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

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL-93/8

Simplified Linear Equation Solvers Users Manual

by

William Gropp
Mathematics and Computer Science Division

Barry Smith
Department of Mathematics
University of California, Los Angeles

February 1993

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy.

Contents

Simplified Linear Equation Solvers
Users Manual

by

William Gropp and Barry Smith

Abstract

The solution of large sparse systems of linear equations is at the heart of many algorithms in scientific computing. The SLES package is a set of easy-to-use yet powerful and extensible routines for solving large sparse linear systems. The design of the package allows new techniques to be used in existing applications without any source code changes in the applications.

Chapter 1

Introduction

The SLES (Simplified Linear Equation Solvers) package provides a powerful, yet easy-to-use interface to methods for solving systems of linear equations of the form

$$Ax = b.$$

This package provides access to both direct and iterative solvers and allows you to switch easily from one method to another. The matrix representation is "data-structure neutral;" this means that virtually any sparse (or dense) matrix representation may be used directly. All that is necessary is to define a few operations on the matrix data structure. In addition, it is relatively easy to extend the set of known methods, including the iterative accelerator, direct factorization, and preconditioning. Any extension may be made without modifying a single line of the SLES package.

SLES provides a large and growing set of matrix representations, including dense, AIJ, and dynamic sparse row-oriented format. These features make this package ideal both for new projects and for existing applications.

There are four steps in using the routines in this package. First, you create a solver context with `SVCreate`. This context holds information specific to each technique. Second, you may change any options, such as the kind of sparse matrix ordering or the iterative accelerator, and then use `SVSetUp` to set up the solvers. Third, you solve the system with `SVSolve`. Fourth, you remove the solver context with `SVDestroy`. Multiple systems may be solved with multiple calls to `SVSolve`.

Among the design requirements for the routines provided by the SLES package is the requirement that the calling sequences and routines be identical for all methods. This means that to change the method used to solve a linear system, you need only change the `method` argument to `SVCreate`, and nothing else.

This manual mentions all of the routines in the SLES package; however, usage instructions are provided only for the more common routines. More detailed information about the routines mentioned in this manual may be found in the man pages (using `toolman`). SLES is part of a larger package, PETSc (Portable, Extensible Tools for Scientific computing). `toolman` is one of the tools provided by PETSc for accessing the detailed documentation on the routines. PETSc also provides an number of routines that may be of interest to users of SLES, including routines to report on floating point errors, memory space tracing, and debugging. See the man pages for more information.

1.1 Why simplified

Despite the apparent power and flexibility of the SLES package, this package actually provides a simplified version of more flexible and powerful routines that are part of the PETSc package. The simplifications include a restriction to uniprocessors and vectors that are contiguous in memory. Other simplifications include less access to all of the options of the iterative solvers and a matrix-based representation of the linear system.

The underlying tools, particularly the iterative routines, have none of these restrictions. They may be easily used in a parallel environment and with vectors with arbitrary storage formats (including vectors distributed across a parallel machine and vectors that are stored in sophisticated data structures, such as oct-trees, or stored out-of-core). In addition, the design of these routines makes it easy to replace each module with one optimized to a specific problem or application.

The SLES package is designed to sit on top of these more powerful but more complicated routines and to make it easier for you to solve linear systems that do not have special needs. You do not need to know about or understand the lower level routines. SLES provides a consistant, simple, and easy to use interface to a more powerful set of routines. If you find that SLES does not give you the functionallity that you need, you should then (and only then) investigate these other parts of PETSc.

1.2 Makefiles

PETSc has a system of makefiles that has been designed to enable the same makefile to build libraries or programs on a wide variety of architectures. For some examples, see the makefiles in any of the example directories. Basically, these use a few variables to control exactly what options the makefile uses.

The variables that must be defined on the `make` command line are listed below:

ARCH Architecture. Common values include `sun4`, `rs6000`, and `intelnx`.

BOPT Level of optimization. Use `BOPT=g` for debugging, `BOPT=Opg` for profiling, and `BOPT=O` for production.

Note that at least `BOPT` and `ARCH` must be set on the make command line or defined with shell environment variables; without these values, the makefiles in PETSc will not work.

In addition, a variety of variables are defined for use within the user's makefile. The most important of these are as follows:

BASEOPT Flags for the C compiler. This includes options like `-g`.

BASEOPTF Flags for the Fortran compiler.

SLIB System libraries that PETSc need. Often, this variable is empty, but it may include special libraries that are needed for the implementation of PETSc for particular architectures (the SGI workstations are an example). The math library (`-lm`) is not included by default, though many of the routines will require that library.

These are values that are provided by the makefile system for your use; they should not be changed.

1.3 Linking

To build programs with PETSc, you need to link with a number of libraries. To simplify the use of PETSc for both program development and production computing, PETSc has separate libraries for debugging, profiling, and production. These libraries are in the following directories:

debugging '`tools.core/libs/libsg`'

profiling 'tools.core/libs/libsOpg'

production 'tools.core/libs/libsO'

So that the libraries for many different architectures can reside on the same filesystem, the name of the architecture (such as 'sun4' or 'rs6000') defines an additional directory level. For example, the debugging libraries for the Sun 4 are found in the directory 'tools.core/libs/libsg/sun4'.

There are two libraries that you may need to link with. These are 'tools.a' and 'system.a'. For example, a partial makefile is shown below that builds the program 'example' for Sun 4's:

```
ITOOLSDIR = /usr/local/tools.core
LIBDIR    = $(ITOOLSDIR)/libs/libsO/sun4
example: example.o
        $(CLINKER) -o example -O example.o \
               $(LIBDIR)/tools.a $(LIBDIR)/system.a -lm
include $(ITOOLSDIR)/bmake/$(ARCH).O
include $(ITOOLSDIR)/bmake/$(ARCH)
```

This builds a production version of 'example' on a Sun 4. The `include` lines include definitions for `CLINKER` (the linker for C programs), as well as the rule to compile a C program that uses the PETSc macros (making sure the appropriate flags are defined). The makefiles that come with the standard PETSc distribution use the macro `$(ARCH)` to hold the place of one of the many architectures, including Sun 4, to which PETSc has been ported.

SLES includes some graphical aids for displaying, for example, the progress of the solution algorithm. Using these requires a few additional libraries. In the example above, we simply add the 'Xtools.a' library (part of PETSc) and the X11 Window System library '-lX11':

```
ITOOLSDIR = /usr/local/tools.core
LIBDIR    = $(ITOOLSDIR)/libs/libsO/sun4
example: example.o
        $(CLINKER) -o example -O example.o \
             $(LIBDIR)/Xtools.a $(LIBDIR)/tools.a \
             $(LIBDIR)/system.a -lX11 -lm
include $(ITOOLSDIR)/bmake/$(ARCH).O
include $(ITOOLSDIR)/bmake/$(ARCH)
```

1.4 Installing the SLES package

The SLES package is available by anonymous ftp from 'info.mcs.anl.gov' in the directory 'pub/pdetools'. The file 'sles.tar.Z' is a compressed tar file containing all of the code and documentation. The file 'solvers.ps.Z' is a compressed postscript file containing this document. To install the package, transfer the tar file, uncompress it, and do

```
tar xf tools_solvers.tar
```

This will create a directory 'tools.core' as a subdirectory of the current directory. Then do

```
cd tools.core
bin/install >&install.log &
```

(assuming the C-shell). This will create all versions of the package (debugging, profiling, and production). Should you wish to produce only a single version such as the debugging version, (for example, to limit the amount of disk space used by the package), do

```
bin/install -libs g >&install.log &
```

1.5 Restrictions

This package is intended primarily for linear systems that arise from the discretization of partial differential equations and that are not extremely poorly conditioned. Methods for more poorly conditioned problems, or ones that contain singular principal submatrices, will be included at a later date.

1.6 Further information

Every routine mentioned here has a Unix man page. For brevity, these are not attached to this document. They are found in 'tools.core/man'.

