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

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-93/30

Users Manual for KSP
Data-Structure-Neutral Codes
Implementing Krylov Space Methods

by

William Gropp
Mathematics and Computer Science Division

Barry Smith
Department of Mathematics
University of California, Los Angeles

Current address:
Mathematics and Computer Science Division
Argonne National Laboratory

August 1994

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy and in part by the Office of Naval Research under Contract ONR N0001492-J1890.

MASTER
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED *de*

DISCLAIMER

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

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Contents

Abstract	1
1 Introduction	2
1.1 Data-Structure-Neutral Programming	2
1.2 Organization of the Code	3
1.2.1 Vector Context	3
1.2.2 Iterative Context	4
1.3 Using KSP with PETSc	4
1.3.1 Makefiles	4
1.3.2 Linking	5
1.3.3 Using KSP from Fortran	6
1.3.4 Installing the PETSc Package	7
1.4 Further Information	7
1.5 Why This Approach?	8
2 Using the Krylov Space Methods Package	9
2.1 Getting Started	9
2.2 Linear Operator and Preconditioner	12
2.3 Vector Operations	13
2.3.1 Fortran-defined Vectors	13
2.3.2 Writing a New Vector Class	14
3 Optional Parameters	16
3.1 Common Parameters	16
3.2 Parameters for Particular Methods	17
3.3 Functions	17
3.4 Command Line Arguments	19
4 Adding New Methods	20
4.1 Solve	20
4.1.1 Work Arrays	22
4.1.2 Vector Operations	22
4.2 Create	22
4.3 Setup	23
4.4 Destroy	23

5	Debugging and Optimization	24
5.1	Error Messages	24
5.2	Performance Debugging	24
5.3	Troubleshooting	25
6	Hints on Choosing Methods	26
7	Summary of Routines	27
7.1	Common Routines	27
7.2	Method-specific Routines	30
7.3	Routines for Adding Methods	31
7.4	Vector Routines	31
8	Examples	34
8.1	Fortran	34
8.2	C	37
	Bibliography	39
	Function Index	40

Users Manual for KSP Data-Structure-Neutral Codes Implementing Krylov Space Methods

by

William Gropp and Barry Smith

Abstract

The combination of a Krylov space method and a preconditioner is at the heart of most modern numerical codes for the iterative solution of linear systems. This document contains both a users manual and a description of the implementation for the Krylov space methods package KSP included as part of the Portable, Extensible Tools for Scientific computation package (PETSc). PETSc is a large suite of data-structure-neutral libraries for the solution of large-scale problems in scientific computation, in particular on massively parallel computers.

The methods in KSP are conjugate gradient method, GMRES, BiCG-Stab, two versions of transpose-free QMR, and others. All of the methods are coded using a common, data-structure-neutral framework and are compatible with the sequential, parallel, and out-of-core solution of linear systems. The codes make no assumptions about the representation of the linear operator; implicitly defined operators (say, calculated using differencing) are fully supported. In addition, unlike all other iterative packages we are aware of, the vector operations are also data-structure neutral. Once certain vector primitives are provided, the same KSP software runs unchanged using any vector storage format. It is not restricted to a few common vector representations.

The codes described are actual working codes that run on a large variety of machines including the IBM SP1, Intel DELTA, workstations, networks of workstations, the TMC CM-5, and the CRAY C90. New Krylov space methods may be easily added to the package and used immediately with any application code that has been written using KSP; no changes to the application code are needed.

Chapter 1

Introduction

The data-structure-neutral Krylov space iterative package KSP provides a powerful interface to a variety of Krylov space methods for solving systems of linear equations

$$Ax = b.$$

The KSP software is extremely flexible and hence requires a slightly more sophisticated user than much of the rest of the Portable Extensible Tools for Scientific computation (PETSc) package, of which KSP is part. For many application programmers, direct use of the KSP package may be unnecessary. We have written two software packages, SLES (Simplified Linear Equation Solvers) and Parallel SLES, that sit on top of KSP and provide much of the functionality but are simpler to use. These packages provide access to all of the Krylov space methods contained in KSP and many standard preconditioners, but allow for only a limited number of vector representations. We recommend that you first review the documentation for these two packages [4]; if neither package suits your needs—for instance, your application requires an out-of-core use of a matrix-free Newton's method—then you should continue with this document.

This document is not an introduction to Krylov space methods; a recent introduction is available by Freund, Golub, and Nachtigal [1]. If you are interested in the mathematical aspects of these methods, please refer to that document. To use the KSP software, you do **not** have to understand the mathematical aspects of Krylov space methods; indeed you do not even need to know the algorithms or the particular implementations used in KSP.

1.1 Data-Structure-Neutral Programming

To appreciate the design of the KSP package, you must understand the motivation and philosophy behind data-structure-neutral programming. In brief, data-structure-neutral programming is a practical attempt to write portable, reentrant, flexible, and—most important—reusable code in an efficient manner. This is achieved by writing as much of the code as possible in a manner that is independent of the underlying representation of the data. Only the small pieces of code that actually manipulate the data will depend on its representation.

For the solution of linear systems there are essentially two types of data: vectors and finite-dimensional linear operators. (Actually there is a third, scalars, but in our particular implementation we restrict ourselves to double-precision numbers.) In a flexible data-structure-

neutral package, these vectors and linear operators should be thought of as abstract objects onto which certain operations may be performed, not as arrays of numbers.

KSP is designed around these abstract vectors and linear operators. In fact, at the very core, KSP does not provide any of these operations. Rather, you provide realizations of these operations that are appropriate to your application. However, since many applications use fairly conventional vector operations, KSP provides default versions of the vector operations. The matrix operations are more dependent on particular applications; the SLES and Parallel SLES packages provide a higher-level interface to KSP that provides some specific matrix operations.

Data-structure-neutral codes that use block Krylov space methods—that is, solve for several right-hand sides simultaneously for higher efficiency and faster convergence—may also be written in a data-structure-neutral manner. Because of our resource limitations, however, we do not yet support this extension.

1.2 Organization of the Code

KSP uses two *context* variables: a vector context, `VECctx`, and an iterative context, `ITCctx`. The code is organized about these two variables.

In their simplest form, context variables are a way of avoiding long calling sequences. More generally, they can be used to avoid all global states and thus allow the flexible nested (reentrant) use of the routines. The standard use of context variables is the following.

- Create a context variable.
- Set various properties of the context variable.
- Use the context variable; that is, solve one or several problems.
- Free any space used by the context variables.

To increase flexibility even more, the context variables themselves generally contain pointers to functions that act on the data. In this way, you can, for instance, change at run time the vector operations used in the iterative solvers, etc.

1.2.1 Vector Context

Our vector context contains pointers to all the vector primitives. Several sets of default vector primitives are provided in the `vectors` directory. The default parallel version was written using Chameleon [2], our portable message-passing system, which is part of PETSc. To use the parallel version, you will have to obtain the entire PETSc package, not just the SLES subset; see Section 1.3.4. If you need to introduce a new vector storage format for a particular application, see Section 2.3.

Within each vector context is a pointer to a *private* vector context. The private vector context contains any additional information used for that particular implementation. For instance, for standard serial vectors it simply contains the length of the vector stored as an integer. For a simple parallel implementation it may contain the length of the local piece of the vector and information about which processors share the vector.

1.2.2 Iterative Context

The iterative context is rather complicated because it must allow access to most of the internals of the Krylov space methods and to many optional parameters and arguments. Generally one should not access the data in the iterative context directly but rather through the large set of macros and subroutines outlined in the later chapters.

The iterative context itself contains a private subcontext that differs for each of the various Krylov space methods. This private context variable contains the data structures and other odds and ends peculiar to a particular Krylov space method. It is much like the private vector context. The private iterative context need be of concern only to those who are implementing *new* Krylov space methods using the KSP framework. Users of KSP need not concern themselves with details about it.

The iterative context also contains a single vector context that determines which vector primitives are to be used in the Krylov space codes.

Many other features of the iterative context are discussed in the later chapters.

1.3 Using KSP with PETSc

KSP is part of a larger package of tools called PETSc. This section describes how to use KSP as part of PETSc. This discussion includes the portable makefiles and the Fortran interface.

1.3.1 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. Basically, these use a few variables to control exactly what options the makefile uses. For some examples, see the makefiles in any of the directories.

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

ARCH Architecture. Common values include `ARCH=sun4`, `ARCH=rs6000`, and `ARCH=intelnx`.
BOPT Level of optimization. Use `BOPT=g` for debugging, `BOPT=0pg` for profiling, and `BOPT=0` 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 your makefile. The most important of these are as follows:

BASEOPT Flags for the C compiler. These include options such as `-g`, which is automatically set when you use `BOPT=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 values are provided by the makefile system for your use; they should not be changed.

1.3.2 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/${ARCH}'
profiling      'tools.core/libs/libspg/${ARCH}'
production     'tools.core/libs/libso/${ARCH}'
```

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    = $(ITOOLSDIR)/libs/libso/${ARCH}
example: example.o
    $(CLINKER) -o example example.o \
        $(LIBDIR)/tools.a $(LIBDIR)/system.a \
        $(SLIB) -lm
include $(ITOOLSDIR)/bmake/${ARCH}.O
include $(ITOOLSDIR)/bmake/${ARCH}
```

The include lines include files that contain 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 workstations, on which PETSc runs.

