UNIVERSIDAD POLITECNICA DE VALENCIA

Departamento de Sistemas Informáticos y Computación

DSIC

Technical Report DSIC-II/24/02

SLEPc Users Manual

Scalable Library for Eigenvalue Problem Computations

http://www.grycap.upv.es/slepc

Vicente Hernández
José E. Román
Andrés Tomás
Vicent Vidal

To be used with SLEPc 2.3.3
June, 2007

Abstract

This document describes SLEPc, the *Scalable Library for Eigenvalue Problem Computations*, a software package for the solution of large sparse eigenproblems on parallel computers. It can be used for the solution of problems formulated in either standard or generalized form, as well as other related problems such as the singular value decomposition. SLEPc is a general library in the sense that it covers both Hermitian and non-Hermitian problems, with either real or complex arithmetic.

The emphasis of the software is on methods and techniques appropriate for problems in which the associated matrices are large and sparse, for example, those arising after the discretization of partial differential equations. Therefore, most of the methods offered by the library are projection methods such as Arnoldi, Lanczos or Subspace Iteration. In addition to its own solvers, SLEPc provides transparent access to some external software packages such as ARPACK. These packages are optional and their installation is not required to use SLEPc. See section 5.5 for details. Apart from eigensolvers and SVD solvers, SLEPc also provides built-in support for spectral transformations such as shift-and-invert.

SLEPc is built on top of PETSc, the Portable, Extensible Toolkit for Scientific Computation [Balay *et al.*, 2007]. It can be considered an extension of PETSc providing all the functionality necessary for the solution of eigenvalue problems. This means that PETSc must be previously installed in order to use SLEPc. PETSc users will find SLEPc very easy to use, since it enforces the same programming paradigm. Those readers that are not acquainted with PETSc are highly recommended to familiarize with it before proceeding with SLEPc.

How to get SLEPc

All the information related to SLEPc can be found at the following web site:

http://www.grycap.upv.es/slepc.

The distribution file is available for download at this site. Other information is provided there, such as installation instructions and contact information. Instructions for installing the software can also be found in section 1.2 of this document.

PETSc can be downloaded from http://www.mcs.anl.gov/petsc. PETSc is supported, and information on contacting support can be found at that site.

Additional Documentation

This manual provides a general description of SLEPc. In addition, manual pages for individual routines are included in the distribution file in hypertext format, and are also available on-line at http://www.grycap.upv.es/slepc/document.htm. These manual pages provide hyperlinked indices to the source code and enable easy movement among related topics. Finally, there are also several hands-on exercises available, which are intended to learn the basic concepts easily.

How to Read this Manual

Users that are already familiar with PETSc can read chapter 1 very fast. Section 2.1 provides a brief overview of eigenproblems and the general concepts used by eigensolvers, so it can be skipped by experienced users. Chapters 2, 3 and 4 describe the main SLEPc functionality, and include an advanced usage section that can be skipped at a first reading. Finally, chapter 5 contains less important, additional information.

SLEPc Technical Reports

The information contained in this manual is complemented by a set of Technical Reports, which provide technical details that normal users typically do not need to know but may be useful for experts in order to identify the particular method implemented in SLEPc. These reports are not included in the SLEPc distribution file but can be accessed via the SLEPc web site. A list of available reports is included at the end of the Bibliography.

Acknowledgments

We thank all the PETSc team for their help and support. We also thank Osni Marques and Tony Drummond for helping us raise awareness of SLEPc in the context of the ACTS project.

The initial development of SLEPc was partially funded by the Science and Technology Office of the Valencian Regional Government under grant number CTIDB/2002/54. Additional funding is listed below:

- Directorate of Research and Technology Transfer, Valencian Regional Administration, grant number GV06/091, PI: Jose E. Roman.

Conditions of Use

[1] V. Hernandez, J. E. Roman and V. Vidal (2005), SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems, ACM Trans. Math. Softw. 31(3), 351-362.

Contents

Chapter 1

Getting Started

SLEPc, the *Scalable Library for Eigenvalue Problem Computations*, is a software library for the solution of large sparse eigenvalue problems on parallel computers.

Together with linear systems of equations, eigenvalue problems are a very important class of linear algebra problems. The need for the numerical solution of these problems arises in many situations in science and engineering, in problems associated with stability and vibration analysis in practical applications. These are usually formulated as large sparse eigenproblems.

Computing eigenvalues is essentially more difficult than solving linear systems of equations. This has resulted in a very active research activity in the area of computational methods for eigenvalue problems in the last years, with many remarkable achievements. However, these state-of-the-art methods and algorithms are not easily transferred to the scientific community, and, apart from a few exceptions, most user still rely on simpler, well-established techniques.

The reasons for this situation are diverse. First, new methods are increasingly complex and difficult to implement and therefore robust implementations must be provided by computational specialists, for example as software libraries. The development of such libraries requires to invest a lot of effort but sometimes they do not reach normal users due to a lack of awareness.

In the case of eigenproblems, using libraries is not straightforward. It is usually recommended that the user understands how the underlying algorithm works and typically the problem is successfully solved only after several cycles of testing and parameter tuning. Methods are often specific for a certain class of eigenproblems (e.g. complex symmetric) and this leads to an explosion of available algorithms from which the user has to choose. Not all these algorithms are available in the form of software libraries, even less frequently with parallel capabilities.

Another difficulty resides in how to represent the operator matrix. Unlike in dense methods, there is no widely accepted standard for basic sparse operations in the spirit of BLAS. This is due to the fact that sparse storage is more complicated, admitting of more variation, and therefore

less standardized. For this reason, sparse libraries have an added level of complexity. This holds even more so in the case of parallel distributed-memory programming, where the data of the problem have to be distributed across the available processors.

The first implementations of algorithms for sparse matrices required a prescribed storage format for the sparse matrix, which is an obvious limitation. An alternative way of matrix representation is by means of a user-provided subroutine for the matrix-vector product. Apart from being format-independent this solution allows to solve problems in which the matrix is not available explicitly. The drawback is the restriction to a fixed-prototype subroutine.

A better solution for the matrix representation problem is the well-known reverse communication interface, a technique that allows to implement iterative methods disregarding the implementation details of various operations. Whenever the iterative method subroutine needs the results of one of the operations, it returns control to the user's subroutine that called it. The user's subroutine then invokes the module that performs the operation. The iterative method subroutine is invoked again with the results of the operation.

Several libraries with any of the interface schemes mentioned above are publicly available. For a survey of such software see the SLEPc Technical Report [STR-6], "A Survey of Software for Sparse Eigenvalue Problems", and references therein. Some of the most recent libraries are even prepared for parallel execution (some of them can be used from within SLEPc, see section 5.5). However, they still lack some flexibility or require too much programming effort from the user.