This document is available in "latexinfo" form for users of GNU Emacs. A sample 'localdir' file is in 'tools.core/docs/localdir'. Emacs must be informed about these "info" files with a command like

```
(setq Info-directory-list (list "/usr/local/tools.core/docs"
Info-directory))
```

where '/usr/local/tools.core' is the home of the PETSc code.

This package is continually growing through the addition of new routines. Suggestions (and bug reports) should be e-mailed to 'gropp@mcs.anl.gov'.

Chapter 2

Solver Methods

A wide variety of techniques, including preconditioned iterative methods and direct sparse elimination, are available for the solution of systems of linear equations. The method to be used is selected when the routine `SVCreate` is called. This routine returns a pointer to a structure, called a *context*, that holds the data needed to specify the type of method and any parameters that it may need.

2.1 Solver context

The solver context is the key to using the solver library. This context, created with the `SVCreate` routine, contains all of the data needed to describe the solver method. You can think of the solver context as a way to encapsulate a large number of subroutine arguments into a single structure. The advantages to using the context instead of individual parameters are many; they include

- much shorter argument lists,
- easy-to-add features without changing calling sequences, and
- easy-to-use, nested (as opposed to recursive) algorithms.

The solver context is of type `(SVctx *)`. The format of `SVCreate` is

```
SVctx *SVCreate( mat, name )
```

where `mat` is a sparse matrix context and `name` is one of

SVLU Direct (possibly sparse) factorization

SVNOPRE No preconditioning. This method and all that follow it are preconditioners for an iterative method.

SVJacobi Jacobi preconditioning

SVSSOR SSOR preconditioning

SVILU Incomplete factorization

SVICC Incomplete Cholesky factorization

SVICCJP Incomplete Cholesky factorization (Jones and Plassmann) [3]

SVBDD Block diagonal

SVOSM Overlapping Schwarz (additive and multiplicative)

Following the `SVCreate` call, you may set various options. These are described in more detail in Section 5.3 and have the form `SVSetxxx()`. An important feature of the optional arguments is that any that are inappropriate for a particular solver are simply ignored. This means that you do not need to change the source code when choosing a different solver method.

Once any options have been chosen, you use the routine `SVSetUp` to initialize all of the data for the chosen method. The format of this routine is

```
SVSetUp( ctx );
```

At this point, you may also set options for the iterative method. Then the system may be solved with the routine `SVSolve`. The format of this routine is

```
SVSolve( ctx, b, x )
double *b, *x;
```

where `b` is the right-hand side and `x` is the computed solution.

Multiple right-hand sides `b` may be solved by calling `SVSolve` multiple times.

When you are done solving this linear system, use the routine `SVDestroy` to recover the solver context (not doing so will create a storage leak). The format of this routine is

```
SVDestroy( ctx )
```

This frees only the space associated with the solver. It does not free the matrix that was input to `SVCreate`.

2.2 Getting the methods from the command line

SLES provides three routines that may be used to extract the desired method from the command line. The routine `SVGetMethod` gets the solver's method. The format is

```
SVGetMethod( Argc, argv, name, svmethod )
int      *Argc;
char     **argv, *name;
SVMETHOD *svmethod;
```

where the first two arguments are the arguments to the main routine, `name` is the name to be used on the command line (`-svmethod` is the default if `name` is `NULL`), and the chosen method is returned in `svmethod`. If no method is specified on the command line, `svmethod` is unchanged. This routine purges the solver method arguments from the argument list.

The routine `ITGetMethod` does the same thing, but for the iterative accelerators. The default value of `name` is `-itmethod`. The format of this routine is

```
ITGetMethod( Argc, argv, name, itmethod )
int      *Argc;
char     **argv, *name;
ITMETHOD *itmethod;
```

The routine `SpGetOrdering` does the same for the matrix orderings. The default value of `name` is `-ordering`. The format of this routine is

```
SpGetOrdering( Argc, argv, name, ordering )
int         *Argc;
char        **argv, *name;
SPORDERTYPE *ordering;
```

These are only available for C programs as Fortran does not provide a portable way to access the command line arguments.

Chapter 3

Creating the Matrix

The SLES package uses matrices in the SMEIT sparse format. These routines will be described in complete detail in another manual. Here we give you a subset sufficient for using SLES.

Just as the SLES package uses a context variable to hide various details from you, the SMEIT package uses its own context to provide a general, data-structure-independent interface to a collection of sparse matrix routines.

The routine `SpCreate` creates the most general supported type of sparse matrix. Most users of SLES should use this type. The format of this routine is

```
SpMat *SpCreate( rows, cols, nmax )
int rows, cols, nmax;
```

The matrix is of size `rows` $\times$ `cols`. The value `nmax` is the number of nonzeros in each row; this may be used to speed up the process of adding elements to the matrix. This is *only* an optimization; it is *not* a limit on the number of nonzeros in each row of the matrix. A good value of `nmax` is often zero, allowing the matrix to grow dynamically.

The more sophisticated user may choose other storage formats with the `Sp<format>Create` routines, where `<format>` is one of the known formats. Formats currently available include dense (`format` = `Dn`) and AIJ (`<format>` = `AIJ`). Additional formats may be defined by the expert user.

If the matrix is already available, a routine of the form `Sp<format>CreateFromData` should be used. For example, the routine `SpDnCreateFromData` creates a matrix context from a dense matrix; the routine `SpAIJCreateFromData` does the same for a matrix in AIJ format.

To add an element to a sparse matrix created with any type of `SpCreate`, use the routine `SpAddValue`. The format of this routine is

```
SpAddValue( mat, value, row, col )
SpMat *mat;
double value;
int    row, col;
```

Parameters `row` and `col` are zero-origin indexed; that is, they take on values between zero and `rows-1`, (respectively `cols-1`). The parameter `value` is added to the matrix element already present. In other words, after the execution of

```
SpAddValue( mat, 1.0, 0, 0 );
SpAddValue( mat, 1.0, 0, 0 );
```

the value of `mat[0,0]` is two, not one. The routine `SpSetValue` may be used to set a single value. To add to every element in a row, use the routine `SpAddInRow`. The format of this routine is

```
SpAddInRow( mat, row, n, v, cols )
SpMat *mat;
int   row, n, *cols;
double *v;
```

This adds `v[i]` to row `row` and column `cols[i]`, for `n` elements.

When the matrix is no longer needed, the routine `SpDestroy` will recover all of the space that it is using. This routine will work regardless of the underlying data structure used to store the sparse matrix.

Chapter 4

Direct Methods

Sparse direct methods are appropriate for relatively small systems of equations and for cases where iterative methods are not robust enough. The solver name `SVLU` selects this method. This forms an LU factorization of the matrix (perhaps with a permuted ordering of rows and columns).

Three options are supported for this direct factorization. Two control the ordering of the matrix; the third is used to (possibly) speed the factorization. These options are ignored when an iterative solver is used.

4.1 Sparse matrix orderings

The routine `SVSetLUOrdering` choses an ordering strategy that is used to reduce fill and to speed the solution process. The currently recognized orderings are

ORDER_ND Nested dissection. This is often the most efficient ordering for matrices that come from discretizations of PDEs.

ORDER_QMD Quotient Minimum Degree. This is often the best ordering in terms of space, but it is often the most expensive in time.

ORDER_RCM Reverse Cuthill-McGee. This is often a good ordering for skyline-like methods.

ORDER_1WD One-way dissection.

The format of this routine is

`SVSetLUOrdering( svctx, ordering )`

where `svctx` is a solver context from `SVCreate` and `ordering` is one of the orderings (i.e., `ORDER_ND`). The orderings provided at this time come from the *netlib* versions from SPARSPAK (see [2] for more information on these routines). Since these ordering routines are intended for symmetric matrices, SLES applies them to the matrix structure of the lower triangular part of the matrix. This ensures that the methods produce valid orderings for most nonsymmetric matrices and produce the same ordering as they would on a symmetric A.

In addition to these, additional ordering methods may be defined with the routine `SpOrderRegister`.