PETSc includes some graphical aids for displaying, for example, the progress of the solution algorithm. Using these requires a few additional libraries. To the example above, we simply add the 'Xtools.a' library (part of PETSc) and the X11 Window System library 'X11LIB':

```
ITOOLSDIR = /usr/local/tools.core
LIBDIR    = $(ITOOLSDIR)/libs/libso/${ARCH}
example: example.o
    $(CLINKER) -o example example.o \
        $(LIBDIR)/Xtools.a $(LIBDIR)/tools.a \
        $(LIBDIR)/system.a $(X11LIB) $(SLIB) -lm
include $(ITOOLSDIR)/bmake/${ARCH}.O
include $(ITOOLSDIR)/bmake/${ARCH}
include $(ITOOLSDIR)/bmake/${ARCH}.x11
```

When using the default parallel vector routines, you need to add to the makefile an additional library, 'tools\$(COMM).a', and include file '\$(ITOOLSDIR)/bmake/\${ARCH}.\${(COMM)}'. For example:

```

ITOOLSDIR = /usr/local/tools.core
LIBDIR    = $(ITOOLSDIR)/libs/libs0/$(ARCH)
example: example.o
        $(CLINKER) -o example example.o \
            $(LIBDIR)/tools$(COMM).a \
            $(LIBDIR)/tools.a $(LIBDIR)/system.a \
            $(CLIB) $(SLIB) -lm
include $(ITOOLSDIR)/bmake/$(ARCH).O
include $(ITOOLSDIR)/bmake/$(ARCH)
include $(ITOOLSDIR)/bmake/$(ARCH).$(COMM)

```

The order of the include files is important. The variable COMM, which may be set at the 'make' command line or as an environmental variable, determines the type of parallel message-passing system to use. At present, it may be p4, pvm, eui, euih, or empty for Intel systems or Thinking Machines CM-5. See the Chameleon documentation [2] for more information.

1.3.3 Using KSP from Fortran

KSP is almost fully usable 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. Only a few of the routines that manipulate the command-line arguments or do memory allocation (such as VCREATE) are not available to the Fortran programmer.

All "pointers" should be declared as integers in Fortran, including the iterative context variable (itctx). The include file 'tools.core/iter/fiter.h' contains parameter definitions for the iterative methods. 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 as for the C version.

The library 'tools.core/fort/\$(ARCH)/fort.a' provides a Fortran interface to the PETSc routines. This library *must* occur ahead of the 'tools' libraries. For example, the following 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 PETSc 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 outperform portable versions). SLES comes with default BLAS and LAPACK libraries, in the directories 'blas' and 'lapack'. Programs that do not use these routines can leave out the libraries \$(LAPACK) and \$(BLAS).

This Fortran interface library is constructed automatically from the C program files. Thus, it should always match the C versions. Any new routine added to PETSc 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.

1.3.4 Installing the PETSc Package

The PETSc 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 related to the KSP package, as well as other code. The file 'ksp.ps.Z' is a compressed postscript file containing this document. To install the package, transfer the tar file, uncompress it, and do

```
tar xf sles.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 &
```

The file tools.core/readme contains more detailed instructions on how to install the entire package.

If you wish to use the default parallel vectors or any of the other features of PETSc, you should obtain the Chameleon package. This software and its documentation may be obtained from the same ftp site, under the names 'chameleon.tar.Z' and 'chameleon.ps'.

1.4 Further Information

Every routine mentioned in this manual has a Unix man page. For brevity, these are not attached to this document. They are found in 'tools.core/man' and may be accessed with the X window interface 'tools.core/bin/toolman'.

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'.

1.5 Why This Approach?

In order to best understand how to use this package, it is important to understand the rationale behind our design decisions. More discussion on this may be found in [3].

Given the concept of abstract vectors and linear operators, we first have to determine which operations must be applied to them for a particular class of applications. In our case, the class of applications is Krylov methods. These operations, which we will call *primitives*, can be determined only by experience with the applications. After coding several Krylov space accelerator methods, we arrived at the vector primitives that are needed for the majority of the accelerators (see Table 1.1).

Table 1.1: Vector Operations

Name	Description	Calling Sequence	BLAS Equivalent
Create	a vector		
Destroy	a vector	Vector *v	
Obtain	n vectors	int n	
Release	n vectors	int n, Vector **v	
Dot	$z \leftarrow x^H * y$	Vector *x, Vector *y, Scalar *z	DDOT
Norm	$z \leftarrow \sqrt{x^H * x}$	Vector *x, Scalar *z	DNRM2
Max	$z \leftarrow \max(x)$	Vector *x, Scalar *z, int *idx	
Scale	$x \leftarrow \alpha x$	Scalar α , Vector *x	DSCAL
Copy	$y \leftarrow x$	Vector *x, Vector *y	DCOPY
Set	$x_i \leftarrow \alpha, \forall i$	Scalar α , Vector *x	
AXPY	$y \leftarrow \alpha x + y$	Scalar α , Vector *x, Vector *y	DAXPY
AYPX	$y \leftarrow \alpha y + x$	Scalar α , Vector *x, Vector *y	
Swap	swap x and y	Vector *x, Vector *y	DSWAP
WAXPY	$w \leftarrow \alpha x + y$	Scalar α , Vector *x, Vector *y, Vector *w	

The vector primitives contain not only many of the standard Level 1 BLAS operations but also routines to create space for additional work vectors and routines to free vectors no longer needed, as well as several others.

The primitives for the linear operators for Krylov space methods are very simple. One must be able to apply the operator to a vector; and, for some methods, one must also be able to apply the transpose of the operator to a vector.

Once the vector and linear operator primitives had been decided on, we next decided on how the code should be logically organized to allow for flexibility, maintainability, reentrance, and ease of use. Again this is an on-going process. Even for a simple project like Krylov space methods, many iterations were needed in order to learn, by experience, which techniques work. We expect to continue to add to the operations supported by both the vector and iterative contexts as we continue to experiment with new methods. Fortunately, our approach allows us to do this in an (almost completely) upwardly compatible way.

Chapter 2

Using the Krylov Space Methods Package

To use the KSP package, you must provide the vector primitives and routines for the application of the linear operator and (optionally) the preconditioner. In addition, for certain methods, the transpose of the linear operator and preconditioner must also be provided. Several of these sets of routines are available in the SLES and Parallel SLES packages [4]; we recommend checking there first before writing your own code. KSP comes with two default vector packages; these should be suitable for almost all users.

In this chapter we first describe the procedure for solving linear systems once the linear operator, vector operations, etc., have been written. Then we explain the calling sequence of the linear operators required. Finally we discuss how new vector types may be introduced.

2.1 Getting Started

The standard use of the KSP package involves the following steps:

- Create an iterative context with `ITCreate()`.
- Set the vector context, linear operator, and preconditioner.
- Set various optional arguments.
- Call `ITSetup()`.
- Solve some linear systems with `ITSolve()`.
- Free the space used by KSP with `ITDestroy()`.

Though this might sound like a complicated procedure to solve a simple system of linear equations, it is actually easier than the technique of calling a single function with twenty or thirty arguments. Moreover, it offers much more flexibility.

A new iterative context, `ITctx`, is created with the command `itctx = ITCreate(ITMETHOD)`, where `ITMETHOD` is one of the installed methods. Currently, the package supports the following:

<code>ITCG</code>	Conjugate gradient
<code>ITCGS</code>	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

More are easily added; see Chapter 4.

Once the iterative context has been created and the linear operator and (optional) preconditioner code has been written, you must set the fields in the context that indicate that the code is to use them; this is done with `ITSetAmult(itctx, amult, amultP)` and `ITSetBinv(itctx, binv, binvP)`, where `amult` and `binv` are the linear operator and preconditioner routines. The arguments `amultP` and `binvP` are optional context variables containing, for example, pointers to matrices needed by the routines.

In addition, you must inform the iterative context of the location of the vector containing the right-hand side and the location of the vector where the solution is to be stored. You do this with `ITSetSolution(itctx, x)` and `ITSetRhs(itctx, b)`, respectively. To get back the location of the solution and right-hand side from the iterative context, you use `ITGetSolution(itctx)` and `ITGetRhs(itctx)`, respectively.

If you are using a nonstandard vector storage format, you must write the vector primitives as discussed in the preceding chapter and in Section 2.3 and inform the iterative context that it should be using the new, user-provided vector context. This is best done by writing a routine that, given the address of a vector context, fills in the locations (names) of the vector primitives; see, for example, the routine in `vectors/dvec.c`. So, for instance, if you are using the default serial vectors, you would use `DVSetDefaultFunctions( ITVectorContext( itctx ), n )`. For the default parallel vectors you would use `DVPSetDefaultFunctions( ITVectorContext( itctx ), n, pset )`.

After the optional arguments have been set (we will discuss these in Chapter 3), the routine `ITSetUp(itctx)` is called to allocate work space, etc. At this point the linear system may be solved by calling `ITSolve(itctx)`. Several linear systems may be solved sequentially by calling the macros `ITSetSolution(itctx, x)` and `ITSetRhs(itctx, b)` followed by repeated calls to `ITSolve(itctx)`. You must not call `ITSetUp(itctx)` between these calls.