A further obstacle appears when these libraries have to be used in the context of large software projects carried out by inter-disciplinary teams. In this scenery, libraries must be able to interoperate with already existing software and with other libraries. In order to cope with the complexity associated with such projects, libraries must be designed carefully in order to overcome hurdles such as different storage formats or programming languages. In the case of parallel software, care must be taken also to achieve portability to a wide range of platforms with good performance and still retain flexibility and usability.

1.1 SLEPc and PETSc

The SLEPc library is an attempt to provide a solution to the situation described in the previous paragraphs. It is intended to be a general library for the solution of eigenvalue problems that arise in different contexts, covering standard and generalized problems, both Hermitian and non-Hermitian, with either real or complex arithmetic. Issues such as usability, portability, efficiency and interoperability are addressed, and special emphasis is put on flexibility, providing data-structure neutral implementations and multitude of run-time options. SLEPc offers a growing number of eigensolvers as well as interfaces to integrate well-established eigenvalue packages such as ARPACK. In addition, a specific module for SVD computation is included as well.

SLEPc is based on PETSc, the Portable, Extensible Toolkit for Scientific Computation [Balay *et al.*, 2007], and, therefore, a large percentage of the software complexity is avoided since many PETSc developments are leveraged, including matrix storage formats and linear solvers, to name

a few. SLEPc focuses on high level features for eigenproblems, structured around a few object types as described below.

PETSc uses modern programming paradigms to ease the development of large-scale scientific application codes in Fortran, C, and C++ and provides a powerful set of tools for the numerical solution of partial differential equations and related problems on high-performance computers. Its approach is to encapsulate mathematical algorithms using object-oriented programming techniques, which allow to manage the complexity of efficient numerical message-passing codes. All the PETSc software is free and used around the world in a variety of application areas.

The design philosophy is not to try to completely conceal parallelism from the application programmer. Rather, the user initiates a combination of sequential and parallel phases of computations, but the library handles the detailed message passing required during the coordination of computations. Some of the design principles are described in [Balay *et al.*, 1997].

PETSc is built around a variety of data structures and algorithmic objects. The application programmer works directly with these objects rather than concentrating on the underlying data structures. Each component manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on the objects. The three basic abstract data objects are index sets, vectors and matrices. Built on top of this foundation are various classes of solver objects, which encapsulate virtually all information regarding the solution procedure for a particular class of problems, including the local state and various options such as convergence tolerances, etc.

SLEPc can be considered an extension of PETSc providing all the functionality necessary for the solution of eigenvalue problems. Figure 1.1 shows a diagram of all the different objects included in PETSc (on the left) and those added by SLEPc (on the right). PETSc is a prerequisite for SLEPc and users should be familiar with basic concepts such as vectors and matrices in order to use SLEPc. Therefore, together with this manual we recommend to use the PETSc Users Manual [Balay *et al.*, 2007].

Each of these components consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. Both PETSc and SLEPc are written in C, which lacks direct support for object-oriented programming. However, it is still possible to take advantage of the three basic principles of object-oriented programming to manage the complexity of such a large package. PETSc uses data encapsulation in both vector and matrix data objects. Application code accesses data through function calls. Also, all the operations are supported through polymorphism. The user calls a generic interface routine, which then selects the underlying routine that handles the particular data structure. Finally, PETSc also uses inheritance in its design. All the objects are derived from an abstract base object. From this fundamental object, an abstract base object is defined for each PETSc object (`Mat`, `Vec` and so on), which in turn has a variety of instantiations that, for example, implement different matrix storage formats.

PETSc/SLEPc provide clean and effective codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or eigensolvers). Hence, PETSc, together with SLEPc, provide a rich environment

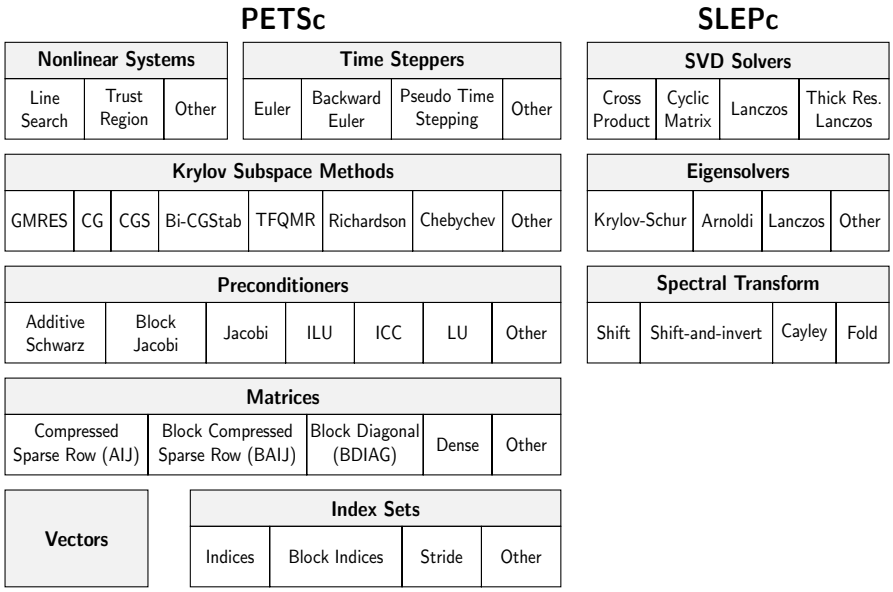

Figure 1.1: Numerical components of PETSc and SLEPc.

for modeling scientific applications as well as for rapid algorithm design and prototyping.

Options can be specified by means of calls to subroutines in the source code and also as command-line arguments. Runtime options allow the user to test different tolerances, for example, without having to recompile the program. Also, since PETSc provides a uniform interface to all of its linear solvers —the Conjugate Gradient, GMRES, etc.— and a large family of preconditioners —block Jacobi, overlapping additive Schwarz, etc.—, one can compare several combinations of method and preconditioner by simply specifying them at execution time. SLEPc shares this good property.

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

1.2 Installation

This section gives an overview of the installation procedure. For full up-to-date installation instructions see http://www.grycap.upv.es/slepc/install.htm.

Previously to the installation of SLEPc, the system must have an appropriate version of PETSc installed. Table 1.1 shows a list of SLEPc versions and their corresponding PETSc versions. SLEPc

SLEPc version	PETSc versions	Major	Release date
2.1.0	2.1.0	⋆	Not released
2.1.1	2.1.1, 2.1.2, 2.1.3		Dec 2002
2.1.5	2.1.5, 2.1.6		May 2003
2.2.0	2.2.0	⋆	Apr 2004
2.2.1	2.2.1	⋆	Aug 2004
2.3.0	2.3.0	⋆	Jun 2005
2.3.1	2.3.1		Mar 2006
2.3.2	2.3.1, 2.3.2	⋆	Oct 2006
2.3.3	2.3.3	⋆	Jun 2007

Table 1.1: Correspondence between SLEPc and PETSc releases.

versions marked as major releases are those which incorporate some new functionality. The rest are just adaptations required for a new PETSc release and may also include bug fixes.