4.2 Pivoting

Pivoting is used to avoid or reduce numerical problems that can arise in solving a linear system. The routine `SVSetLUPivoting` allows you to select a pivoting method. Currently, the only pivoting strategies supported do *a priori* pivoting, moving small elements off the diagonal of the unfactored matrix. These are `PIVOT_PRE_SYM` (for symmetric pivoting) and `PIVOT_PRE_NONSYM` (for nonsymmetric pivoting). The format of this routine is

`SVSetLUPivoting( ctx, pivot_type )`

By default, `PIVOT_NONE` (for no pivoting) is used.

SLES currently does not support partial pivoting for numerical stability, except for the dense matrix type (where it is the default choice).

4.3 Blocking

The routine `SVSetLUThreshold` sets a minimum block size to use in searching for blocks of rows in the matrix. If the problem has a natural blocksize, such as a multicomponent PDE with `m` unknowns per mesh point, set the threshold to `m`. The format of this routine is

`SVSetLUThreshold( ctx, bsize )`

where `bsize` is the block size. A good default threshold is four.

Blocking affects only the efficiency of memory access; it does not change the algorithms used (that is, it does not use a blocked LU factorization; rather, it arranges to compute the same LU factorization but with aggregate operations, similar to BLAS Levels 2 and 3).

4.4 Example

This section presents a code fragment that illustrates the use of these routines. Note that neither of the `SVSetLUxxx` routines is required; they are provided to allow you to customize the solver.

This code uses LU factorization with quotient minimum degree ordering to solve the problem `mat*x=b` for `x`. The routine is told that a block size of 8 is a good choice for this problem.

```
svctx = SVCreate( mat, SVLU );
SVSetLUOrdering( svctx, ORDER_QMD );
SVSetLUThreshold( svctx, 8 );
SVSetUp( svctx );
SVSolve( svctx, b, x );
SVDestroy( svctx );
```

Chapter 5

Iterative Methods

The SLES package provides a wide variety of iterative methods. These are organized by the choice of preconditioner rather than the choice of iterative method. This approach was taken because the choice of preconditioner more strongly determines the performance and behavior of the method than the choice of iterative method.

5.1 Preconditioners

The preconditioners supported include

SVILU	Incomplete factorization with fill
SVICC	Incomplete Cholesky
SVICCJP	Jones and Plassmann's incomplete Cholesky
SVSSOR	SSOR
SVBDD	Block diagonal decomposition
SVOSM	Overlapping Schwarz method
SVJacobi	Jacobi
SVNOPRE	No preconditioning (identity matrix)

The matrix used for constructing the preconditioner is the input matrix to `SVCreate`; you may change this matrix with `SVSetPrecondMat`. This routine is particularly useful when a good preconditioner can be defined by a matrix with fewer nonzero elements or a special structure. The routine must be called before `SVSetUp`.

5.2 Initial guess

The initial guess is taken as zero unless the routine `SVSetUseInitialGuess` is used. The initial guess is placed in the "`x`" vector; this is the vector that holds the solution in `SVSolve`. If the initial guess is not selected, it is not necessary to preset the `x`-vector to zero. The format of this routine is

```
SVSetUseInitialGuess( ctx, flag )
```

where `flag` is one if the value in `x` is to be used as the initial guess, and zero if the initial guess is the zero vector. Here, `x` is the solution argument to `SVSolve( ctx, b, x )`.

5.3 Method options

Several of the preconditioners have a number of options that may be used to improve the method's performance. These options are ignored when they are not appropriate for the solver being used.

5.3.1 ICC

The incomplete Cholesky methods have no parameters, but there are two diagnostics. These are `SVGetICCAlpha` and `SVGetICCFailures`. These routines may be called after `SVSolve` to determine how well the ICC preconditioner worked with the matrix.

5.3.2 ILU

ILU (Incomplete LU factorization) is a powerful and general method. The simplest ILU involves computing the LU factor of the matrix, discarding any elements that would introduce fill (nonzero values where A has a zero element). Better performance can often be obtained by allowing some of the elements in L and U to fill in. There are a number of ways to select which elements are allowed. One is a level of fill, which looks at the ancestry of an element. If the fill would be caused by an element that was in the original matrix, it is fill of level 1. Fill caused by level 1 fill is called fill of level 2, and so on. The amount of such fill may be set with the routine `SVSetILUFill`. The format of this routine is

```
SVSetILUFill( svctx, fill )
```

where `fill` is the level of fill. The default level of fill is zero (no fill).

Another way of choosing the amount of fill is to set a numeric tolerance. Values smaller than this are discarded. This tolerance is set with the routine `SVSetILUDropTol`. The format of this routine is

```
SVSetILUDropTol( svctx, rtol )
double rtol;
```

where `rtol` is a relative tolerance; values smaller than this, relative to the norm of the row, are discarded. If `rtol` is greater than zero, `fill` (set with `SVSetILUFill`) specifies the maximum number of nonzeros allowed in each row of the factored matrix.

One problem with ILU is that matrices that are nonsingular may have singular factors or poorly conditioned factors. The routine `SVSetILUPivoting` gives you some control over the type of pivoting that is applied to the matrix. The pivot choices are the same as for `SVLU` (see Section 4.2).

5.3.3 SSOR

The SSOR method has one parameter, the acceleration value omega. This value should be between zero and two, and is set with the routine `SVSetSSOROmega`. The default value is 1.0 (giving the Gauss-Seidel method).

5.3.4 BDD

The block diagonal method uses (possibly approximate) solutions to a block-diagonal restriction of the original matrix to precondition the problem. This preconditioner is most popular on parallel computers, where its application is perfectly parallel. However, this method is often less effective than others in this package, including the highly parallel overlapping Schwarz methods (`SVOSM`).

There are many degrees of freedom in defining a block-diagonal preconditioner. These include the precise decomposition and the methods used on each block. Naturally, we use the solver package to solve the systems for each block. Default decompositions and methods are provided, though these are not optimal choices. It is legal to use `SVBDD` as the solver method on each domain.

5.3.4.1 Solver method

To set the solver method to be used on each domain, use `SVSetBDDDefaultMethod`. The format of this routine is

```
SVSetBDDDefaultMethod( svctx, svmethod )
```

where `svmethod` is any valid solver method (such as `SVILU`). Note, however, that many iterative accelerators require the same preconditioner to be applied at each step. If this is the case (for example, using `ITGMRES`), use the routine `SVSetIts` to set some small limit on the number of iterations to take. Alternatively, set the relative convergence tolerance to be roughly machine epsilon. Take note that using inner iterations is not for the faint-hearted.

5.3.4.2 Decomposition

The routine `SVSetBDDDomainsNumber` sets the number of blocks in the block-diagonal decomposition. The format of this routine is

```
SVSetBDDDomainsNumber( svctx, n )
```

where `n` is the number of blocks. The blocks are defined as consecutive rows in the matrix. For example, if the matrix has 1000 rows and `SVSetBDDDomainsNumber(svctx,10)` is used, then there are 10 blocks, consisting of the submatrices of the original matrix A (in Fortran-90 array notation) `A(1:100,1;100)`, `A(101:200,101:200)`, ..., `A(901:1000,901:1000)`. A default choice is made by the package for the number of domains; this is currently the square root of the size of the matrix. This choice generates line-blocked decomposition for square meshes in two dimensions.

You may define more general decompositions by using the routine `SVSetBDDDomainsNumber`, followed by `SVSetBDDDecomp` for each of the blocks. `SVSetBDDDecomp` is given the rows (numbered from zero) that belong to each block (or domain). The format of this routine is

```
SVSetBDDDecomp( svctx, i, idx, nv )
```

where `i` is the number of the domain (numbered from zero), `idx` is an array of rows that belong to that domain, and `nv` is the number of rows in this domain.

5.3.4.3 Special decompositions

For discrete operators arising from two-dimensional Cartesian meshes, where the matrix is ordered according to the natural ordering, the routine `SVSetBDDRegularDomains2d` may be used to specify the decomposition. The format of this routine is

`SVSetBDDRegularDomains2d( svctx, nx, ny, nc )`
where the mesh is `nx` by `ny`, with `nc` components at each mesh point. The components are assumed to be numbered first (so a two-component problem has the first two matrix rows at the first mesh point). This routine uses `SVSetBDDDecomp` to define the domains.