When you no longer need to solve linear systems using a particular iterative context, you should call `ITDestroy(itctx)` to free all the space that has been internally allocated. If you are using your own `amult`, `binv`, etc., context variables (see below), you must remember to free that space, or it will be lost.

Below we list an artificial example using the KSP package.

```
#include <stdio.h>
#include "tools.h"
#include "iter/itall.h"
void amult(), binv();
main(Argc, Args)
int Argc;
```

```

char *Args[];
{
    ITCntx    *itctx;
    double    *x,*b;
    int        n = 100;
    ITMETHOD itmethod = ITCG;
    itctx = ITCreat(itmethod);
    /* set vector functions */
    DVSetDefaultFunctions( ITVectorContext( itctx ), n );
    /* allocate space for solution and right hand side */
    x = VCREATE(ITVectorContext( itctx ));
    b = VCREATE(ITVectorContext( itctx ));
    /* set convergence criteria */
    ITSetRelativeTolerance(itctx,1.e-10);
    ITSetIterations(itctx,25);
    /* set linear operator and preconditioner */
    ITSetAmult(itctx,amult, &n );
    ITSetBinv(itctx,binv, &n );
    /* set locations of rhs and solution vectors */
    ITSetSolution(itctx,x);
    ITSetRhs(itctx,b);
    /* solve linear system */
    ITSetUp(itctx);
    its = ITSolve(itctx);
    /* free allocated space */
    VDESTROY(ITVectorContext( itctx ),x);
    VDESTROY(ITVectorContext( itctx ),b);
    ITDestroy(itctx);
}

```

In Fortran, the same code is

```

      program main
      external amult, binv
      include "iter/fiter.h"
      double precision x(100), y(100)
      integer itctx, its
c
      itctx = ITCreat( ITCG )
      call DVSetDefaultFunctions( ITVectorContext( itctx ),
+                               100 )
      call ITSetRelativeTolerance(itctx,1.d-10)
      call ITSetIterations(itctx,25)
      call ITSetAmult(itctx,amult, n)
      call ITSetBinv(itctx,binv, n)
      call ITSetSolution(itctx,x)
      call ITSetRhs(itctx,b)

```

```

call ITSetUp(itctx)
call its = ITSolve(itctx)
call ITDestroy(itctx)
end

```

The only special feature to note is that the relative tolerance is written as 1.e-10 in C and 1.d-10 in Fortran. This quantity is *always* a double-precision quantity; Fortran and C have different syntaxes for floating-point numbers. Also, the value 0 in Fortran has the same effect as a null pointer in C.

Note that since Fortran has no convenient mechanism for allocating storage, the solution and right-hand side arrays are declared explicitly, while in C they are allocated with VCREATE.

2.2 Linear Operator and Preconditioner

Since the application-provided routines (i.e., `amult` and `binv`) are called directly by the Krylov space solvers code, they must follow a prescribed calling sequence.

The calling sequence for the linear operator is of the form `amult(mctx, vin, vout)`, where `mctx` is the linear operator context and `vin` and `vout` are the pointers to the input and output vectors, respectively. The calling sequence for the preconditioner routine is similar.

To obtain the linear operator context from the iterative context, so as to free the space after a successful run, use `ITGetAmultContext(itctx)`. Similarly, use `ITGetBinvContext(itctx)` for the preconditioner context.

Below we list an artificial sample linear operator. In this case the operator context is simply an integer giving the size of the linear system.

```

subroutine amult(n,vin,vout)
integer n
double precision vin(n), vout(n)
vout(1) = 2.0*vin(1) - vin(2)
do 10 i=2,n-1
    vout(i) = 2.0*vin(i) - vin(i-1) - vin(i+1)
10 continue
vout(n) = 2.0*vin(n) - vin(n-1)
end

```

In some applications it may be more computationally effective to calculate the effect of BA (AB for right preconditioning) as a single operation, rather than first applying the operator and then the preconditioner. Several of the methods, including GMRES, can take advantage of this. KSP allows the user to, optionally, provide a single function that will perform the combined operation (the `amult` and `binv` must also be provided). This is done with the command `ITSetMatop(itctx,matop)`. The calling sequence for the `matop` function must be `matop(amultP,binvP,vin,vout)`. You, as the user, are responsible for providing the correct `matop` depending on left or right preconditioning.

Finally, for the methods that require the action of the transposes of the linear operator and preconditioner, two additional user-provided functions are needed: `tamult` and `tbinv`. These must be set with `ITSetAmultTranspose(itctx, tamult)` and `ITSetBinvTranspose(itctx, tbinv)`. Note that these routines do not get their own private context; those are shared with

the operator and preconditioner. In addition, if you wish to provide a transpose of the matop routine, you must set this with `ITSetMatopTranspose(itctx,tmatop)`.

2.3 Vector Operations

The PETSc package comes with two main default vector implementations: serial double-precision vectors and parallel double-precision vectors. These should support most users needs. We first discuss using these two classes and then explain how new, specialized vector classes may be written.

The vector operations are based around a vector context. Each different type of vector must have a different vector context. A vector context is simply a data structure that contains pointers to all the functions that act on that particular vector type and any private data used by these functions. So, for instance, for the standard serial vectors, the private data is simply an integer indicating the length of the vector. For the standard parallel double-precision vectors, the private data consists of an integer indicating the length of the local of the local component of the vector and a structure that contains information about which processors share the vector.

All of the vector operations should be accessed through the macros in the file `vectors/vector.h`. These are listed in Table 2.1. By using these macros, your code will run with any new vector type introduced, without change. The implementation of these macros is straightforward; for instance, the dot product is defined by

```
#define VDOT(vp,x,y,val) (*(vp)->dot)((vp)->vecP,x,y,val)
```

An empty vector context is created with the command `VECreate()`. You are then responsible for filling the vector context with the appropriate function pointers and private vector context. Use the function `DVSetDefaultFunctions(vectx,n)` to set the default serial double-precision vectors with length `n`. A convenience routine `DVCreate(n)` will create and fill a vector context for the standard serial vectors. The routine `VEDestroy(vectx)` will free the space used by a vector context created with `VECreate()` or `DVCreate()`. The function `DVBSetDefaultFunctions(vectx,n)` is the same as `DVSetDefaultFunctions()` except that it uses the Level 1 BLAS routines if they are available. On some machines this may improve the performance of these operations.

To construct a default parallel vector context, you must first decide which processors are to share the distributed vector and must then construct a processor set, `ProcSet`, using Chameleon [2]. To use all active processors, you merely need to use `PIAllProcs` for the processor set. To create the default parallel vector context, use the convenience function `DVPCreate(n,pset)`, where `n` is length of the local piece of the vector and `pset` is the processor set, for instance, `PIAllProcs`.

2.3.1 Fortran-defined Vectors

When vectors are contiguous in memory (e.g., declared as double precision `x(100)`, `b(100)`), they may be used with the standard serial default vector primitives.

Note that the KSP code will still manage the dynamic creation and destruction of any necessary work vectors. The default serial vectors were designed to be usable from Fortran. The default parallel vectors, those with names `DVP*`, are more complicated and cannot be declared directly in the Fortran code. You can still use the create and destroy routines to

Table 2.1: Vector Macros

<code>void *x = VCREATE(vp)</code>	returns pointer to new vector
<code>VDESTROY(vp,x)</code>	frees space used by vector
<code>void **v = VGETVECS(vp,m)</code>	returns pointer to array of vector pointers
<code>VFREEVECS(vp,vv,m)</code>	frees space used by vectors
<code>VDOT(vp,x,y,result)</code>	$\text{result} \leftarrow x \cdot \bar{y}$
<code>VNORM(vp,x,result)</code>	$\text{result} \leftarrow \ x\ _2$
<code>VMAX(vp,x,result,n)</code>	$\text{result} \leftarrow \ x\ _\infty$
<code>VTDOT(vp,x,y,result)</code>	$\text{result} \leftarrow x \cdot y$
<code>VSCALE(vp,alpha,x)</code>	$x \leftarrow \alpha * x$
<code>VCOPY(vp,x,y)</code>	$y \leftarrow x$
<code>VSET(vp,alpha,x)</code>	$x_i \leftarrow \alpha, \text{ for } i = 0, \dots, ni - 1$
<code>VAXPY(vp,alpha,x,y)</code>	$y \leftarrow x + \alpha * y$
<code>VAYPX(vp,alpha,x,y)</code>	$y \leftarrow y + \alpha * x$
<code>VSWAP(vp,x,y)</code>	$x \leftarrow y; y \leftarrow x$
<code>VWAXPY(vp,alpha,x,y,w)</code>	$w \leftarrow x + \alpha * y$
<code>VPMULT(vp,x,y,w)</code>	$w = x_i * y_i, \text{ for } i = 0, \dots, ni - 1$
<code>VPDIV(vp,x,y,w)</code>	$w = x_i / y_i, \text{ for } i = 0, \dots, ni - 1$
<code>VSCATTER(vp,x,ix,ni,y)</code>	$y[ix[i]] = x[i], \text{ for } i = 0, \dots, ni - 1$
<code>VSCATTERADD(vp,x,ix,y)</code>	$y[ix[i]] += x[i], \text{ for } i = 0, \dots, ni - 1$
<code>VGATHER(vp,x,ix,ni,y)</code>	$y[i] = x[ix[i]], \text{ for } i = 0, \dots, ni - 1$

create parallel vectors from within a Fortran program. These vectors may then be passed in Fortran subroutines as arrays, and the part local to that processor may be accessed directly.