The installation process for SLEPc is very similar to PETSc, with two stages: configuration and compilation. SLEPc configuration is much simpler because most of the configuration information is taken from PETSc, including compiler options and scalar type (real or complex). Several configurations can coexist in the same directory tree, being referred by different values of `PETSC_ARCH`, so that one can, for instance, have a SLEPc compiled with real scalars and another one with complex scalars.

The main steps for the installation are described next. Note that prior to this steps, optional packages must have been installed. If any of these packages is installed afterwards, reconfiguration and recompilation is necessary. Refer to http://www.grycap.upv.es/slepc/install.htm or to section 5.5 for details about installation of some of these packages.

1. Unbundle the distribution file with `gunzip -c slepc.tgz | tar xvf -` or an equivalent command. This will create a directory and unpack the software there.

2. Refer to http://www.grycap.upv.es/slepc/download.htm for available patches to the latest SLEPc release.

3. Set the environment variable `SLEPC_DIR` to the full path of the SLEPc home directory, for example,

   ```
   setenv SLEPC_DIR /home/username/slepc-2.3.x
   ```

   In addition to this variable, `PETSC_DIR` (and optionally `PETSC_ARCH`) must also be set appropriately.

4. In the SLEPc directory, execute

   ```
   ./config/configure.py
   ```

Note that in order to enable external packages (see subsection 5.5), this command must be run with appropriate options. To see all the available options use `./config/configure.py --help`.

5. In the SLEPc home directory, type

```
make
```

6. Optionally, if an installation directory has been specified during configuration (with option `--prefix` in step 4 above), then type

```
make install
```

This is useful for building as a regular user and then copying the libraries and include files to the system directories as root.

7. If the installation went smoothly, then try running some test examples with

```
make test
```

Examine the output for any obvious errors or problems.

Note about complex scalar versions: PETSc supports the use of complex scalars by defining the data type `PetscScalar` either as a real or complex number. This implies that two different versions of the PETSc libraries can be built separately, one for real numbers and one for complex numbers, but they cannot be used at the same time. SLEPc inherits this property. In SLEPc it is not possible to completely separate real numbers and complex numbers because the solution of non-symmetric real-valued eigenvalue problems may be complex. SLEPc has been designed trying to provide a uniform interface to manage all the possible cases. However, there are slight differences between the interface in each of the two versions. In this manual, differences are clearly identified.

1.3 Running SLEPc Programs

Before using SLEPc, the user must first set the environment variable `SLEPC_DIR`, indicating the full path of the directory in which SLEPc has been installed. For example, under the UNIX C shell a command of the form

```
setenv SLEPC_DIR /software/slepc
```

can be placed in the user's `.cshrc` file. In addition, the user must set the environment variable required by PETSc, that is, `PETSC_DIR`, to indicate the full path of the PETSc installation. Optionally, the variable `PETSC_ARCH` can be set to specify a particular architecture and set of options.

All PETSc programs use the MPI (Message Passing Interface) standard for message-passing communication [MPI Forum, 1994]. Thus, to execute SLEPc programs, users must know the

procedure for launching MPI jobs on their selected computer system(s). For instance, when using the MPICH implementation of MPI and many others, the `mpirun` command can be used to initiate a program as in the following example that uses eight processes:

```
mpirun -np 8 slepc_program [command-line options]
```

In MPI-2 compliant systems, the command `mpiexec` can be used instead. Note that MPI may be deactivated during configuration of PETSc, if one wants to run only serial programs in a laptop, for example.

All PETSc-compliant programs support the use of the `-h` or `-help` option as well as the `-v` or `-version` option. In the case of SLEPc programs, specific information for SLEPc is also displayed.

1.4 Writing SLEPc Programs

Most SLEPc programs begin with a call to `SlepcInitialize`

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

which initializes SLEPc, PETSc and MPI. This subroutine is very similar to `PetscInitialize`, and the arguments have the same meaning. In fact, internally `SlepcInitialize` calls `PetscInitialize`.

After this initialization, SLEPc programs can use communicators defined by PETSc. In most cases users can employ the communicator `PETSC_COMM_WORLD` to indicate all processes in a given run and `PETSC_COMM_SELF` to indicate a single process. MPI provides routines for generating new communicators consisting of subsets of processes, though most users rarely need to use these features. SLEPc users need not program much message passing directly with MPI, but they must be familiar with the basic concepts of message passing and distributed memory computing.

All SLEPc programs should call `SlepcFinalize` as their final (or nearly final) statement

```
ierr = SlepcFinalize();
```

This routine handles options to be called at the conclusion of the program, and calls `PetscFinalize` if `SlepcInitialize` began PETSc.

Note to Fortran Programmers: In this manual all the examples and calling sequences are given for the C/C++ programming languages. However, Fortran programmers can use most of the functionality of SLEPc and PETSc from Fortran, with only minor differences in the user interface. For instance, the two functions mentioned above have their corresponding Fortran equivalent:

```
call SlepcInitialize(file,ierr)
call SlepcFinalize(ierr)
```

Section 5.6 provides a summary of the differences between using SLEPc from Fortran and C/C++, as well as a complete Fortran example.