5.3.5 OSM

The overlapping Schwarz method uses (possibly approximate) solutions to a collection of smaller problems, usually generated by decomposing a physical domain into pieces, then defining overlaps in terms of neighboring points. A critical component for efficient overlapping Schwarz methods is a small global or coarse-grid problem that is defined on the smaller domains. However, the method can be applied without this global problem, at some (potentially large) cost in efficiency.

Many degrees of freedom exist in defining an overlapping Schwarz preconditioner. These include the precise decomposition, the amount of overlap, and the methods used on each domain. Naturally, we use the solver package to solve the systems for each domain. Default decompositions and methods are provided, though these are not optimal choices. It is legal to use `SVOSM` as the solver method on each domain.

5.3.5.1 Solver method

To set the solver method to be used on each domain, use `SVSetOSMDefaultMethod`. The format of this routine is
`SVSetOSMDefaultMethod( svctx, svmethod )`
where `svmethod` is any valid solver method (such as `SVILU`).

5.3.5.2 Decomposition

The routine `SVSetOSMDomainsNumber` sets the number of blocks in the block-diagonal decomposition. The format of this routine is
`SVSetOSMDomainsNumber( svctx, n )`
where `n` is the number of blocks. The blocks are defined as consecutive rows in the matrix. For example, if the matrix has 1000 rows and `SVSetOSMDomainsNumber(svctx,100)` is used and the overlap is zero, then there are 10 blocks, consisting of the submatrices of the original matrix A (in Fortran-90 array notation) `A(1:100,1;100)`, `A(101:200,101:200)`, ..., `A(901:1000,901:1000)`. A default choice is made by the package for the

number of domains; this is currently the square root of the size of the matrix. This choice generates line-blocked decomposition for square meshes in two dimensions.

5.3.5.3 Overlap

The amount of overlap is set with `SVSetOSMDefaultOverlap`. The format of this routine is

```
SVSetOSMDefaultOverlap( svctx, j )
```

where `j` is the amount of overlap. Here, overlap is defined algebraically. An overlap of zero is identical to the block-diagonal decomposition. An overlap of one or greater augments each diagonal block with additional rows (and corresponding columns) according to the following recursive rule. For each block in the partition with overlap $k - 1$, list the indices of all columns outside of the block that contain a nonzero entry in any row of the block. Add the rows corresponding to indices in this list to the current block to obtain the block of overlap k.

5.3.5.4 General decompositions

A more general decomposition may be defined by using the routine `SVSetOSMDomainsNumber`, followed by `SVSetOSMDecomp` for each of the blocks. This routine is given the rows (numbered from zero) that belong to each block (or domain). The format of this routine is

```
SVSetOSMDecomp( svctx, i, idx, nv )
```

where `i` is the number of the domain (numbered from zero), `idx` is an array of rows that belong to that domain, and `nv` is the number of rows in this domain. Among other uses, you can use this decomposition to select disjoint sets of rows with the same "color" in multicolored preconditioners.

5.3.5.5 Special decompositions

For two-dimensional Cartesian meshes, where the matrix is ordered according to the natural ordering, you may use the routine `SVSetOSMRegularDomains2d` to specify the decomposition. The format of this routine is

```
SVSetOSMRegularDomains2d( svctx, nx, ny, nc )
```

where the mesh is `nx` by `ny`, with `nc` components at each mesh point. The components are assumed to be numbered first (so a two-component problem has the first two matrix rows at the first mesh point). This routine uses `SVSetOSMDecomp` to define the domains.

5.3.5.6 Additive and multiplicative OSM

There are two principal types of Schwarz methods: additive (like Jacobi) and multiplicative (like Gauss-Seidel). For uniprocessors, the multiplicative method is usually the best choice. However, for parallel processors, the additive technique may be more efficient (or may not, depending on many factors). Multicoloring can be used to create intermediate methods. The routine `SVSetOSMSDefaultMult` sets the choice of method. The format is

```
SVSetOSMDefaultMult( svctx, q )
```

where `q` is `1` for multiplicative Schwarz and `0` for additive Schwarz. By default, multiplicative Schwarz is used.

5.3.5.7 Defining the global problem

The global, or coarse, problem is key to getting the best performance from domain decomposition preconditioners. While a discussion of the details of this problem is beyond the scope of this manual (see, for example [1] for a discussion of the importance of the global problem), the basic idea is to restrict the number of unknowns to a smaller space, solve an appropriate system in that space, and interpolate that solution back to the full problem. This is reminiscent of multigrid, and there are a number of practical and theoretical similarities.

To specify the global problem, we need to specify the restriction and interpolation operations, the matrix for the global problem, and the method used to solve the global problem. In addition, the mapping from a domain to the global problem (that is, the indices of the global vector `c[i]` that "border" a domain) need to be specified. Each of these can be set once `SVCreate` is called.

The routine `SVSetOSMGlobalInterpolation` sets the interpolation routine. This routine must interpolate from the entire global problem to the entire original problem.

The routine `SVSetOSMGlobalRestriction` sets the restriction routine. This routine must restrict from the entire full problem to the global problem.

The routine `SVSetOSMGlobalMatrix` specifies the global problem to solve by giving the matrix in SMEIT sparse matrix form.

The routine `SVSetOSMGlobalSolver` specifies the solver *context* to use in solving the global problem.

The routine `SVSetOSMGlobalDecomp` specifies the mapping to global points from the indicated subdomain. The routines defined by `SVSetOSMGlobalRestriction` and `SVSetOSMGlobalProjection` may use this information.

More information on these routines may be found in the man pages or the reference manual.

5.4 Accelerators

The available accelerators comprise

ITCG	Conjugate gradient
ITCGS	Conjugate gradient squared
ITBCGS	BiCG-Stab
ITTFQMR	Freund's transpose-free QMR
ITGMRES	Generalized minimum residual
ITTCQMR	Chan's transpose-free QMR
ITCHEBYCHEV	Chebychev
ITRICHARDSON	Richardson

The choice of accelerator is made with `SVSetAccelerator`. The format is

```
SVSetAccelerator( svctx, itmethod )
```

where `itmethod` is one of the iterative accelerators (e.g., `ITTFQMR`). This must be called *before* `SVSetUp`. Accelerator options must be set *after* `SVSetUp` is called but before the call to `SVSolve`.

5.4.1 Convergence tests and iteration control

By default, all accelerators have a simple convergence test (based on a relative reduction in the norm of the residual), an upper limit on the number of iterations, and no runtime display of the progress of the iteration. Each of these defaults may be individually overridden.

The routine `SVSetConvergenceTest` allows you to provide a different routine to test for convergence.

The routine `SVSetMonitor` allows you to provide a routine that is called once per iteration. The routine `ITXMonitor` may be of interest; this monitor displays a graph of the progress of the iteration. To use this routine, use this code fragment before calling `SVSetup`:

```
SVSetMonitor(svctx,ITXMonitor,(void*)0);
ITXMonitor(0,0,-2,0.0);
```

(the call to `ITXMonitor` is needed to initialize `ITXMonitor`; it is unnecessary if only one system of equations is to be solved). You will also need to add the libraries '`tools.core/libs/libs$(BOPT)/$(ARCH)/Xtools.a -lX11`', where `BOPT` is one of `g` (debugging), `O` (production), or `Opg` (profiling).

The routine `SVSetIts` allows you to set the maximum number of iterations that an iterative method will use in attempting to solve the problem. The format of this routine is

```
SVSetIts( svctx, maxits )
```

where `maxits` is the maximum number of iterations allowed.

The routine `SVSetRelativeTol` sets the convergence tolerance if the default convergence criteria is used. The format of this routine is

```
SVSetRelativeTol( svctx, tol )
double tol;
```

where `tol` is the relative reduction in the residual needed before the convergence criterion is satisfied.

The routine `SVSetAbsoluteTol` sets the absolute convergence tolerance if the default convergence criteria is used. The format of this routine is

```
SVSetAbsoluteTol( svctx, tol )
double tol;
```

where `tol` is the absolute size of the residual needed before the convergence criterion is satisfied.