2.3.2 Writing a New Vector Class

Before writing a new vector class, one of the first decisions you must make is what information will be needed by the vector primitives and should be stored in the private vector context. For instance, if you are implementing an out-of-core vector class, the private vector context may contain file names, file pointers, disk block sizes, etc. All of this information should then be encapsulated in a C structure. You then need to write all the vector primitives; a good starting point may be the file `vectors/dvec.c`. Once the primitives have been written, you should provide a routine, modeled on `DVSetDefaultFunctions()`, which fills an empty vector context with the appropriate vector primitives and the private vector context. Finally, a convenience function, like `DVCreate()`, should be provided.

To demonstrate the simplicity of the vector interface, we list several of the default vector routines. `DVPSetDefaultFunctions()` takes three arguments: an empty vector context, an integer denoting the length of the local part of the vector, and a processor set. It fills in the vector context for our standard parallel vectors.

```
void DVPSetDefaultFunctions( vopP, n, pset )
VECctx *vopP;
int      n;
ProcSet *pset;
```



```

{
    DVSetDefaultFunctions( vopP, n );
    VEDestroyDefaultUsrCntx( vopP->vecP );
    vopP->vecP = VEPCreateDefaultUsrCntx(n, pset);
    vopP->dot   = DVPdot;
    vopP->norm  = DVPnorm;
    vopP->max   = DVPmax;
    vopP->tdot  = DVPdot;
}

```

Note that we use the standard default serial vector primitives for most operations. Actually, even the vector reduction operations are based on the serial version. For instance, the parallel dot product is calculated by

```

void DVPdot( N, x, y, z )
VEPDefaultUsrCntx *N;
register double *x, *y, *z;
{
    double sum, work;
    DVdot( (VEPDefaultUsrCntx*)N, x, y, &sum );
    PIdsum( &sum, 1, &work, N->pset );
    *z = sum;
}

```

The macro PIdsum is the parallel global sum across a processor set in Chameleon. The routine DVdot() is the standard serial dot product primitive.

Chapter 3

Optional Parameters

The KSP package allows for the use of many optional arguments. To simplify their usage, each argument has associated with it a function or macro that is used to set that argument. Since different Krylov space methods may have different options (for example, GMRES has the number of search directions), if the argument is inappropriate for a particular method, it is simply ignored. In this way you can change methods without changing your code.

3.1 Common Parameters

To set the maximum number of iterations to be used by the Krylov space method, use the command `ITSetIterations`. For instance, to allow at most ten iterations, use `ITSetIterations(itctx,10)`. By default the maximum number of iterations is 10,000.

The simplest convergence tests for Krylov space methods is to measure the discrete two norm of the residual. By default the KSP package uses either a relative decrease in the two norm of the residual or an absolute tolerance. The commands `ITSetRelativeTolerance` and `ITSetAbsoluteTolerance` determine these. By default, the relative tolerance decrease is 10^{-5} and the absolute tolerance is 10^{-50} . The example below will force the Krylov space method to stop if the value of the discrete two norm of the residual is less than 10^{-8} or the decrease from the initial residual is 10^{-10} .

```
ITSetRelativeTolerance(itctx,1.e-10);  
ITSetAbsoluteTolerance(itctx,1.e-8);
```

The convergence test may be applied to either the true residual or, if a preconditioner is used, to the preconditioned residual (the residual of the preconditioned problem). By default it is applied to the true residual where possible. To use the preconditioned residual, call `ITSetUsePreconditionedResidual` before calling `ITSolve`. In addition, to detect the divergence of the iterative method, if the two-norm of the residual grows more than 10,000 times its initial value, the iteration is stopped. You may provide more sophisticated convergence tests using the routine `ITSetConvergenceTest` discussed below.

In several of the methods, for instance Chebychev, the algorithm does not need to calculate the residual at each iteration. If it is not required for your convergence test, you may avoid calculating it, by calling the routine `ITSetDoNotCalculateResidual`. By default, all methods calculate the two norm of the residual or some approximation to it at each iteration.

In some applications you may wish to keep a record of the two-norms of the residuals for each iteration. This is done with `ITSetResidualHistory`. The `ITSetResidualHistory` macro takes three arguments: the iterative context, a pointer to an array of doubles in which the residual norms will be placed, and an integer indicating the length of the residual history buffer.

Most Krylov space methods support the use of either left or right preconditioning. That is, the preconditioner operator is applied to either the left or right side of the linear operator. By default, KSP uses left preconditioning. To force the use of right preconditioning, use the command `ITSetRightPreconditioner`. At the present time not all the codes in the KSP package allow for right preconditioning. To determine from an iterative context whether left or right preconditioning is to be used, use the routine `ITGetPreconditioningSide(itctx)`, which returns 1 for right preconditioning and 0 for left.

If the Krylov space method knows that the initial guess is zero, it may save an application of the linear operator in calculating the initial step. To inform the iterative context that your initial guess is zero, use the macro `ITSetInitialGuessZero`. Note that this will not set your initial guess to zero. You have to make sure that your code has done this; otherwise, incorrect answers may result.

Several of the Krylov space methods may, as part of their iterative procedure, calculate estimates of the extreme eigenvalues of the preconditioned operator by using the Lanczos or Arnoldi process. In order to allow this estimation to occur, use the `ITSetCalculateEigenvalues` macro. At the moment this is supported only by the conjugate gradient code. To obtain the eigenvalue estimates after the solution process is complete, use the `ITCGGetEigenvalues` routine.

Once an iterative context is defined, the type of Krylov space method it is using can be determined by `ITGetMethodFromContext(itctx)`, which returns the `ITMETHOD`, for instance, `ITGMRES`.

3.2 Parameters for Particular Methods

Several methods have parameters appropriate only for that particular method. To allow for their flexible use, they are simply ignored in all other cases.

Richardson's method will often diverge unless a suitable damping factor is applied. Often this may not be known a priori; but when it is, it may be set with the command `ITRichardsonSetScale`. The default value is 1.0 (no damping).

Chebyshev's method requires good estimates for the extreme eigenvalues of the preconditioned operator in order to obtain any reasonable convergence. If these are known, they may be set with `ITChebyshevSetEigenvalues`. The default values are 0.01 and 100.

When the number of iterations required for convergence gets large, the GMRES method will require more and more memory and floating-point operations, because it keeps information on all the previous search directions. Often, restarted GMRES is used; thus, after a fixed number of iterations is run, the iteration is restarted using the latest approximate solution as the initial guess. The number of search directions used before a restart may be set with the command `ITGMRESSetDirections`. By default the restart is 5.

3.3 Functions

Not only can you set simple variable arguments in the KSP package, you may also change some of the functions used, without changing a line of the original source code.

The two functions of most interest are the function to monitor the iterations and the function to determine whether the method has converged.

By default the Krylov space methods print nothing during the iteration process. The macro `ITSetMonitor` sets a function to be called at each iteration to monitor convergence. The user-provided monitor function must have the following call sequence:

```
void UserMonitor(itctx,n,rnorm)
ITCctx *itctx;
int    n;
double rnorm;
```

The first argument is the usual iterative context. The second argument is a count of the number of iterations of the Krylov space method that have occurred so far. The final argument is the estimate of the two-norm of the residual.

In order to allow the monitor routine to use external data, for instance to plot the residual or error using X windows, the iterative context has a slot reserved for storing a pointer to this information, the monitor context. The pointer to this information may be obtained inside the user-provided monitor routine by using the `ITGetMonitorContext` routine. For example, `mydata = (MyData *) ITGetMonitorContext(itctx)` will allow access to the structure `mydata`. The command `ITSetMonitor(itctx, UserMonitor, mydata)` will now force the Krylov space method using the iterative context, `itctx`, to use your private monitoring routine, with a monitor context pointed to by `mydata`.

KSP comes with three simple monitoring routines: `ITDefaultMonitor`, which simply prints out the residual at each iteration; `ITCGDefaultMonitor`, which prints out the residual and the estimated extreme eigenvalues obtained via the Lanczo process; and `ITLineGraphMonitor`, which displays the residual norm in a window under X11. If no monitor is selected, there is no action.

The example below demonstrates the use of `ITLineGraphMonitor`.

```
XBLineGraph *lg;
char          *host;
lg = ITLineGraphMonitorCreate("Residual",
                               host,0,0,400,400);
ITSetMonitor(itctx,ITLineGraphMonitor,lg);
ITSolve(itctx);
/* set a new right hand side */
ITLineGraphMonitorReset(lg);
ITSolve(itctx);
ITLineGraphMonitorDestroy(lg);
```

This will open a window on the X server `host` of size 400 by 400 and title it "Residual." The command `ITLineGraphMonitorReset` resets the line graph for the next iteration. Once the iteration is complete, the space used by the line graph should be freed with the command `ITLineGraphMonitorDestroy`.