1.4.1 Simple SLEPc Example

A simple example is listed next that solves an eigenvalue problem associated with the one-dimensional Laplacian operator discretized with finite differences. This example can be found in `${SLEPC_DIR}/src/examples/ex1.c`. Following the code we highlight a few of the most important parts of this example.

```
/*
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      SLEPc - Scalable Library for Eigenvalue Problem Computations
      Copyright (c) 2002-2007, Universidad Politecnica de Valencia, Spain

      This file is part of SLEPc. See the README file for conditions of use
      and additional information.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/

static char help[] = "Standard symmetric eigenproblem corresponding to the Laplacian operator in 1 dimension.\n\n"
  "The command line options are:\n"
  "  -n <n>, where <n> = number of grid subdivisions = matrix dimension.\n\n";

#include "slepceps.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main( int argc, char **argv )
{
  Mat            A;           /* operator matrix */
  EPS            eps;         /* eigenproblem solver context */
  EPSType        type;
  PetscReal      error, tol,re, im;
  PetscScalar    kr, ki;
  Vec            xr, xi;
  PetscErrorCode ierr;
  PetscInt       n=30, i, Istart, Iend, col[3];
  int            nev, maxit,its, nconv;
  PetscTruth     FirstBlock=PETSC_FALSE, LastBlock=PETSC_FALSE;
  PetscScalar    value[3];

  SlepcInitialize(&argc,&argv,(char*)0,help);

  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"\n1-D Laplacian Eigenproblem, n=%d\n\n",n);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Compute the operator matrix that defines the eigensystem, Ax=kx
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
  ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);

  ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
  if (Istart==0) FirstBlock=PETSC_TRUE;
  if (Iend==n) LastBlock=PETSC_TRUE;
  value[0]=-1.0; value[1]=2.0; value[2]=-1.0;
  for( i=(FirstBlock? Istart+1: Istart); i<(LastBlock? Iend-1: Iend); i++ ) {
    col[0]=i-1; col[1]=i; col[2]=i+1;
    ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
  }
  if (LastBlock) {
    i=n-1; col[0]=n-2; col[1]=n-1;
```

```
      ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
    }
    if (FirstBlock) {
      i=0; col[0]=0; col[1]=1; value[0]=2.0; value[1]=-1.0;
      ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
    }

    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

    ierr = MatGetVecs(A,PETSC_NULL,&xr);CHKERRQ(ierr);
    ierr = MatGetVecs(A,PETSC_NULL,&xi);CHKERRQ(ierr);

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                  Create the eigensolver and set various options
       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
    /*
       Create eigensolver context
    */
    ierr = EPSCreate(PETSC_COMM_WORLD,&eps);CHKERRQ(ierr);

    /*
       Set operators. In this case, it is a standard eigenvalue problem
    */
    ierr = EPSSetOperators(eps,A,PETSC_NULL);CHKERRQ(ierr);
    ierr = EPSSetProblemType(eps,EPS_HEP);CHKERRQ(ierr);

    /*
       Set solver parameters at runtime
    */
    ierr = EPSSetFromOptions(eps);CHKERRQ(ierr);

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      Solve the eigensystem
       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

    ierr = EPSSolve(eps);CHKERRQ(ierr);
    ierr = EPSGetIterationNumber(eps, &its);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD," Number of iterations of the method: %d\n",its);CHKERRQ(ierr);
    /*
       Optional: Get some information from the solver and display it
    */
    ierr = EPSGetType(eps,&type);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD," Solution method: %s\n\n",type);CHKERRQ(ierr);
    ierr = EPSGetDimensions(eps,&nev,PETSC_NULL);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD," Number of requested eigenvalues: %d\n",nev);CHKERRQ(ierr);
    ierr = EPSGetTolerances(eps,&tol,&maxit);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD," Stopping condition: tol=%.4g, maxit=%d\n",tol,maxit);CHKERRQ(ierr);

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                    Display solution and clean up
       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
    /*
       Get number of converged approximate eigenpairs
    */
    ierr = EPSGetConverged(eps,&nconv);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD," Number of converged eigenpairs: %d\n\n",nconv);CHKERRQ(ierr);

    if (nconv>0) {
      /*
         Display eigenvalues and relative errors
      */
      ierr = PetscPrintf(PETSC_COMM_WORLD,
```

```
          "           k          ||Ax-kx||/||kx||\n"
          "   ----------------- ------------------\n" );CHKERRQ(ierr);

      for( i=0; i<nconv; i++ ) {
        /*
          Get converged eigenpairs: i-th eigenvalue is stored in kr (real part) and
          ki (imaginary part)
        */
        ierr = EPSGetEigenpair(eps,i,&kr,&ki,xr,xi);CHKERRQ(ierr);
        /*
           Compute the relative error associated to each eigenpair
        */
        ierr = EPSComputeRelativeError(eps,i,&error);CHKERRQ(ierr);

#ifdef PETSC_USE_COMPLEX
        re = PetscRealPart(kr);
        im = PetscImaginaryPart(kr);
#else
        re = kr;
        im = ki;
#endif
        if (im!=0.0) {
          ierr = PetscPrintf(PETSC_COMM_WORLD," %9f%+9f j %12g\n",re,im,error);CHKERRQ(ierr);
        } else {
          ierr = PetscPrintf(PETSC_COMM_WORLD,"   %12f       %12g\n",re,error);CHKERRQ(ierr);
        }
      }
      ierr = PetscPrintf(PETSC_COMM_WORLD,"\n" );CHKERRQ(ierr);
    }

    /*
       Free work space
    */
    ierr = EPSDestroy(eps);CHKERRQ(ierr);
    ierr = MatDestroy(A);CHKERRQ(ierr);
    ierr = VecDestroy(xr);CHKERRQ(ierr);
    ierr = VecDestroy(xi);CHKERRQ(ierr);
    ierr = SlepcFinalize();CHKERRQ(ierr);
    return 0;
}
```

Include Files

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

```
#include "slepceps.h"
```

where `slepceps.h` is the include file for the `EPS` component. Each SLEPc program must specify an include file that corresponds to the highest level SLEPc objects needed within the program; all of the required lower level include files are automatically included within the higher level files. For example, `slepceps.h` includes `slepcst.h` (spectral transformations), and `slepc.h` (base SLEPc file). Some PETSc header files are included as well, such as `petscksp.h`. The SLEPc include files are located in the directory `${SLEPC_DIR}/include`.

The Options Database

All the PETSc functionality related to the options database is available in SLEPc. This allows the user to input control data at run time very easily. In this example the command

`PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);` checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged.

Vectors and Matrices

Usage of matrices and vectors in SLEPc is exactly the same as in PETSc. The user can create a new parallel or sequential matrix, `A`, which has `M` global rows and `N` global columns, with

```
MatCreate(MPI_Comm comm,Mat *A);
MatSetSizes(Mat A,int m,int n,int M,int N);
MatSetFromOptions(Mat A);
```

where the matrix format can be specified at runtime. The example creates a matrix, sets the nonzero values with `MatSetValues` and then assembles it.

Eigensolvers

Usage of eigensolvers is very similar to other kinds of solvers provided by PETSc. After creating the matrix (or matrices) that define the problem, $Ax = kx$ (or $Ax = kBx$), the user can then use `EPS` to solve the system with the following sequence of commands:

```
EPSCreate(MPI_Comm comm,EPS *eps);
EPSSetOperators(EPS eps,Mat A,Mat B);
EPSSetProblemType(EPS eps,EPSProblemType type);
EPSSetFromOptions(EPS eps);
EPSSolve(EPS eps);
EPSGetConverged(EPS eps, int *nconv);
EPSGetEigenpair(EPS eps,int i,PetscScalar *kr,PetscScalar *ki,Vec xr,Vec xi);
EPSDestroy(EPS eps);
```

The user first creates the `EPS` context and sets the operators associated with the eigensystem as well as the problem type. The user then sets various options for customized solution, solves the problem, retrieves the solution, and finally destroys the `EPS` context. Chapter 2 describes in detail the `EPS` package, including the options database that enables the user to customize the solution process at runtime by selecting the solution algorithm and also specifying the convergence tolerance, the number of eigenvalues, the dimension of the subspace, etc.

Spectral Transformation

In the example program shown above there is no explicit reference to spectral transformations. However, an `ST` object is handled internally so that the user is able to request different transformations such as shift-and-invert. Chapter 3 describes the `ST` package in detail.

Error Checking

All SLEPc routines return an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. The PETSc macro `CHKERRQ(ierr)` checks the value of `ierr` and calls the PETSc error handler upon error detection. `CHKERRQ(ierr)` should be placed after all subroutine calls to enable a complete error traceback. See the PETSc documentation for full details.