When both `SVSetRelativeTol` and `SVSetAbsoluteTol` are used, convergence is signaled when either criterion is satisfied.

Additional options are supported by the iterative package for use by the expert user. These include whether the preconditioner is applied on the right, on the left, or symmetrically and whether a residual history should be kept. In addition, there are a few options that are specific to particular accelerators. Changing these requires knowledge of the iterative package.

5.4.2 GMRES

The GMRES method has one parameter. This is the number of iterations between restarts (strictly speaking, the GMRES here is restarted GMRES(k)). This parameter may be set with the routine `SVSetGMRESRestart`. The format of this routine is

```
SVSetGMRESRestart( svctx, its )
```

where `its` is the number of iterations (or directions) before restarting GMRES.

Chapter 6

Monitoring Performance

The SLES package provides a convenient way to monitor the performance of the various methods. After each call to a SLES routine, you may request the number of floating-point operations and amount of memory used so far.

The macro `SVGetFlops` gives the number of floating-point operations (flops) so far. The format is

`SVGetFlops( svctx, a )`

where `a` is an `int` that will be assigned the number of flops.

The routine `SVSetFlopsZero` sets the running count of flops to zero. The format is `SVSetFlopsZero(svctx)`.

The macro `SVGetMemory` gives the amount of space used by the solver context, in bytes. The format is

`SVGetMemory( svctx, a )`

where `a` is an `int` that will be assigned the amount of memory used.

The routine `SYGetCPUTime` returns the time used by the process in seconds. For elapsed time, the routines `SYusc_clock` and `SYuscDiff` may be used to access the elapsed time, with microsecond granularity on those systems that support fine-grain clocks, for example,

```
SYtime_t t1, t2;
SYusc_clock( &t1 );
<code to time>
SYusc_clock( &t2 );
printf( "Time is %lf\n", SYuscDiff( &t2, &t1 ) );
```

Note: These features are not fully implemented in the current release. These routines will return values, but they may be underestimates.

Chapter 7

Adding New Methods

You can easily add new methods to the SLES package, without changing a single line of code in the SLES package. You simply call a routine to insert the new methods into a list of known methods. These methods are then known to the appropriate `GetMethod` routine as if it were part of the core package. Thus, with the SLES package you can make use of new developments in algorithms and implementations without changing your program; you need only relink your code. No longer must you choose between using older algorithms or making significant changes to your code.

7.1 Method registry

A method is added to the SLES package by registering a creation routine. You do this with the routine `SVRegister`. The format of this routine is

```
SVRegister( id, name, routine )
int  id;
char *name;
void *(*routine)( SVCtx *svctx, SpMat *mat );
```

where `id` is a unique identifier, `name` is a string that names the method, and `routine` is a routine that creates the method. The best way to see how to write such a routine is to examine the implementation of the methods included with the SLES package, such as `SVILU` (in '`tools.core/solvers/ilu.c`'or `SVOSM` (in file '`tools.core/solvers/osm.[ch]`'). After this routine is called, `SVGetMethod` will accept `name` as a known solver method and return `id` and the corresponding solver's type.

7.2 Accelerator registry

Iterative accelerators are registered in much the same way as solver methods, using the routine `ITRegister` instead of `SVRegister`. `ITGetMethod` will accept `name` as a known iterative accelerator.

7.3 Matrix orderings registry

Matrix orderings (for sparse direct factorizations) are registered with the routine `SpOrderRegister`. `SpGetOrdering` will now accept `name` as a known matrix reordering.

7.4 Sparse matrix formats

Sparse matrices are not registered in the same way as solver methods or iterative accelerators. Instead, you must write three routines that respectively create, extract a row from, and insert a row into the matrix. In many cases, an additional routine that forms a row-format SMEIT sparse matrix from an existing one should also be written. However, once a SMEIT sparse matrix is generated, all of the other routines in SLES will work. The details of this process are beyond the scope of this manual.

You may also provide other routines, such as a matrix-vector product that is optimized for the particular data structure. For each of these routines, if you do not provide one, a default routine (built by using the routine to extract rows from the matrix) will be used. Thus, all of the operations used by the solver package are available regardless of the sparse matrix data-structure; for more efficient execution, you may provide customized routines.

7.5 Restricting the choices

One disadvantage of providing a large variety of methods, all of which are available at run time, is that it can greatly increase the size of an executable program. While this may not be serious for virtual-memory machines, it can be a major problem for programs running on massively parallel computers, where each processor has a copy of the executable image. To change the routines that are *loaded* when the executable is created, write a routine called `SVRegisterAll`, and link it into your program ahead of the PETSc library. This routine should contains `SVRegister` calls for each method desired. For example, to restrict the program to direct LU factorization and ILU preconditioning, use

```
void SVRegisterAll()
{
  SVRegister(SVLU,    "lu",     SViCreateLU);
  SVRegister(SVILU,   "ilu",    SViCreateILU);
}
```

Similarly, the iterative methods may be restricted by providing a routine `ITRegisterAll`. For example, to restrict the iterative methods to CG and GMRES, use

```
void ITRegisterAll()
{
  ITRegister(ITCG,          "cg",         ITCGCreate);
  ITRegister(ITGMRES,       "gmres",      ITGMRESCreate);
}
```

The matrix orderings for the direct sparse factorizations may be restricted by providing a routine `SpOrderRegisterAll`. For example, to restrict the orderings to nested dissection, use

```
void SpOrderRegisterAll()
{
  SpOrderRegister(ORDER_ND,   "nd",  SpOrderND);
}
```

To recover the small amount of space used by the registries, use the routine `SVRegisterDestroy` for the solver's registry, the routine `ITRegisterDestroy` for the iterative accelerator's registry, and the routine `SpOrderRegisterDestroy` for the ordering's registry. Of course, once the registries are eliminated, they are no longer available.

Chapter 8

Using SLES with Fortran

SLES is fully callable from Fortran. All routines have the same names as the C versions. The arguments follow the usual Fortran conventions; you do not need to worry about passing pointers or values.

All "pointers" should be declared as integers in Fortran; this includes the solver context variable (`svctx`). The include file '`tools.core/solvers/svfort.h`' contains parameter definitions for the solver and iterative methods and matrix orderings. Error messages generated by the PETSc package at run time will not indicate the Fortran file and line number where they occurred; instead, they will indicate the line in the interface file. Other than this, everything is the same.

The library '`tools.core/fort/$(ARCH)/fort.a`' provides a Fortran interface to the SLES routines. This library *must* occur ahead of the '`tools`' libraries. For example, this makefile fragment links a Fortran program (`example`) with the appropriate libraries:

```
ITOOLSDIR = /usr/local/tools.core
LDIR      = $(ITOOLSDIR)/libs/libs$(BOPT)$(PROFILE)/$(ARCH)
LIBS      = $(LDIR)/tools.a $(LDIR)/system.a -lm
FLIBS     = $(ITOOLSDIR)/fort/$(ARCH)/fort.a
include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)$(PROFILE)
include $(ITOOLSDIR)/bmake/$(ARCH)
example: example.o
         $(FLINKER) -o example $(BASEOPTF) example.o $(FLIBS)
$(LIBS) \
                      $(LAPACK) $(BLAS)
         $(RM) example.o
```

This assumes that SLES is installed in '`/usr/local/tools.core`' and that the program may be using LAPACK and/or the BLAS (these are standardized linear algebra packages available on many systems. In particular, some vendors provide optimized versions of the BLAS that significantly out perform portable versions). Programs that do not use these routines can leave the libraries `$(LAPACK)` and `$(BLAS)` out. Note that the dense matrix routines (associated with `SpDnCreate` and `SpDnCreateFromData`) require both LAPACK and the BLAS.

This interface library is constructed automatically from the C program files. Thus, it should always match the C versions (any new routine added to SLES automatically becomes available to both C and Fortran users; no special interface code needs to be written). If you are interested in how this is accomplished, look in the directory '`tools.core/c2fort`'. The program in this directory uses the same approach that is used to generate the manual pages from the C source files.