The second major user-replaceable routine is the convergence tester. The (optional) user-provided routine has the calling sequence

```
int UserConverged(itctx,n,rnorm)
ITCctx *itctx;
```

Table 3.1: Methods and Their Names

Method	Name
ITRICHARDSON	richardson
ITCHEBYCHEV	chebychev
ITCG	cg
ITGMRES	gmres
ITTCQMR	tcqmr
ITBCGS	bcbgs
ITCGS	cgs
ITTFQMR	tfqmr

```
int    n;
double rnorm;
```

This routine should return 1 to indicate convergence, -1 to indicate divergence, or 0 to indicate that another iteration should be taken.

The convergence routine is set by using `ITSetConvergenceTest( itctx, UserConverge, cctx )`, where `cctx` is the convergence context, which, similarly to the monitor context, may be obtained by `ITGetConvergenceContext(itctx)`.

3.4 Command Line Arguments

It is often convenient to be able to set the Krylov space method from the command line. KSP provides a simple mechanism to achieve this. For instance,

```
ITMETHOD method = ITGMRES;
ITGetMethod(argc,argv,1,(char *) 0,&method);
itctx = ITCreate(method);
```

will extract out from the command line arguments of the form `-itmethod methodname` and set `method` to the appropriate value. In this example, if no command line arguments are of the correct type, GMRES is used by default. If the third argument, here `(char *) 0`, is replaced with a character string; that string is used in place of `-itmethod`. The Krylov methods and their names are shown in Table 3.1. Note that the names are all lowercase.

Chapter 4

Adding New Methods

You can easily add new methods to the KSP package without changing a single line of code in KSP. You simply call a routine to insert the new methods into a list of known methods. These methods are then known to the `ITCreate` routine as if it were part of the core package. Thus, with the KSP package you can use new developments in algorithms and implementations without changing your application program; you need only relink your code. No longer must you choose between using older algorithms or making significant changes to your code. If you do not plan to add a new method to KSP, you can skip the rest of this chapter.

Iterative accelerators are registered by using the routine `ITRegister`. For instance, `ITRegister( ITCG, "cg", ITCGCreate )` registers the routine `ITCGCreate` with the name "cg" and the integer name `ITCG`. Thus, a call to `ITCreate` with an argument of `ITCG` will create an iterative context appropriate for the conjugate gradient method. This is done by having `ITCreate` simply call the routine that has been registered under that name, in this case `ITCGCreate()`.

The best way to code a new Krylov space method is to choose a simple method, say CG (conjugate gradient), and copy it into a new directory. Your additions to the KSP are welcome and will be added to the public distribution with your permission.

For each Krylov space implementation, four major functions are needed, each associated with the four states: `ITCreate`, `ITSetup`, `ITSolve`, and `ITDestroy`. To maintain uniformity in the codes, we call these `IT<method>Create`, `IT<method>Setup`, `IT<method>Solve`, and `IT<method>Destroy`. The iterative context contains many fields. Three of them hold the functions `IT<method>Setup`, `IT<method>Solve`, and `IT<method>Destroy`. These, along with the other fields, are set by `IT<method>Create`. The functions `ITSetup()`, `ITSolve()`, and `ITDestroy()` are simply macros that call the appropriate function stored in the iterative context.

The rest of this chapter presents by example how a method may be added to KSP. This code is in file `'tools.core/iter/cgs/cgs.c'`.

4.1 Solve

The solution routine is central and provides the information needed to write the other routines.

This version of the `ITCGSSolve` routine has been annotated to make it easier to read. Specific features are discussed below.

```
int  ITCGSSolve(itP)
ITCntx *itP;
```

```

{
  int      i = 0, maxit, res, pres, hist_len, cerr;
  double   rho, rhoold, a, s, b, *history, dp;
  void     *X,*B,*V,*P,*R,*RP,*T,*Q,*U, *BINVF, *AUQ;

  /* Unpack frequently used values into local variables */
  res      = itP->calc_res;
  pres     = itP->use_pres;
  maxit    = itP->max_it;
  history   = itP->residual_history;
  hist_len = itP->res_hist_size;

  /* Get the solution and right-hand-side vectors */
  X        = itP->vec_sol;
  B        = itP->vec_rhs;

  /* Work space is allocated by the setup routine; we access
     it here. This implementation makes little effort to re-use
     temporary space. */
  R        = itP->work[0];
  RP       = itP->work[1];
  V        = itP->work[2];
  T        = itP->work[3];
  Q        = itP->work[4];
  P        = itP->work[5];
  BINVF    = itP->work[6];
  U        = itP->work[7];
  AUQ      = V;

  /* Compute initial preconditioned residual. This routine
     handles the application of the preconditioner */
  ITResidual(itP,X,V,T, R, BINVF, B );

  /* Test for nothing to do (residual norm already 0) */
  NORM(R,&dp);
  if (CONVERGED(dp,0)) return RCONV(0);
  MONITOR(dp,0);
  if (history) history[0] = dp;

  /* Make the initial RP == R */
  COPY(R,RP);

  /* Set the initial conditions */
  DOT(RP,R,&rhoold);
  COPY(R,U);
  COPY(R,P);
  MATOP(P,V,T);

  for (i=0; i<maxit; i++) {
    DOT(RP,V,&s);          /* s <- rp' v          */
    a = rhoold / s;       /* a <- rho / s        */
    DWAXPY(-a,V,U,Q);     /* q <- u - a v        */
    DWAXPY(1.0,U,Q,T);    /* t <- u + q          */
  }

```

```

DAXPY(a,T,X);          /* x <- x + a (u + q) */
MATOP(T,AUQ,U);
DAXPY(-a,AUQ,R);       /* r <- r - a K (u + q) */
NORM(R,&dp);

/* Remember residual history if possible */
if (history && hist_len > i + 1) history[i+1] = dp;
/* MONITOR calls the user monitor, if defined */
MONITOR(dp,i+1);
if (CONVERGED(dp,i+1)) break;

DOT(RP,R,&rho);         /* newrho <- rp' r */
b = rho / rhoold;       /* b <- rho / rhoold */
DWAXPY(b,Q,R,U);        /* u <- r + b q */
DAXPY(b,P,Q);
DWAXPY(b,Q,U,P);        /* p <- u + b(q + b p) */
MATOP(P,V,Q);           /* v <- K p */
rhoold = rho;
}
if (i == maxit) i--;
if (history) itP->res_act_size =
    (hist_len < i + 1) ? hist_len : i + 1;

/* This routine unwinds the preconditioner, if necessary */
ITUnwindPre( itP, X, T );
return RCONV(i+1);
}

```

4.1.1 Work Arrays

Work arrays are allocated by the setup routine and placed into an array of pointers in `itP->work`.

4.1.2 Vector Operations

Vector operations (such as DOT) are defined in the file `'tools.core/iter/itpriv.h'`. These are fairly straightforward, and most match the vector operations in Table 2.1. There are also macros to call the user monitor routine (`MONITOR( residual-norm, iteration )`), to check for convergence (`CONVERGED( residual-norm, iteration )`), and to compute the product of the matrix and preconditioner with a vector (`MATOP( VIN, VOUT, VTEMP )`).

4.2 Create

The create routine is very simple. It must allocate the iterative context (`NEW(a)` is a PETSc macro for `(a *)MALLOC(sizeof(a))`). The helper routine `ITSetDefaults` initializes the fields in the iterative context to the common defaults. The routines to perform the setup and solve actions are set to be the routines defined for CGS. Both the adjustwork routine and the closedown (destroy) routine use helper defaults.

```
ITCntx *ITCGSCreate()
```



```

{
ITCntx    *itP;

itP = NEW(ITCntx);  CHKPTRV(itP,0);
ITSetDefaults( itP );
itP->MethodPrivate = (void *) 0;

itP->method          = ITCGS;
itP->right_inv       = 0;
itP->calc_res        = 1;

itP->setup           = ITCGSSetUp;
itP->solver          = ITCGSSolve;
itP->adjustwork      = ITDefaultAdjustWork;
itP->closedown       = ITDefaultDestroy;
return(itP);
}

```

4.3 Setup

The setup routine is quite simple. It checks that the parameters are valid (using the helper routine `ITCheckDef`) and allocates the number of required temporary vectors using `ITDefaultGetWork`. In this case, eight work vectors are allocated. Note that `ITDefaultGetWork` uses the vector allocation routines defined for the vector type provided; thus, this space allocation is appropriate for any kind of vector.

```

void ITCGSSetUp(itP)
ITCntx *itP;
{
if (itP->method != ITCGS) {
    SETERRC(1,"Attempt to use CGS Setup on wrong context");
    return;}

/* check user parameters and functions */
if (ITCheckDef( itP )) return;

/* get work vectors from user code */
ITDefaultGetWork( itP, 8 );
}

```

4.4 Destroy

Because CGS requires only simple work vectors, the destroy routine (that recovers workspace) uses the helper routine `ITDefaultDestroy`. This is set in `ITCGSSetup`.

Chapter 5

Debugging and Optimization

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

One tool that can help in debugging a KSP code is the iterative monitor (set with `ITSetMonitor`), 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 `ITLineGraphMonitor`, which graphs the norm of the residual and could easily be modified to graph the norm of the error, etc.