1.4.2 Writing Application Codes with SLEPc

The examples provided in the `src/examples` directory demonstrate the software usage and can serve as templates for developing custom applications. To write a new application program using SLEPc, we suggest the following procedure:

1. Install and test SLEPc according to the instructions given in the documentation.
2. Copy the SLEPc example that corresponds to the class of problem of interest (e.g. singular value decomposition).
3. Copy the makefile within the example directory (or create a new one as explained below); compile and run the example program.
4. Use the example program as a starting point for developing a custom code.

Application program makefiles can be set up very easily just by including one file from the SLEPc makefile system. All the necessary PETSc definitions are loaded automatically. The following sample makefile illustrates how to build C and Fortran programs:

```
default: ex1

include ${SLEPC_DIR}/bmake/slepc_common

ex1: ex1.o chkopts
        -${CLINKER} -o ex1 ex1.o ${SLEPC_LIB}
        ${RM} ex1.o

ex1f: ex1f.o chkopts
        -${FLINKER} -o ex1f ex1f.o ${SLEPC_LIB}
        ${RM} ex1f.o
```

Chapter 2

EPS: Eigenvalue Problem Solver

The Eigenvalue Problem Solver (`EPS`) is the main object provided by SLEPc. It is used to specify an eigenvalue problem, either in standard or generalized form, and provides uniform and efficient access to all of the eigensolvers included in the package. Conceptually, the level of abstraction occupied by `EPS` is similar to other solvers in PETSc such as `KSP` for solving linear systems of equations.

2.1 Eigenvalue Problems

In this section, we present very briefly some basic concepts about eigenvalue problems as well as general techniques used to solve them. The description is not intended to be exhaustive. The objective is simply to define terms that will be referred to throughout the rest of the manual. Readers who are familiar with the terminology and the solution approach can skip this section. For a more comprehensive description, we refer the reader to monographs such as [Stewart, 2001], [Bai *et al.*, 2000], [Saad, 1992] or [Parlett, 1980]. A historical perspective of the topic can be found in [Golub and van der Vorst, 2000]. See also the SLEPc technical reports.

In the standard formulation, the eigenvalue problem consists in the determination of $\lambda \in \mathbb{C}$ for which the equation

$$Ax = \lambda x \tag{2.1}$$

has nontrivial solution, where $A \in \mathbb{C}^{n\times n}$ and $x \in \mathbb{C}^n$. The scalar λ and the vector x are called eigenvalue and (right) eigenvector, respectively. Note that they can be complex even when the matrix is real. If λ is an eigenvalue of A then $\bar{\lambda}$ is an eigenvalue of its conjugate transpose, A^*, or equivalently

$$y^*A = \lambda\, y^* \quad , \tag{2.2}$$

where y is called the left eigenvector.

In many applications, the problem is formulated as

$$Ax = \lambda Bx \quad , \tag{2.3}$$

where $B \in \mathbb{C}^{n\times n}$, which is known as the generalized eigenvalue problem. Usually, this problem is solved by reformulating it in standard form, for example $B^{-1}Ax = \lambda x$ if B is non-singular.

SLEPc focuses on the solution of problems in which the matrices are large and sparse. Hence, only methods that preserve sparsity are considered. These methods obtain the solution from the information generated by the application of the operator to various vectors (the operator is a simple function of matrices A and B), that is, matrices are only used in matrix-vector products. This not only maintains sparsity but allows the solution of problems in which matrices are not available explicitly.

In practical analyses, from the n possible solutions, typically only a few eigenpairs (λ, x) are considered relevant, either in the extremities of the spectrum, in an interval, or in a region of the complex plane. Depending on the application, either eigenvalues or eigenvectors or both are required. In some cases, left eigenvectors are also of interest.

Projection Methods. Most eigensolvers provided by SLEPc perform a Rayleigh-Ritz projection for extracting the spectral approximations, that is, they project the problem onto a low-dimensional subspace that is built appropriately. Suppose that an orthogonal basis of this subspace is given by $V_j = [v_1, v_2, \ldots, v_j]$. If the solutions of the projected (reduced) problem $B_j s = \theta s$ (i.e., $V_j^T A V_j = B_j$) are assumed to be (θ_i, s_i), $i = 1, 2, \ldots, j$, then the approximate eigenpairs $(\tilde{\lambda}_i, \tilde{x}_i)$ of the original problem (Ritz value and Ritz vector) are obtained as

$$\tilde{\lambda}_i = \theta_i \quad , \tag{2.4}$$

$$\tilde{x}_i = V_j s_i \quad . \tag{2.5}$$

Starting from this general idea, eigensolvers differ from each other in which subspace is used, how it is built and other technicalities aimed at improving convergence, reducing storage requirements, etc.

The subspace

$$\mathcal{K}_m(A, v) \equiv \text{span}\left\{v, Av, A^2v, \ldots, A^{m-1}v\right\} \quad , \tag{2.6}$$

is called the m-th Krylov subspace corresponding to A and v. Methods that use subspaces of this kind to carry out the projection are called Krylov methods. One example of such methods is the Arnoldi algorithm: starting with v_1, $\|v_1\|_2 = 1$, the Arnoldi basis generation process can be expressed by the recurrence

$$v_{j+1}h_{j+1,j} = w_j = Av_j - \sum_{i=1}^{j} h_{i,j}v_i \quad , \tag{2.7}$$

where $h_{i,j}$ are the scalar coefficients obtained in the Gram-Schmidt orthogonalization of Av_j with respect to v_i, $i = 1, 2, \ldots, j$, and $h_{j+1,j} = \|w_j\|_2$. Then, the columns of V_j span the Krylov

subspace $\mathcal{K}_j(A, v_1)$ and $Ax = \lambda x$ is projected into $H_j s = \theta s$, where H_j is an upper Hessenberg matrix with elements $h_{i,j}$, which are 0 for $i \geq j + 2$. The related Lanczos algorithms obtain a projected matrix that is tridiagonal.

A generalization to the above methods are the block Krylov strategies, in which the starting vector v_1 is replaced by a full rank $n \times p$ matrix V_1, which allows for better convergence properties when there are multiple eigenvalues and can provide better data management on some computer architectures. Block tridiagonal and block Hessenberg matrices are then obtained as projections.

It is generally assumed (and observed) that the Lanczos and Arnoldi algorithms find solutions at the extremities of the spectrum. Their convergence pattern, however, is strongly related to the eigenvalue distribution. Slow convergence may be experienced in the presence of tightly clustered eigenvalues. The maximum allowable j may be reached without having achieved convergence for all desired solutions. Then, restarting is usually a useful technique and different strategies exist for that purpose. However, convergence can still be very slow and acceleration strategies must be applied. Usually, these techniques consists in computing eigenpairs of a transformed operator and then recovering the solution of the original problem. The aim of these transformations is twofold. On one hand, they allow to obtain eigenvalues other than those lying in the boundary of the spectrum. On the other hand, the separation of the eigenvalues of interest is improved in the transformed spectrum thus leading to fast convergence. The most commonly used spectral transformation is called shift-and-invert, which works with operator $(A - \sigma I)^{-1}$. It allows to compute the eigenvalues closest to σ with very good separation properties. When using this approach, a linear system of equations, $(A - \sigma I)y = x$, must be solved in each iteration of the eigenvalue process.