Chapter 9

Debugging and Optimizations

A number of tools are available to aid in debugging a program that uses the SLES package.

The simplest is to use `SpMult` to check that the computed solution is actually a good one by calculating the residual with the original matrix. Note that if the original matrix is very poorly conditioned, a preconditioned iterative method could find an excellent solution to the preconditioned problem that was a poor solution to the original problem.

Another tool that can help in debugging a SLES code is the iterative monitor (set with `SVSetMonitor`) which displays, for example, the solution or the residual at the current iteration. The PETSc package provides some graphics display tools that may also be helpful. These include `ITXMonitor` (graphs the norm of the residual) and `ITXMonitorRVal` (graphs the value of the residual vector on a rectangular mesh). The routine `XBQMatix` may be used to display the matrix; this can be useful in verifying that the problem being solved is the one intended.

Finally, do not forget to take advantage of the debugging library in '`tools.core/libs/libsg`' and `dbx`. For performance debugging, use the profiling library, in '`tools.core/libs/libsOpg`', and the `-pg` compiler switch (or `BOPT=Opg`, if you are using the PETSc makefiles) when compiling and linking your application. Then the usual tools, such as `gprof`, may be used to gain insight into the execution-time performance of the program.

9.1 Error messages

The debugging version of the PETSc package will generate error tracebacks of the form

`Line` *linenumber* `in` *filename: message*
`Line` *linenumber* `in` *filename: message*
...
`Line` *linenumber* `in` *filename: message*

The first line indicates the file where the error was detected; the subsequent lines give a traceback of the routines that were calling the routine that detected the error. A message may or may not be present; if present, it gives more details about the cause of the error.

The production libraries (‘`libsO/tools.a`’) are often built without the ability to generate these tracebacks (or even detect many errors).

9.2 Performance debugging

There are a number of ways to identify performance bugs or problems. One is to look at the achieved computational rate (so-called megaflops) for the setup and solve phases. Values that are low (relative, for example, to the LINPACK benchmark numbers) may indicate that the implementation is not making effective use of the computer hardware. This problem may be caused, for example, by the choice of sparse matrix format. Choosing a different format, particularly one that is not dynamic (such as IJA or AIJ), may give better performance.

More detailed information may be gather by using the profiling library (‘`tools.core/libs/libsOpg/$ARCH/tools.a`’ and the `gprof` utility (available on most though not all Unix systems).

Chapter 10

Hints on Choosing Methods

This chapter contains some hints for choosing the methods to use in the SLES package. Please note that each problem is different and may have special features that make other choices more appropriate.

If the problem is small (n<100), use `SVLU` with the default ordering (`ORDER_ND`). Direct factorization may also be appropriate if high relative accuracy is required and the problem is of moderate size.

For larger problems, if the matrix is symmetric, use either of the approximate factorizations `SVICC` or `SVICCJP`, or `SVSSOR`, with iterative accelerator `ITCG`.

For nonsymmetric matrices, use incomplete factorization with fill (`SVILU`) and an iterative accelerator `ITGMRES`, `ITBCGS`, or `ITTFQMR`.

Large systems of equations that are generated by discretizations of PDEs can solved more effectively with `SVOSM` as long as a global problem can be provided.

Some singular systems can be solved if the null-space is known in advance. For example, the "pressure" equation often has Neumann boundary conditions, making the vector $(1, 1, \ldots, 1)$ a null-vector for the matrix.

Chapter 11

Examples

This chapter contains a few examples of programs that use SLES to solve linear systems. More examples may be found in the directory 'tools.core/solvers/examples'.

11.1 Poisson problem

Our first example solves a simple Poisson problem on the unit square using any of the available methods. A C shell script following this example shows how to generate a table comparing 68 different methods for this model problem. This example demonstrates how easy it is to produce a comparison of methods by using this package.

```
#include "tools.h"
#include "solvers/svctx.h"
main(argc,argv)
int  argc; char *argv;
{
SVMETHOD    svmethod;
ITMETHOD    itmethod;
SPORDERTYPE ordering;
SpMat       *mat;
SVCtx       *svctx;
int         n, m;
double      *b, *x;

n        = 16;
```

```
svmethod = SVILU;
itmethod = ITGMRES;
ordering = ORDER_ND;
SYArgGetInt( &argc, argv, 1, "-n", &n );
SVGetMethod( &argc, argv, 0, &svmethod );
ITGetMethod( &argc, argv, 0, &itmethod );
SpGetOrdering( &argc, argv, 0, &ordering );

m     = n * n;
mat   = FDBuildLaplacian2d( n, n, 0.0, 0.0, 0.0 );
svctx = SVCreate( mat, svmethod );
SVSetAccelerator( svctx, itmethod );
SVSetLUOrdering( svctx, ordering );
SVSetUp( svctx );

x = DVCreateVector( &m ); DVset( &m, 1.0, x );
b = DVCreateVector( &m );
SpMult( mat, x, b );
printf( "Solved in %d iterations\n",
         SVSolve( svctx, b, x ) );
}
```

To produce a table comparing various methods, run this csh script:

```
foreach ordering ( nd rcm qmd 1wd )
   echo "lu $ordering"
   example -n $N -sv lu -ordering $ordering
end
foreach svmethod (jacobi ssor ilu icc \
                  iccjp bdd osm nopre)
   foreach itmethod ( richardson chebychev cg \
                      gmres tcqmr bcgs cgs tfqmr )
      echo "$svmethod $itmethod"
      example -n $N -sv $svmethod -itmethod $itmethod
   end
end
```

11.2 Fortran example

This example shows how SLES may be used from Fortran. This example solves a simple 1-dimenisional Poisson problem. A sample makefile follows the program, showing how to use the Fortran interface libraries.

```
      Program Main
C     Include PARAMETERS for methods etc.
      include '/usr/local/tools.core/solvers/svfort.h'
      integer   spcreate, svcreate
      integer   svsolve
c
c
      parameter ( N = 20 )
      integer          matrix, solver, flag, its
      double precision b(N), x(N), r(N), norm
c
c   create example right hand side
c
      do 10 i=1,N
         b(i) = i
 10   continue
c
c   create example matrix, 1D discrete Laplaciain
c
      flag = 0
      matrix = spcreate(N,N,0)
      if ( matrix .eq. 0 ) goto 100
c
      call spaddvalue(matrix,2.d0,0,0)
      call spaddvalue(matrix,-1.d0,0,1)
      do 20 i=1,N-2
        call spaddvalue(matrix,-1.d0,i,i-1)
        call spaddvalue(matrix,2.d0,i,i)
        call spaddvalue(matrix,-1.d0,i,i+1)
 20   continue
      call spaddvalue(matrix,2.d0,N-1,N-1)
      call spaddvalue(matrix,-1.d0,N-1,N-2)
c
c   create solver context
c
      flag = 2
      solver = svcreate(matrix,SVSSOR)
```

```
      if ( solver .eq. 0 ) goto 100
c
c   set nested dissection for matrix ordering
c
      call svsetluordering(solver,ORDER_ND)
c
c   setup solver context
c
      call svsetup(solver)
      call svsetrelativetol(solver,1.d-8)
c
c   solve a linear system
c
      flag = 4
      its = svsolve(solver,b,x)
      if ( its .eq. -1 ) goto 100
c
c   free the space used by solver
c
      call svdestroy(solver)
c
c   calculate residual
c
      call spmult(matrix,x,r)
      norm = 0.d0
      do 30 i=1,N
        r(i) = b(i) - r(i)
        norm = norm + r(i)*r(i)
 30   continue
      norm = dsqrt(norm)
      print*, 'Iterations ',its, ' Norm of residual',norm
c
c   free the space used by matrix
c
      call spdestroy(matrix)
      stop

 100  continue
      print*, 'Error in call to SLES libraries',flag
      end
```

The makefile for this program is (from '`tools.core/solvers/examples/makefile`'):

```
ALL:  example
```

```
ITOOLSDIR = /usr/local/tools.core

LDIR     = $(ITOOLSDIR)/libs/libs$(BOPT)$(PROFILE)/$(ARCH)
LIBS     = $(LDIR)/tools.a $(LDIR)/Xtools.a $(LDIR)/system.a
FLIBS    = $(ITOOLSDIR)/fort/$(ARCH)/fort.a

example: example.o
        $(FLINKER) -o example $(BASEOPTF) example.o \
                      $(FLIBS) $(LIBS)

include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)$(PROFILE)
include $(ITOOLSDIR)/bmake/$(ARCH)
```

11.3 Graphical interface

SLES can be used with a graphical user interface (GUI). Tcl/Tk (a shell-based X interface) makes it easy to provide a menu-driven interface.