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/libsgpg'` and the `-pg` compiler switch (or `BOPT=0pg`, 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.

5.1 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 (`'libs0/tools.a'`) are often built without the ability to generate these tracebacks (or even detect many errors).

5.2 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 hardware.

More detailed information may be gathered by using the profiling library ('tools.core/libs/libsOpg/\$ARCH/tools.a') and the gprof utility (available on most Unix systems).

For coarser-grained information, the routine `ITGetWorkCounts` returns the number of matrix and vector operations.

5.3 Troubleshooting

If you have trouble, check Table 5.1. If you run into something that is not in this table, please let us know.

Table 5.1: Troubleshooting

Problem	Possible Cause	Solution
Residual small but error large	1) operator ill-conditioned	a) use smaller tolerance
	2) preconditioner ill-conditioned	b) use different operator c) use right rather than left preconditioning d) change preconditioners
	3) Convergence tolerance not tight enough	e) use <code>ITSetAbsoluteTolerance</code> and <code>ITSetRelativeTolerance</code> to set smaller values f) use <code>ITSetConvergenceTest</code> to use your own convergence test
Diverges	1) using CG with nonsymmetric matrix	a) use different method
	2) preconditioner ineffective	b) fix preconditioner
	3) preconditioner code buggy	
	4) GMRES needs more directions	c) use <code>ITGMRESSetDirections</code> or change methods
Incorrect answer	1) preconditioner code buggy	fix the code
	2) operator code buggy	
Residual monitor ignored	called before <code>ITSetup</code>	move call after <code>ITSetup</code>

Chapter 6

Hints on Choosing Methods

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

- For symmetric, positive definite matrices one should probably always use the conjugate gradient method (ITCG).
- For nonsymmetric problems, if the number of iterations required for convergence is reasonably small, then GMRES (ITGMRES) may be best.
- If the number of iterations is large, we recommend that you find better preconditioners for your problem. A place to start would be trying some experimental runs using the SLES package or Parallel SLES package.
- If the number of iterations required is still large you will probably wish to use the BiCG-Stab method (ITBCGSTAG) or the transpose free QMR method (ITTFQMR).
- If memory space is at a premium, BiCG-Stab is probably a better choice than GMRES.

It is important to remember that switching between Krylov space methods is trivial, so you should try all the methods and find out which method is most suited to your problem.

Note that Krylov space methods for nonsymmetric problems have pathological cases; see Nachtigal, Reddy, and Trefethen [5]. The flexibility of KSP in allowing many different Krylov methods to be used is very important when solving real (as opposed to model) problems.

Chapter 7

Summary of Routines

This chapter contains a brief summary of the routines in this manual. These are divided into common routines, method-specific routines, and routines for writing and adding methods.

All of these routines require the includes

```
#include "tools.h"
#include "iter/itall.h"
```

7.1 Common Routines

<code>void ITCGDefaultMonitor(itP, n, rnorm)</code> <code>ITCntx *itP;</code> <code>int n;</code> <code>double rnorm;</code>	The default iterative monitor routine for CG; it prints the two-norm of the true residual and estimation from Lanczo of the extreme eigenvalues of the preconditioned problem at each iteration.
<code>void ITClearWorkCounts(itP)</code> <code>ITCntx *itP;</code>	Clears the work counts that are maintained for the iterative solvers.
<code>ITCntx *ITCreate(itmethod)</code> <code>ITMETHOD itmethod;</code>	Builds ITCntx for a particular solver. ITMETHOD is, for instance, ITCG or ITGMRES.
<code>int ITDefaultConverged(itP, n, rnorm)</code> <code>ITCntx *itP;</code> <code>int n;</code> <code>double rnorm;</code>	Default code to determine convergence in the iterative solvers.
<code>void ITDefaultMonitor(itP, n, rnorm)</code> <code>ITCntx *itP;</code> <code>int n;</code> <code>double rnorm;</code>	Default code to print residual at each iteration in the iterative solvers.
<code>MACRO void ITDestroy(itP)</code> <code>ITCntx *itP;</code>	Destroys ITCntx created with ITCreate().
<code>MACRO void *ITGetAmultContext(itP)</code> <code>ITCntx *itP;</code>	Returns a pointer to the operator context set with ITSetAmult().

MACRO void *ITGetBinvContext(itP) ITCntx *itP;	Returns a pointer to the preconditioner context set with ITSetBinv().
MACRO void *ITGetConvergenceContext(itP) ITCntx *itP;	Gets the convergence context set with ITSetConvergenceTest.
MACRO int ITGetMethodFromContext(itP) ITCntx *itP;	Returns the chosen method type.
void ITGetMethod(Argc, argv, flag, sname, itmethod) int *Argc, flag; char **argv, *sname; ITMETHOD *itmethod;	Sets the selected method, given the argument list.
MACRO void *ITGetMonitorContext(itP) ITCntx *itP;	Gets the context provided by ITSetMonitor.
MACRO int ITGetPreconditionerSide(itP) ITCntx *itP;	Gets preconditioning side.
MACRO void *ITGetRhs(itP) ITCntx *itP;	Gets the right-hand side for the linear system to be solved.
MACRO void *ITGetSolution(itP) ITCntx *itP;	Gets the location of the solution for the linear system to be solved.
void ITGetWorkCounts(itP, matop, amult, binv, vecs, scalars) ITCntx *itP; int *matop, *amult, *binv, *vecs, *scalars;	Gets the counts of the number of operations performed by the iterative routines
MACRO void ITSetAbsoluteTolerance(itP, a) ITCntx *itP; double a;	Sets the convergence tolerance as an absolute decrease in the residual of tol. Use ITSetRelativeTolerance() for relative tolerance.
MACRO void ITSetAmultTranspose(itP, tamult) ITCntx *itP; void (*tamult)();	Sets the function to be used to calculate the transpose of the matrix vector product.
MACRO void ITSetAmult(itP, amult, amultP) ITCntx *itP; void (*amult)(), *amultP;	Sets the function to be used to calculate the matrix vector product. Use ITGetAmultContext() to retrieve the multiply context (say, at the end of the computations).
MACRO void ITSetBinvTranspose(itP, tbinv) ITCntx *itP; void (*tbinv)();	Sets the function to be used to calculate the application of the transpose of the preconditioner on a vector.
MACRO void ITSetBinv(itP, binv, binvP) ITCntx *itP; void (*binv)(), *binvP;	Sets the function to be used to calculate the application of the preconditioner on a vector. Use ITGetBinvContext() to retrieve the preconditioner context (say, to free it at the end of the computations).

<pre>MACRO void ITSetCalculateEigenvalues(itP) ITCntx *itP;</pre>	<p>Sets a flag so that the the method will calculate the extreme eigenvalues via a Lanczo or Arnoldi process as it solves the linear system.</p>
<pre>MACRO void ITSetCalculateResidual(itP) ITCntx *itP;</pre>	<p>Sets a flag so that the two-norm of the residual is calculated at each iteration.</p>
<pre>MACRO void ITSetConvergenceTest(itP, converge, cctx) ITCntx *itP; int (*converge)(); void *cctx;</pre>	<p>Sets the function to be used to determine convergence.</p>
<pre>void ITSetDefaults(itP) ITCntx *itP;</pre>	<p>Sets the most basic defaults for the iterative solver context.</p>
<pre>MACRO void ITSetDoNotCalculateResidual(itP) ITCntx *itP;</pre>	<p>Sets a flag so that the two-norm of the residual is not calculated at each iteration.</p>
<pre>MACRO void ITSetInitialGuessZero(itP) ITCntx *itP;</pre>	<p>Tells the iterative solver that the initial guess is zero; otherwise it assumes it is nonzero. If the initial guess is zero, this saves one matrix multiply in the calculation of the initial residual.</p>
<pre>MACRO void ITSetIterations(itP, its) ITCntx *itP; int its;</pre>	<p>Sets the maximum number of iterations to use.</p>
<pre>MACRO void ITSetMatopTranspose(itP, tmatop) ITCntx *itP; void (*tmatop)();</pre>	<p>Sets the function to be used to calculate the action of the transpose of the Matop.</p>
<pre>MACRO void ITSetMatop(itP, matop) ITCntx *itP; void (*matop)();</pre>	<p>Sets the function to be used to calculate the application of the preconditioner, followed by the application of the matrix multiplier on a vector. For left preconditioner the order is reversed.</p>
<pre>MACRO void ITSetMonitor(itP, monitor, mctx) ITCntx *itP; void (*monitor)(), *mctx;</pre>	<p>Sets the function to be used at every iteration of the iterative solution.</p>
<pre>MACRO void ITSetRelativeTolerance(itP, tol) ITCntx *itP; double tol;</pre>	<p>Sets the convergence tolerance as a relative decrease in the residual of tol.</p>
<pre>MACRO void ITSetResidualHistory(itP, a, na) ITCntx *itP; double *a; int na;</pre>	<p>Sets the array used to hold the residual history.</p>

MACRO void ITSetRhs(itP, x) ITCntx *itP; void *x;	Sets the right-hand side for the linear system to be solved.
MACRO void ITSetRightPreconditioner(itP) ITCntx *itP;	Sets right preconditioning.
MACRO void ITSetSolution(itP, b) ITCntx *itP; void *b;	Sets the location of the solution for the linear system to be solved.
MACRO void ITSetup(itP) ITCntx *itP;	Sets up the internal data structures for the later use of an iterative solver.
MACRO void ITSetUsePreconditionedResidual(itP) ITCntx *itP;	Sets a flag so that the two-norm of the preconditioned residual is used rather than the true residual.
ITCntx *ITSimpleCreate(itmethod, amult, binv, x, b, n) ITMETHOD itmethod; void (*amult)(), (*binv)(); double *x, *b; int n;	Builds ITCntx for a particular solver; simple interface.
MACRO int ITSolve(itP) ITCntx *itP;	Solves linear system; call it after calling ITCreat(), ITSetup(), and ITSet*().
MACRO VECntx *ITVectorContext(itP) ITCntx *itP;	Returns a pointer to the vector context in an iterative context.