Related Problems. In many applications such as the analysis of damped vibrating systems the eigenproblem to be solved is quadratic,

$$(A\lambda^2 + B\lambda + C)x = 0 \ . \tag{2.8}$$

It is possible to transform this problem to a generalized eigenproblem by increasing the order of the system. For example, let the eigenvector be $v = [\lambda x, x]^T$, then the equivalent system is

$$\begin{bmatrix} -B & -C \\ I & 0 \end{bmatrix} v = \lambda \begin{bmatrix} A & 0 \\ 0 & I \end{bmatrix} v \ . \tag{2.9}$$

Another linear algebra problem that is very closely related to the eigenvalue problem is the *singular value decomposition* (SVD). SLEPc provides a specific package for SVD computation, so the description is postponed until chapter 4.

2.2 Basic Usage

The `EPS` module in SLEPc is used in a similar way as PETSc modules such as `KSP`. All the information related to an eigenvalue problem is handled via a context variable. The usual object management functions are available (`EPSCreate`, `EPSDestroy`, `EPSView`, `EPSSetFromOptions`).

```
EPS         eps;       /*  eigensolver context  */
Mat         A;         /*  matrix of Ax=kx      */
Vec         xr, xi;    /*  eigenvector, x       */
PetscScalar kr, ki;    /*  eigenvalue, k        */
int         j, nconv;
PetscReal   error;

EPSCreate( PETSC_COMM_WORLD, &eps );
EPSSetOperators( eps, A, PETSC_NULL );
EPSSetProblemType( eps, EPS_NHEP );
EPSSetFromOptions( eps );
EPSSolve( eps );
EPSGetConverged( eps, &nconv );
for (j=0; j<nconv; j++) {
  EPSGetEigenpair( eps, j, &kr, &ki, xr, xi );
  EPSComputeRelativeError( eps, j, &error );
}
EPSDestroy( eps );
```

Figure 2.1: Example code for basic solution with EPS.

In addition, the EPS object provides functions for setting several parameters such as the number of eigenvalues to compute, the dimension of the subspace, the portion of the spectrum of interest, the requested tolerance or the maximum number of iterations allowed.

The solution of the problem is obtained in several steps. First of all, the matrices associated to the eigenproblem are specified via EPSSetOperators and EPSSetProblemType is used to specify the type of problem. Then, a call to EPSSolve is done that invokes the subroutine for the selected eigensolver. EPSGetConverged can be used afterwards to determine how many of the requested eigenpairs have converged to working accuracy. EPSGetEigenpair is finally used to retrieve the eigenvalues and eigenvectors.

In order to illustrate the basic functionality of the EPS package, a simple example is shown in figure 2.1. The example code implements the solution of a simple standard eigenvalue problem. Code for setting up the matrix A is not shown and error-checking code is omitted.

All the operations of the program are done over a single EPS object. This solver context is created in line 8 with the command

```
EPSCreate(MPI_Comm comm,EPS *eps);
```

Here comm is the MPI communicator, and eps is the newly formed solver context. The communicator indicates which processes are involved in the EPS object. Most of the EPS operations are collective, meaning that all the processes collaborate to perform the operation in parallel.

Before actually solving an eigenvalue problem with EPS, the user must specify the matrices associated to the problem, as in line 9, with the following routine

```
EPSSetOperators(EPS eps,Mat A,Mat B);
```

The example specifies a standard eigenproblem. In the case of a generalized problem, it would be necessary also to provide matrix B as the third argument to the call. The matrices specified in this call can be in any PETSc format. In particular, `EPS` allows the user to solve matrix-free problems by specifying matrices created via `MatCreateShell`. A more detailed discussion of this issue is given in section 5.2.

After setting the problem matrices, the problem type is set with `EPSSetProblemType`. This is not strictly necessary since if this step is skipped then the problem type is assumed to be non-symmetric. More details are given in section 2.3. At this point, the value of the different options could optionally be set by means of a function call such as `EPSSetTolerances` (explained later in this chapter). After this, a call to `EPSSetFromOptions` should be made as in line 11,

```
EPSSetFromOptions(EPS eps);
```

The effect of this call is that options specified at runtime in the command line are passed to the `EPS` object appropriately. In this way, the user can easily experiment with different combinations of options without having to recompile. All the available options as well as the associated function calls are described later in this chapter.

Line 12 launches the solution algorithm, simply with the command

```
EPSSolve(EPS eps);
```

The subroutine that is actually invoked depends on which solver has been selected by the user.

After the call to `EPSSolve` has finished, all the data associated to the solution of the eigenproblem is kept internally. This information can be retrieved with different function calls, as in lines 13 to 17. This part is described in detail in subsection 2.5.

Once the `EPS` context is no longer needed, it should be destroyed with the command

```
EPSDestroy(EPS eps);
```

The above procedure is sufficient for general use of the `EPS` package. As in the case of the `KSP` solver, the user can optionally explicitly call

```
EPSSetUp(EPS eps);
```

before calling `EPSSolve` to perform any setup required for the eigensolver.

Internally, the `EPS` object works with an `ST` object (spectral transformation, described in chapter 3). To allow application programmers to set any of the spectral transformation options directly within the code, the following routine is provided to extract the `ST` context,

```
EPSGetST(EPS eps,ST *st);
```

With the command

```
EPSView(EPS eps,PetscViewer viewer);
```

it is possible to examine the information relevant to the `EPS` object, such as the value of the different parameters, including also data related to the associated `ST` object.

Problem Type	EPSProblemType	Command line key
Hermitian	EPS_HEP	-eps_hermitian
Non-Hermitian	EPS_NHEP	-eps_non_hermitian
Generalized Hermitian	EPS_GHEP	-eps_gen_hermitian
Generalized Non-Hermitian	EPS_GNHEP	-eps_gen_non_hermitian
GNHEP with positive (semi-)definite B	EPS_PGNHEP	-eps_pos_gen_non_hermitian

Table 2.1: Problem types considered in EPS.

2.3 Defining the Problem

SLEPc is able to cope with different kinds of problems. Currently supported problem types are listed in table 2.1. An eigenproblem is generalized ($Ax = \lambda Bx$) if the user has specified two matrices (see `EPSSetOperators` above), otherwise it is standard ($Ax = \lambda x$). A standard eigenproblem is Hermitian if matrix A is Hermitian (i.e., $A = A^*$) or, equivalently in the case of real matrices, if matrix A is symmetric (i.e., $A = A^T$). A generalized eigenproblem is Hermitian if matrix A is Hermitian (symmetric) and matrix B is Hermitian (symmetric) and positive (semi-)definite. A special case of generalized non-Hermitian problem is when A is non-Hermitian but B is Hermitian and positive (semi-)definite, see sections 3.4.3 and 3.4.4 for discussion.