Figure 11.1 gives an example of the type of interface that can be constructed with SLES and Tck/Tk.
The source code for this interface can be found in
'`tools.core/solvers/examples/winex`'.

This interface allows you to use menus to choose the solver method and accelerator, parameters (as appropriate, by method and accelerator), and problem size. The graphics area allows for the display of the norm of the residual as a function of interation, or for the actual residual, displayed as a two-dimensional contour plot. A summary line is displayed at the end of the computation in the text window at the bottom of the interface window.

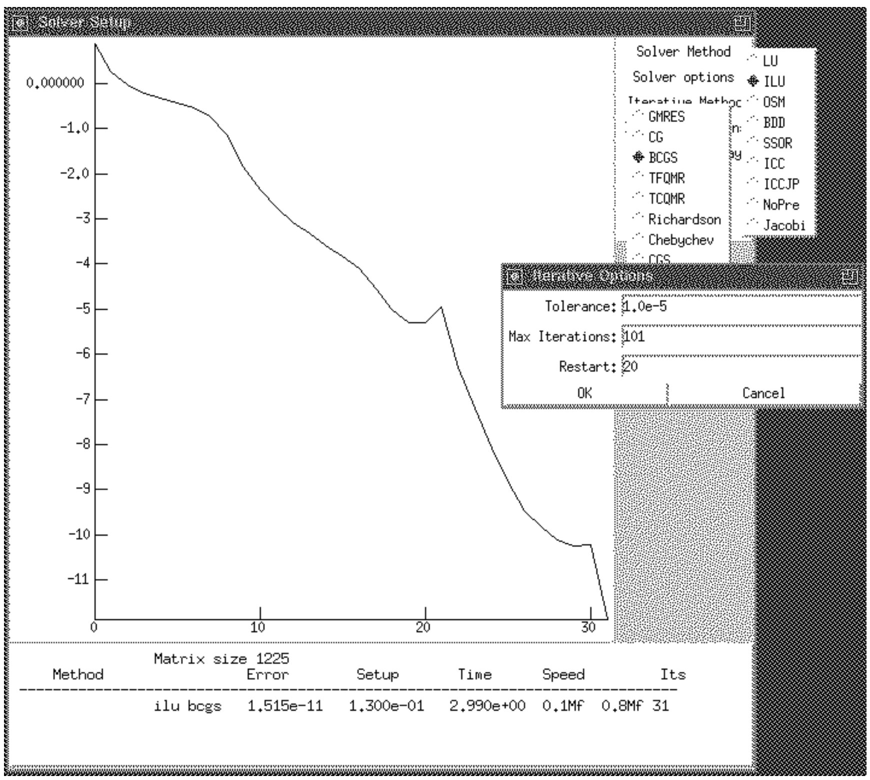

Figure 11.1: Example graphical interface to SLES

Chapter 12

Summary of Routines

This chapter contains a brief summary of the routines in this manual, as well as a few routines (e.g., `SpPrint`) from PETSc that may be of use. This chapter is organized into three major parts: the SLES routines (`SVxxx`), the sparse-matrix routines (`Spxxx`), and miscellaneous routines. This last section has three subsections, containing routines for system timers, registering iterative accelerators, and graphics tools that are designed to work with the SLES routines. The beginning of each section lists the include files that are needed by C programmers. Fortran users should use the '`svfort.h`' file.

12.1 SLES routines

```
#include "tools.h"
#include "solvers/svctx.h"
```

`SVctx *SVCreate(mat,name)`
`SpMat *mat;`
`SVMETHOD name;`
	Given a sparse matrix, creates an SVctx structure; this structure will then be used in solving linear systems.

`MACRO void SVDestroy(ctx)` `SVctx *ctx;`	Destroys a solver context created by SVCreate().

`MACRO void`
`SVGetFlops(ctx,flops)`
`SVctx *ctx;`
`int *flops;`
	Returns number of flops used related to solver context since creation of solver context or since a call to SVSetFlopsZero().

`MACRO void SVGetICCAlpha( ctx, alpha )`
`SVctx *ctx;`
`double *alpha;`
	Returns shift factor needed for incomplete Choleski factorization before positive definite preconditioner was found.

`MACRO void SVGetICCFailures( ctx, count )`
`SVctx *ctx;`
`int *count;`
	Returns the number of shifts needed for incomplete Choleski factorization before positive definite preconditioner was found.

`MACRO void SVGetMemory(ctx,mem)`
`SVctx *ctx;`
`int *mem;`
	Returns the amount of space used by the solver context.

`void SVGetMethod( Argc, argv, sname, svmethod )`
`int *Argc;`
`char *argv, *sname;`
`SVMETHOD *svmethod;`
	Given the argument list, return the selected method

`void SVRegisterAll()`	This routine registers all the SERIAL linear system solvers in the SV package.
`void SVRegisterDestroy()`	Frees the list of SERIAL iterative solvers which have been registered.

`void SVRegister( name, sname, create )`
`int name;`
`char *sname;`
`void (*create)();`
	Given a solver name (integer) and a function pointer; adds the solver to the SERIAL solver package.

`MACRO void SVSetAbsoluteTol( ctx, tol )`
`SVctx *ctx;`
`double tol;`
	Sets the absolute tolerance for convergence.

`MACRO void SVSetAccelerator( ctx, type )`
`SVctx *ctx;`
`ITMETHOD type;`
	Sets the type of accelerator to use for the iterative process.

`void SVSetBDDRegularDomains2d( ctx, n1, n2, nc )`
`SVctx *ctx;`
`int n1, n2, nc;`
	Set the domains for a n1 x n2 regular mesh

`MACRO void SVSetConvergenceTest( ctx, converge, cctx )`
`SVctx *ctx;`
`int (*converge)();`
`void *cctx;`
	Sets the function that is to be used to determine convergence.

`MACRO void SVSetFlopsZero( ctx )` `SVctx *ctx;`	Resets the flop counter associated with a solver context. See SVGetFlops().

`MACRO void SVSetGMRESRestart( ctx, its )`
`SVctx *ctx;`
`int its;`
	Sets the number of iterations before using a restart for GMRES.

`MACRO void SVSetILUDropTol( ctx, fill )`
`SVctx *ctx;`
`int fill;`
	Sets the drop tolerance for the incomplete LU preconditioner.

`MACRO void SVSetILUFill( ctx, fill )`
`SVctx *ctx;`
`int fill;`
	Sets the level of fill for the incomplete LU preconditioner.

`MACRO void SVSetILUPivoting( ctx, pivoting )`
`SVctx *ctx;`
`int pivoting;`
	Sets the pivoting type to be used for the factorization in the linear system solve using LU.

`MACRO void SVSetIts( ctx, max_its )`
`SVctx *ctx;`
`int max_its;`
	Sets the maximum number of iterations allowed.

`MACRO void SVSetLUOrdering( ctx, ordering )`
`SVctx *ctx;`
`int ordering;`
	Sets the order type to be used for the factorization in the linear system solve using LU.

`MACRO void SVSetLUPivoting( ctx, pivoting )`
`SVctx *ctx;`
`int pivoting;`
	Sets the pivoting type to be used for the factorization in the linear system solve using LU.

`MACRO void SVSetLUThreshold( ctx, threshold )`
`SVctx *ctx;`
`int threshold;`
	Sets the minimum block size to use in the LU factorization. Four is generally a good number.