7.2 Method-specific Routines

void ITCGGetEigenvalues(itP, n, emax, emin) ITCntx *itP; double *emax, *emin; int n;	After running ITSolve (with ITCG), calling this routine will return the extreme eigenvalues of the preconditioned problem as calculated by Lanczos.
void ITChebychevSetEigenvalues(itP, emax, emin) ITCntx *itP; double emax, emin;	Called after a call to ITCreat(ITCHEBYCHEV), sets the extreme eigenvalues used by the Chebychev method.
void ITGMRESSetDirections(itP, max_k) ITCntx *itP; int max_k;	Sets the number of search directions for GMRES before restart.
void ITRichardsonSetScale(itP, scale) ITCntx *itP; double scale;	Called after ITCreat(ITRICHARDSON), sets the "damping" factor; if this routine is not called, the factor defaults to 1.0.

7.3 Routines for Adding Methods

int ITCheckDef(itP) ITCntx *itP;	Checks the definition of the iterative context quantities necessary for most of the solvers.
void *ITDefaultBuildResidual(itP, t, v) ITCntx *itP; void *t,*v;	Default code to compute the residual.
void *ITDefaultBuildSolution(itP, v) ITCntx *itP; void *v;	Default code to create/move the solution.
void ITRegisterAll()	Registers all the iterative methods in the KSP package.
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 KSP package, given an iterative name (ITMETHOD) and a function pointer.
int ITResidual(itP, vsoln, vt1, vt2, vres, vbinvf, vb) ITCntx *itP; void *vsoln, *vt1, *vt2, *vres, *vbinvf, *vb;	Computes the initial residual without making any assumptions about the solution. Uses the general iterative structure.
int ITUnwindPre(itP, vsoln, vt1) ITCntx *itP; void *vsoln, *vt1;	Unwinds the preconditioning in the solution.

7.4 Vector Routines

This section lists the vector routines. Most users will need only the various SetDefaultFunctions and perhaps VCREATE and VDESTROY.

```
#include "tools.h"
#include "vectors/vector.h"
```

VECntx *DVBCreate(n) int n;	Creates a vector context for BLAS-based double-precision serial vectors.
void DVBSetDefaultFunctions(vopP,n) VECntx *vopP; int n;	Sets vector operations for serial double-precision vectors, using the BLAS where possible. Use DVBCreate() to create and fill the vector context in one step.
VECntx *DVCreate(n) int n;	Creates a vector context for double-precision serial vectors.
VECntx *DVPCreate(n,pset) int n; ProcSet *pset;	Creates a vector context for parallel double-precision serial vectors.

<pre>void DVPSetDefaultFunctions(vopP, n, pset) VECctx *vopP; int n; ProcSet *pset;</pre>	Sets vector operations for parallel double-precision vectors. Use DVPCreate() to create and fill the vector context in one step.
<pre>void DVSetDefaultFunctions(vopP, n) VECctx *vopP; int n;</pre>	Sets vector operations for serial double-precision vectors. The vector context should have been obtained with VECcreate(). Use DVCreate() to create and fill the vector context in one step.
<pre>MACRO void VAXPY(vp,alpha,x,y) VECctx *vp; void *x, *y; Scalar alpha;</pre>	$y \leftarrow \alpha x + y$.
<pre>MACRO void VAYPX(vp,alpha,x,y) VECctx *vp; void *x, *y; Scalar alpha;</pre>	$y \leftarrow x + \alpha y$.
<pre>MACRO void VCOPY(vp,x,y) VECctx *vp; void *x, *y;</pre>	Copies a vector.
<pre>MACRO void *VCREATE(vp) VECctx *vp;</pre>	Creates a vector from a given context. Use VDESTROY() to free the space. Use VGETVECS() to get several vectors.
<pre>MACRO void VDESTROY(vp,v) VECctx *vp; void *v;</pre>	Destroys a vector created with VCREATE().
<pre>MACRO void VDOT(vp,x,y,val) VECctx *vp; void *x, *y, *val;</pre>	Vector dot product.
<pre>VECctx *VECreate()</pre>	Creates an empty vector context.
<pre>void VEDestroy(v) VECctx *v;</pre>	Destroys a vector context.
<pre>MACRO void **VFREEVECS(vp,vv,m) VECctx *vp; void *vv; int m;</pre>	Frees a block of vectors obtained with VGETVECS().
<pre>MACRO void VGATHER(vp,x,ix,ni,y) VECctx *vp; void *x, *y; int ni, *ix;</pre>	Gathers from a vector into a dense array. Use VSCATTER() to scatter into vector.
<pre>MACRO void **VGETVECS(vp,m) VECctx *vp; int m;</pre>	Obtains several vectors.
<pre>MACRO void VMAX(vp,x,p,val) VECctx *vp; void *x, *val; int *p;</pre>	Maximum of vector and its location.

MACRO void VNORM(vp,x,val) VECntx *vp; void *x, *val;	Vector two-norm.
MACRO void VPDIV(vp,x,y,w) VECntx *vp; void *x, *y, *w;	Componentwise division $w = x/y$.
MACRO void VPMULT(vp,x,y,w) VECntx *vp; void *x, *y, *w;	Componentwise multiplication $w = x*y$.
MACRO void VSCALE(vp,alpha,x) VECntx *vp; void *x; Scalar alpha;	Scales a vector.
MACRO void VSCATTERADD(vp,x,ix,ni,y) VECntx *vp; void *x, *y; int ni, *ix;	Scatters and adds from a dense array to a vector. Use VSCATTER() to insert into vector rather than add.
MACRO void VSCATTER(vp,x,ix,ni,y) VECntx *vp; void *x, *y; int ni, *ix;	Scatters from a dense array to a vector. Use VSCATTERADD() to add to vector rather than replace.
MACRO void VSET(vp,alpha,x) VECntx *vp; void *x; Scalar alpha;	Sets all components of a vector to a scalar.
MACRO void VSWAP(vp,x,y) VECntx *vp; void *x, *y;	Swaps x and y.
MACRO void VTDOT(vp,x,y,val) VECntx *vp; void *x, *y, *val;	Non-Hermitian vector dot product, it does NOT use the complex conjugate.
MACRO void VWXPY(vp,alpha,x,y,w) VECntx *vp; void *x, *y, *w; Scalar alpha;	$w \leftarrow \alpha x + y$.

Chapter 8

Examples

In this chapter we present several programs for numerically solving a two-dimensional Poisson problem using matrix-free methods and some simple preconditionings. Both Fortran and C programs are presented.

For simplicity in defining the boundary conditions and matrix operations, we define the problem as including the boundaries. That is, the boundaries are included in the vector and contain the Dirichlet boundary conditions.

Because the code is basically the same as presented in Section 2.1, we emphasize the definitions of the matrix multiply and preconditioning routines. Two different preconditioning routines are provided: one for diagonal preconditioning and one for Gauss-Seidel preconditioning.