The problem type can be specified at run time with the corresponding command line key or, more usually, within the program with the function

```
EPSSetProblemType(EPS eps,EPSProblemType type);
```

By default, SLEPc assumes that the problem is non-Hermitian. Some eigensolvers are able to exploit symmetry, that is, they compute a solution for Hermitian problems with less storage and/or computational cost than other methods that ignore this property. Also, symmetric solvers are typically more accurate. On the other hand, some eigensolvers in SLEPc only have a symmetric version and will abort if the problem is non-Hermitian. In the case of generalized eigenproblems some considerations apply regarding symmetry, especially in the case of singular B. This topic is tackled in subsections 3.4.3 and 3.4.4. For all these reasons, the user is strongly recommended to always specify the problem type in the source code.

The type of the problem can be determined with the functions

```
EPSIsGeneralized(EPS eps,PetscTruth *gen);
EPSIsHermitian(EPS eps,PetscTruth *her);
```

The user can specify how many eigenvalues (and eigenvectors) to compute. The default is to compute only one. The function

```
EPSSetDimensions(EPS eps,int nev,int ncv);
```

allows the specification of the number of eigenvalues to compute, `nev`. The last argument can be set to prescribe the number of column vectors to be used by the solution algorithm, `ncv`,

EPSWhich	Command line key	Sorting criterion
EPS_LARGEST_MAGNITUDE	-eps_largest_magnitude	Largest $\|\lambda\|$
EPS_SMALLEST_MAGNITUDE	-eps_smallest_magnitude	Smallest $\|\lambda\|$
EPS_LARGEST_REAL	-eps_largest_real	Largest $\mathrm{Re}(\lambda)$
EPS_SMALLEST_REAL	-eps_smallest_real	Smallest $\mathrm{Re}(\lambda)$
EPS_LARGEST_IMAGINARY	-eps_largest_imaginary	Largest $\mathrm{Im}(\lambda)$[1]
EPS_SMALLEST_IMAGINARY	-eps_smallest_imaginary	Smallest $\mathrm{Im}(\lambda)$[1]

Table 2.2: Available possibilities for selection of the eigenvalues of interest.

that is, the largest dimension of the working subspace. These two parameters can also be set at run time with the options `-eps_nev` and `-eps_ncv`. For example, the command line

```
$ program -eps_nev 10 -eps_ncv 24
```

requests 10 eigenvalues and instructs to use 24 column vectors. Note that `ncv` must be at least equal to `nev`, although in general it is recommended (depending on the method) to work with a larger subspace, for instance $\texttt{ncv} \geq 2 \cdot \texttt{nev}$ or even more.

For the selection of the portion of the spectrum of interest, there are several alternatives. In real symmetric problems, one may want to compute the largest or smallest eigenvalues in magnitude, or the leftmost or rightmost ones. In other problems, in which the eigenvalues can be complex, then one can select eigenvalues depending on the magnitude, or the real part or even the imaginary part. Table 2.2 sumarizes all the possibilities available for the function

```
EPSSetWhichEigenpairs(EPS eps,EPSWhich which);
```

which can also be specified at the command line. This criterion is used both for configuring how the eigensolver seeks eigenvalues (note that not all these possibilities are available for all the solvers) and also for sorting the computed values. The default is to compute the largest magnitude eigenvalues, except for those solvers in which this option is not available. To compute eigenvalues located in the interior part of the spectrum, the user should use a spectral transformation (see chapter 3). Note that in this case, the value of `which` applies to the transformed spectrum.

2.4 Selecting the Eigensolver

The available methods for solving the eigenvalue problems are the following:

- Power Iteration with deflation. When combined with shift-and-invert (see chapter 3), it is equivalent to the Inverse Iteration. Also, this solver embeds the Rayleigh Quotient Iteration (RQI) by allowing variable shifts.

[1] If SLEPc is compiled for real scalars, then the absolute value of the imaginary part, $|\mathrm{Im}(\lambda)|$, is used for eigenvalue selection and sorting.

Method	EPSType	Options Database Name
Power / Inverse / RQI	EPSPOWER	power
Subspace Iteration	EPSSUBSPACE	subspace
Arnoldi	EPSARNOLDI	arnoldi
Lanczos	EPSLANCZOS	lanczos
Krylov-Schur	EPSKRYLOVSCHUR	krylovschur
LAPACK solver	EPSLAPACK	lapack
Wrapper to ARPACK	EPSARPACK	arpack
Wrapper to PRIMME	EPSPRIMME	primme
Wrapper to BLZPACK	EPSBLZPACK	blzpack
Wrapper to TRLAN	EPSTRLAN	trlan
Wrapper to BLOPEX	EPSBLOPEX	blopex

Table 2.3: Eigenvalue solvers available in the EPS module.

- Subspace Iteration with Rayleigh-Ritz projection and locking.
- Arnoldi method with explicit restart and deflation.
- Lanczos with explicit restart and deflation, using different reorthogonalization strategies.
- Krylov-Schur, which is a variation of Arnoldi/Lanczos with a very effective restarting technique.

The default solver is Krylov-Schur. A detailed description of the implemented algorithms is provided in the SLEPc Technical Reports. In addition to these methods, SLEPc also provides wrappers to external packages such as ARPACK, BLZPACK, or TRLAN. A complete list of these interfaces can be found in section 5.5.

As an alternative, SLEPc provides an interface to some LAPACK routines. These routines operate in dense mode with only one processor and therefore are suitable only for moderate size problems. This solver should be used only for debugging purposes.

The solution method can be specified procedurally or via the command line. The application programmer can set it by means of the command

```
EPSSetType(EPS eps,EPSType method);
```

while the user writes the options database command `-eps_type` followed by the name of the method (see table 2.3).

Not all the methods can be used for all problem types. Table 2.4 summarizes the scope of each eigensolver by listing which portion of the spectrum can be selected (as defined in table 2.2), which problem types are supported (as defined in table 2.1) and whether they are available or not in the complex version of SLEPc. Also, the default value of some parameters differ from one solver to the other, as shown in Table 2.5. This table also illustrates the different storage requirements. All solvers need memory at least for storing *ncv* vectors, but in addition some

Method	Portion of spectrum	Problem type	Complex
`power`	Largest $\|\lambda\|$	all	yes
`subspace`	Largest $\|\lambda\|$	all	yes
`arnoldi`	all	all	yes
`lanczos`	all	`EPS_HEP`, `EPS_GHEP`	yes
`krylovschur`	all	all	yes
`lapack`	all	all	yes
`arpack`	all	all	yes
`primme`	Largest and smallest Re(λ)	`EPS_HEP`	yes
`blzpack`	Smallest Re(λ)	`EPS_HEP`, `EPS_GHEP`	no
`trlan`	Largest and smallest Re(λ)	`EPS_HEP`	no
`blopex`	Smallest Re(λ)	`EPS_HEP`	no

Table 2.4: Supported problem types for all eigensolvers available in SLEPc.

extra work storage such as auxiliary vectors is necessary. The last columns of Table 2.5 indicates the number of auxiliary vectors required in each case.