`MACRO void SVSetMonitor(ctx, monitor, mctx)`
`SVctx *ctx;`
`void (*monitor)(), *mctx;`
	Sets the routine that monitors the residual at each iteration of the iterative method. The default simple prints the residual at each iteration. Look in the iter directory for more information.

`void SVSetNullSpace( ctx, has_cnst, nv, v )`
`SVctx *ctx;`
`int has_cnst, nv;`
`double **v;`
	Sets the null space for a linear system.

`void SVSetOSMRegularDomains2d( ctx, n1, n2, nc )`
`SVctx *ctx;`
`int n1, n2, nc;`
	Sets the domains for a n1 x n2 regular mesh.

`void SVSetOSMRegularOverlap2d( ctx, n1, n2, nc, w1, w2 )`
`SVctx *ctx;`
`int n1, n2, nc, w1, w2;`
	Sets the overlap indices for a n1 x n2 regular mesh.

`MACRO void SVSetPrecondMat( ctx, bmat )`
`SVctx *ctx;`
`SpMat *bmat;`
	Set the matrix to be used for the preconditioning.

`MACRO void SVSetRelativeTol(ctx, tol)`
`SVctx *ctx;`
`double tol;`
	Sets the tolerance for convergence; by default it is a relative decrease in the two-norm of the residual of tol.

`MACRO void SVSetSSOROmega(ctx,`
`omega)`
`SVctx *ctx;`
`double omega;`
	Sets the relaxation factor for SSOR. The default is one.

`MACRO void SVSetUp(ctx)` `SVctx *ctx;`	Called after a call to SVCreate(), allocates space which will be needed later in the call to SVSolve().

`MACRO void`
`SVSetUseInitialGuess(`
`ctx, flag )`
`SVctx *ctx;`
`int flag;`
	Use the value in "x" as the initial guess for iterative solvers.

`MACRO int SVSolve(ctx,b,x)`
`SVctx *ctx;`
`void *b,*x;`
	Solves the linear system. Called after a call to SVCreate() and a call to SVSetUp().

12.2 Sparse matrix routines

`SpMat *SpAIJCreateFromData( nr,`
`nc, ia, ja, a, maxnz`
`)`
`int nr, nc, *ia, *ja, maxnz;`
`double *a;`
	Creates an AIJ matrix descriptor, given an existing AIJ format matrix

`SpMat *SpAIJCreate( nr, nc,`
`maxnz )`
`int nr, nc, maxnz;`
	Creates an AIJ matrix descriptor

`void SpAddInRow( mat, row, n,`
`v, c )`
`SpMat *mat;`
`int row, n, *c;`
`double *v;`
	Adds a row to a sparse matrix.

`void SpAddValue( mat, val, i, j`
`)`
`SpMat *mat;`
`double val;`
`int i, j;`
	Adds to an entry in a matrix. If the entry is not present, creates it.

`SpMat *SpCreate( n, m, mmax )` `int n, m, mmax;`	Allocates an n x m sparse matrix (row format).
`void SpDestroy( m )` `SpMat *m;`	Frees a sparse matrix (any format).

`SpMat *SpDnClampToSparse( mat,`
`          rtol, n )`
`SpMat *mat;`
`double rtol;`
`int n;`
	Forms a sparse matrix from a dense one.

`SpMat *SpDnCreateFromData( nr,`
`          nrd, nc, p )`
`int nr, nrd, nc;`
`double *p;`
	Creates a dense matrix descriptor, given an existing dense matrix.

`SpMat *SpDnCreate( nr, nc )` `int nr, nc;`	Creates a dense matrix descriptor.

`void SpGetOrdering( Argc, argv,`
`          sname, ordering )`
`int *Argc;`
`char *argv, *sname;`
`SPORDERTYPE *ordering;`
	Return the selected ordering, given the argument list.

`void SpMult( m, vin, vout )`
`SpMat *m;`
`double *vin, *vout;`
	Computes a matrix-vector product.

`void SpOrderRegisterAll()`	Registers all the matrix-ordering methods.
`void SpOrderRegisterDestroy()`	Frees the list of ordering routines that have been registered.

`void SpOrderRegister(name,`
`          sname, order)`
`SPORDERTYPE name;`
`char *sname;`
`void (*order)();`
	Given a matrix ordering routine and an integer, registers that ordering routine, so that a user can call SpOrder() with that integer and have the correct ordering routine called.

`void SpPrintMatlab( fp, B, name`
`          )`
`FILE *fp;`
`SpMat *B;`
`char *name;`
	Prints a sparse matrix to a given FILE, in MATLAB format.

`void SpPrint( fp, B )`
`FILE *fp;`
`SpMat *B;`
	Prints a sparse matrix to a given FILE.

`void SpSetInRow( mat, row, n,`
`         v, c )`
`SpMat *mat;`
`int row, n;`
`int *c;`
`double *v;`
	Sets a row in a sparse matrix.

12.3 Miscellaneous routines

12.3.1 Iterative method routines

```
#include "tools.h"
#include "solvers/svctx.h"
```

`void ITGetMethod( Argc, argv,`
`         sname, itmethod )`
`int *Argc;`
`char **argv, *sname;`
`ITMETHOD *itmethod;`
	Returns the selected method, given the argument list.

`void ITRegisterAll()`	Registers all the iterative methods in the IT package. To prevent all the methods from being registered and thus save memory, copy this routine and register only those methods desired.
`void ITRegisterDestroy()`	Frees the list of iterative solvers registered by ITRegister().

`void ITRegister(name, sname,`
`         create)`
`int name;`
`char *sname;`
`ITCntx *(*create)();`
	Adds the iterative method to the iter package, given an iterative name (ITMETHOD) and a function pointer.

12.3.2 System timers

```
#include "tools.h"
#include "system/system.h"
```

`double SYGetCPUTime()`	Returns the time in seconds used by the process.

```
#include "tools.h"
#include "system/system.h"
#include "system/time/usec.h"
```

`double SYuscDiff( a1, a2 )` `SYusc_time_t *a1, *a2;`	Returns the difference between two values.
`double SYuscValue( a )` `SYusc_time_t *a;`	Converts a fast μsec clock value into seconds.
`MACRO void SYusc_clock(a)` `SYusc_time_t *a;`	Gets a timer value (elapsed time) with high resolution.

12.3.3 Graphics tools

```
#include "tools.h"
#include "solvers/svctx.h"
```

`void ITXMonitorLimits( maxit, mres )`
`int maxit;`
`double mres;`
	Sets the limits for the residual monitor.

`void ITXMonitorMeshSize( nx, ny )` `int nx, ny;`	Sets the mesh size for the display of 2-d contour plots in monitoring iter package performance.

`void ITXMonitorMultiComponent( nc, ncx, ncy, extract, cmp )`
`int nc, ncx, ncy, cmp;`
`void (*extract)();`
	Informs ITXMonitor how to handle multicomponent problems.

`void ITXMonitorRVal( itP, usrP,`
`          n, rnorm )`
`ITCntx *itP;`
`void *usrP;`
`int n;`
`double rnorm;`
	Simple X Windows code to display the value of the residual at each iteration in the iterative solvers.

`void ITXMonitor( itP, usrP, n,`
`          rnorm )`
`ITCntx *itP;`
`void *usrP;`
`int n;`
`double rnorm;`
	Simple X Windows code to display the residual at each iteration in the iterative solvers.

`void XBQMatrix( mat, nc )`
`SpMat *mat;`
`int nc;`
	Displays a sparse matrix on the display.

Acknowledgments

The work described in this report has benefited from conversations with and use by a large number of people. Among the contributers are David Keyes and David Levine, who read the drafts, and the early users who requested additional functionality and were patient with our bug fixes.

Bibliography

[1] M. Dryja and O. Widlund. Towards a unified theory of domain decomposition algorithms for elliptic problems. In T. F. Chan, R. Glowinski, J. Périaux, and O. B. Widlund, editors, *Third International Symposium on Domain Decomposition Methods*, pages 3–21, Philadelphia, 1990. SIAM.

[2] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[3] Mark T. Jones and Paul E. Plassmann. An improved incomplete Cholesky factorization. Preprint MCS-P206-0191, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Il., 1991.

Function Index

X