8.1 Fortran

The routine `amult` performs the matrix-vector product for the five-point discretization of the two-dimensional Laplacian with Dirichlet boundary conditions, on an m by m mesh.

```
      subroutine amult( m, vin, vout )
      integer m
      double precision vin(m,m), vout(m,m)
      integer i, j
c
      do 10 i=1,m
        vout(i,1) = vin(i,1)
10      vout(i,m) = vin(i,m)
        do 20 j=2,m-1
          vout(1,j) = vin(1,j)
20      vout(m,j) = vin(m,j)
          do 30 j=2,m-1
            do 30 i=2,m-1
              vout(i,j) = 4.0 * vin(i,j) -
+                (vin(i,j+1) + vin(i,j-1) +
+                vin(i-1,j) + vin(i+1,j))
30      continue
      end
```

The routine bdiag is a simple interface for diagonal preconditioning. This routine takes advantage of the fact that the matrix elements are known, and uses a common Fortran trick to "reshape" a one-dimensional array as a two-dimensional one.

```

subroutine bdiag( m, vin, vout )
integer m
double precision vin(m,m), vout(m,m)
integer i, j
c
do 10 i=1,m
    vout(i,1) = vin(i,1)
10    vout(i,m) = vin(i,m)
do 20 j=2,m-1
    vout(1,j) = vin(1,j)
20    vout(m,j) = vin(m,j)
do 30 j=2,m-1
    do 30 i=2,m-1
30        vout(i,j) = vin(i,j) * 0.25d0
end

```

The routine bgs implements a simple Gauss-Seidel preconditioner, using the simple structure of the matrix to do this without storing the matrix elements. As in the case of bdiag, this routine reshapes the vector into a two-dimensional array.

```

subroutine bgs( m, vin, vout )
integer m
double precision vin(m,m), vout(m,m)
integer i, j
c
do 10 i=1, m
10    vout(i,1) = vin(i,1)
do 20 j=2, m-1
    vout(1,j) = vin(1,j)
do 30 i=2, m-1
30        vout(i,j) = 0.25*(vin(i,j) +
+            (vout(i-1,j)+vout(i,j-1)))
20    vout(m,j) = vin(m,j)
do 40 i=1,m
40    vout(i,m) = vin(i,m)
do 50 j=m-1,2,-1
    do 50 i=m-1,2,-1
50        vout(i,j) = 0.25*(vin(i,j) +
+            (vout(i-1,j)+vout(i,j-1)+
+            vout(i+1,j)+vout(i,j+1)))
end

```

Next, we list the routine that sets the right-hand-side vector.

```

subroutine setrhs( m, vin )

```

```

integer m
double precision vin(m,m)
integer i, j
do 10 i=1,m
    vin(i,1) = 1.0
10    vin(i,m) = 1.0
do 20 j=2,m-1
    vin(1,j) = 0.0
20    vin(m,j) = 0.0
do 30 j=2,m-1
    do 30 i=2,m-1
30        vin(i,j) = 0
end

```

And finally, we list the main program. Note that the number of unknowns along a side, *m*, is used as the context for the routines *amult* and *bdiag* and *bgs*. This program may be found in 'tools.core/iter/examples/example4.f'; the makefile in that directory will produce an executable.

```

program main
integer itP, gmres_k, n, m
double precision soln(10000), rhs(10000)
external amult, bdiag, bgs
include '../..//iter/fiter.h'
C
gmres_k = 20
m       = 64
n       = m * m
itP     = ITCreat(ITGMRES)
call ITGMRESSetDirections( itP, gmres_k )
call DVSetDefaultFunctions( ITVectorContext( itP ),
+                             n )
C
C   Set the location of the users right-hand-side and solution
call ITSetRhs( itP, rhs )
call ITSetSolution( itP, soln )
C
call ITSetAmult(itP,amult,m)
c   Choose either bgs or bdiag
c   call ITSetBinv(itP,bgs,m)
call ITSetBinv(itP,bdiag,m)
call ITSetMonitor(itP,ITDefaultMonitor,0)
C
call ITSetUp(itP)

C   set the rhs and the initial solution
call setrhs( m, rhs )
call VSET( ITVectorContext( itP ), 0.0d0, soln )

```

C

```
call ITSolve(itP)
call ITDestroy(itP)
stop
end
```

8.2 C

The C program uses the same Fortran routines to perform the matrix operations as the Fortran version above. It takes advantage of the standard way in which the command-line is passed to C programs to simplify the comparison of the available methods and preconditioners with a single program (see the shell script following this code). This program may be found in file 'tools.core/iter/examples/example5.c'.

```
#include "tools.h"
#include "iter/itall.h"

/* Handle the various Fortran interface cases */
#ifdef FORTRANCAPS
#define amult_ AMULT
#define bdiag_ BDIAG
#define bgs_ BGS
#define setrhs_ SETRHS
#elif !defined(FORTRANUNDERSCORE)
#define amult_ amult
#define bdiag_ bdiag
#define bgs_ bgs
#define setrhs_ setrhs
#elif defined(FORTRANCAPS)
#endif

extern void    amult_(), bdiag_(), bgs_();

main( argc, argv )
int  argc;
char **argv;
{
    ITCntx    *itP;
    ITMETHOD itmethod = ITGMRES;
    int       gmres_k, n, m;
    double    *soln, *rhs;
    void      (*binv)() = bdiag_;

    gmres_k = 20;
    m       = 64;
    SYArgGetInt( &argc, argv, 1, "-n", &m );
    n       = m * m;
```

```

ITGetMethod( &argc, argv, 1, (char *)0, &itmethod );
if (SYArgHasName( &argc, argv, 1, "-gs" )) binv = bgs_;
itP      = ITCreat( itmethod );
ITGMRESSetDirections( itP, gmres_k );
DVSetDefaultFunctions( ITVectorContext( itP ), n );

/* Allocate rhs and solution */
rhs = VCREATE(ITVectorContext( itP ));
soln = VCREATE(ITVectorContext( itP ));

/* Set the users right-hand side and solution */
ITSetRhs( itP, rhs );
ITSetSolution( itP, soln );

ITSetAmult(itP, amult_, &m);
ITSetBinv(itP, binv, &m)

ITSetMonitor(itP, ITDefaultMonitor, (void *)0);

ITSetUp(itP);

/* set the rhs and the initial solution */
setrhs_( &m, rhs );
VSET( ITVectorContext( itP ), 0.0, soln );

ITSolve(itP);
ITDestroy(itP);
}

```

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

```

#!/bin/csh
if ($#argv > 0) then
    set N = $1
else
    set N = 64
endif
foreach itmethod ( richardson chebychev cg \
                    gmres tcqmr bcgs cgs tfqmr )
    foreach precondition ("" -gs)
        example -n $N -itmethod $itmethod $precondition
    end
end

```

Note that the matrix here is not symmetric; using a preconditioner with this matrix may result in divergence for methods that need a symmetric matrix (such as CG).

Bibliography

- [1] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica 1992. Cambridge University Press, 1991.
- [2] William D. Gropp and Barry F. Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.
- [3] William D. Gropp and Barry F. Smith. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. Technical Report MCS-P356-0393, Mathematics and Computer Science Division, Argonne National Laboratory, May 1993.
- [4] William D. Gropp and Barry F. Smith. Simplified Linear Equation Solvers users manual. Technical Report ANL-93/8, Argonne National Laboratory, March 1993.
- [5] Noël M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. How fast are nonsymmetric matrix iterations? Technical Report 90-2, Massachusetts Institute of Technology, March 1990.

Function Index

D

DVBSetDefaultFunctions	13
DVCreate	13, 14
DVPCreate	13
DVPSetDefaultFunctions	10, 14
DVSetDefaultFunctions	10, 13, 14

I

ITCGDefaultMonitor	18
ITCGGetEigenvalues	17
ITChebychevSetEigenvalues	17
ITCreate	9, 20
ITDefaultMonitor	18
ITDestroy	9, 10
ITGetAmultContext	12
ITGetBinvContext	12
ITGetConvergenceContext	19
ITGetMethodFromContext	17
ITGetMonitorContext	18
ITGetPreconditioningSide	17
ITGetRhs	10
ITGetSolution	10
ITGetWorkCounts	25
ITGMRESSetDirections	17
ITLineGraphMonitor	24
ITLineGraphMonitorCreate	18
ITLineGraphMonitorDestroy	18
ITLineGraphMonitorReset	18
ITRegister	20
ITRichardsonSetScale	17
ITSetAbsoluteTolerance	16
ITSetAmult	10
ITSetAmultTranspose	12
ITSetBinv	10
ITSetBinvTranspose	12
ITSetCalculateEigenvalues	17
ITSetConvergenceTest	16, 19
ITSetDoNotCalculateResidual	16
ITSetIterations	16
ITSetMatop	12
ITSetMatopTranspose	13
ITSetMonitor	18, 24
ITSetRelativeTolerance	16

ITSetResidualHistory	17
ITSetRhs	10
ITSetRightPreconditioner	17
ITSetSolution	10
ITSetUp	9, 10
ITSetUsePreconditionedResidual	16
ITSolve	9, 10
ITVectorContext	10

P

PIAllProcs	13
------------	----

V

VAXPY	13
VAYPX	13
VCOPY	13
VCREATE	13
VDESTROY	13
VDOT	13
VECreate	13
VFREEVECS	13
VGATHER	13
VGETVECS	13
VMAX	13
VNORM	13
VPDIV	13
VPMULT	13
VSCALE	13
VSCATTER	13
VSCATTERADD	13
VSET	13
VSWAP	13
VTDOT	13
VWAXPY	13

Distribution for ANL-93/30

Internal:

J. M. Beumer (20)
F. Y. Fradin
W. D. Gropp (5)
G. W. Pieper
B. F. Smith (5)
R. L. Stevens
C. L. Wilkinson
TIS File

External:

DOE-OSTI, for distribution per UC-405 (54)
ANL-E Library (2)
ANL-W Library
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
F. Berman, University of California at LaJolla
B. L. Buzbee, National Center for Atmospheric Research
J. G. Glimm, State University of New York at Stony Brook
M. T. Heath, University of Illinois at Urbana
E. F. Infante, University of Minnesota
K. Kunen, University of Wisconsin at Madison
D. O'Leary, University of Maryland
R. E. O'Malley, Rensselaer Polytechnic Institute
M. H. Schultz, Yale University
J. Cavallini, Department of Energy - Office of Scientific Computing
F. Howes, Department of Energy - Office of Scientific Computing