 21, 26
 VecCreateSeq(), 21
 VecDestroy(), 22
 VecDestroyVecs(), 22
 VecDestroyVectors(), 91
 VecDuplicate(), 10, 22
 VecDuplicateVecs(), 22, 91
 VecGetArray(), 23, 89, 115
 VecGetLocalSize(), 23
 VecGetOwnershipRange(), 22
 VecGetSize(), 23
 VecLoad(), 120
 VecScatterBegin(), 24
 VecScatterCreate(), 24
 VecScatterDestroy(), 24
 VecScatterEnd(), 24
 VecSet(), 10, 21
 VecSetValues(), 10, 21, 22, 25
 VecView(), 22
 Viewer, 119
 VIEWER_DRAWX_SELF, 84, 119
 VIEWER_DRAWX_WORLD, 22, 45, 84, 119
 VIEWER_FORMAT_ASCII_DEFAULT, 120
 VIEWER_FORMAT_ASCII_IMPL, 120
 VIEWER_FORMAT_ASCII_MATLAB, 120
 VIEWER_STDOUT_SELF, 22, 119
 VIEWER_STDOUT_WORLD, 22, 119
 ViewerDestroy(), 120
 ViewerDrawGetDraw(), 84
 ViewerDrawOpenX, 45
 ViewerDrawOpenX(), 84
 ViewerFileOpenASCII(), 119
 ViewerFileOpenBinary, 119
 ViewerMatlabOpen(), 120
 ViewerPopFormat, 120
 ViewerPushFormat(), 120
 ViewerSetFormat(), 120