2.5 Retrieving the Solution

Once the call to `EPSSolve` is complete, all the data associated to the solution of the eigenproblem is kept internally in the `EPS` object. This information can be obtained by the calling program by means of a set of functions described in this section.

As explained below, the number of computed solutions depends on the convergence and, therefore, it may be different from the number of solutions requested by the user. So the first task is to find out how many solutions are available, with

```
EPSGetConverged(EPS eps,int *nconv);
```

Usually, the number of converged solutions, `nconv`, will be equal to `nev`, but in general it will be a number ranging from 0 to `ncv` (here, `nev` and `ncv` are the arguments of function `EPSSetDimensions`).

2.5.1 The Computed Solution

Normally, the user is interested in the eigenvalues, or the eigenvectors, or both. The function

```
EPSGetEigenpair(EPS eps,int j,PetscScalar *kr,PetscScalar *ki, Vec xr, Vec xi);
```

returns the j-th computed eigenvalue/eigenvector pair. Typically, this function is called inside a loop for each value of `j` from 0 to `nconv`–1. Note that eigenvalues are ordered according to the same criterion specified with function `EPSSetWhichEigenpairs` for selecting the portion of the spectrum of interest.

Method	`ncv`	`max_it`	Storage
`power`	nev	$\max(2000, 100N)$	2
`subspace`	$\max(2 \cdot nev, nev + 15)$	$\max(100, \lceil 2N/ncv \rceil)$	ncv
`arnoldi`	$\max(2 \cdot nev, nev + 15)$	$\max(100, \lceil 2N/ncv \rceil)$	1
`lanczos`	$\max(2 \cdot nev, nev + 15)$	$\max(100, \lceil 2N/ncv \rceil)$	1
`krylovschur`	$\max(2 \cdot nev, nev + 15)$	$\max(100, \lceil 2N/ncv \rceil)$	1
`lapack`	N	-	N
`arpack`	$\max(20, 2 \cdot nev+1)$	$\max(300, \lceil 2N/ncv \rceil)$	4
`primme`	$\max(20, 2 \cdot nev+1)$	$\max(1000, N)$	3
`blzpack`	$\min(nev+10, 2 \cdot nev)$	$\max(1000, N)$	> 187
`trlan`	nev	$\max(1000, N)$	$nev + 1$
`blopex`	nev	$\max(100, \lceil 2N/ncv \rceil)$	nev

Table 2.5: Default parameter values for all eigensolvers available in SLEPc.

The meaning of the last 4 parameters depends on whether SLEPc has been compiled for real or complex scalars, as detailed below. In all cases, the eigenvectors are normalized so that they have a unit 2-norm.

Real SLEPc**.** In this case, all `Mat` and `Vec` objects are real. The computed approximate solution returned by the function `EPSGetEigenpair` is stored in the following way: `kr` and `ki` contain the real and imaginary parts of the eigenvalue, respectively, and `xr` and `xi` contain the associated eigenvector. Two cases can be distinguished:

- When `ki` is zero, it means that the j-th eigenvalue is a real number. In this case, `kr` is the eigenvalue and `xr` is the corresponding eigenvector. The vector `xi` is set to all zeros.

- If `ki` is different from zero, then the j-th eigenvalue is a complex number and, therefore, it is part of a complex conjugate pair. Thus, the j-th eigenvalue is `kr`$+ i \cdot$`ki`. With respect to the eigenvector, `xr` stores the real part of the eigenvector and `xi` the imaginary part, that is, the j-th eigenvector is `xr`$+ i \cdot$`xi`. The $(j+1)$-th eigenvalue (and eigenvector) will be the corresponding complex conjugate and will be returned when function `EPSGetEigenpair` is invoked with index `j`+1. Note that the sign of the imaginary part is returned correctly in all cases (users need not change signs).

Complex SLEPc**.** In this case, all `Mat` and `Vec` objects are complex. The computed solution returned by function `EPSGetEigenpair` is the following: `kr` contains the (complex) eigenvalue and `xr` contains the corresponding (complex) eigenvector. In this case, `ki` and `xi` are not used (set to all zeros).

2.5.2 Reliability of the Computed Solution

In this subsection, we discuss how a-posteriori error bounds can be obtained in order to assess the accuracy of the computed solutions. These bounds are based on the so-called residual vector, defined as

$$r = A\tilde{x} - \tilde{\lambda}\tilde{x} \quad , \tag{2.10}$$

or $r = A\tilde{x} - \tilde{\lambda}B\tilde{x}$ in the case of a generalized problem, where $\tilde{\lambda}$ and $\tilde{x}$ represent any of the `nconv` computed eigenpairs delivered by `EPSGetEigenpair` (note that this function returns a normalized $\tilde{x}$).

In the case of Hermitian problems, it is possible to demonstrate the following property (see for example [Saad, 1992, ch. 3]):

$$|\lambda - \tilde{\lambda}| \leq \|r\|_2 \quad , \tag{2.11}$$

where λ is an exact eigenvalue. Therefore, the 2-norm of the residual vector can be used as a bound for the absolute error in the eigenvalue. The following SLEPc function

```
EPSComputeResidualNorm(EPS eps,int j,PetscReal *norm)
```

computes the 2-norm of r_j. If we want to express the error relative to the eigenvalue, then the following function can be used instead:

```
EPSComputeRelativeError(EPS eps,int j,PetscReal *error);
```

In the case of non-Hermitian problems, the situation is worse because no simple relation such as Eq. 2.11 is available. This means that in this case the error bounds may still give an indication of the actual error but the user should be aware that they may sometimes be completely wrong, especially in the case of highly non-normal matrices.

With respect to eigenvectors, we have a similar scenario in the sense that bounds for the error may be established in the Hermitian case only, for example the following one:

$$\sin\theta(x,\tilde{x}) \leq \frac{\|r\|_2}{\delta} \quad , \tag{2.12}$$

where $\theta(x,\tilde{x})$ is the angle between the computed and exact eigenvectors, and δ is the distance from $\tilde{\lambda}$ to the rest of the spectrum. This bound is not provided by SLEPc because δ is not available. The above expression is given here simply to warn the user about the fact that accuracy of eigenvectors may be deficient in the case of clustered eigenvalues.

In the case of non-Hermitian problems, SLEPc provides the alternative of retrieving an orthonormal basis of an invariant subspace instead of getting individual eigenvectors. This is done with function

```
EPSGetInvariantSubspace(EPS eps,Vec *v)
```

This is sufficient in some applications and is safer from the numerical point of view.

2.5.3 Controlling and Monitoring Convergence

All the eigensolvers provided by SLEPc are iterative in nature, meaning that the solutions are (usually) improved at each iteration until they are sufficiently accurate, that is, until convergence is obtained. The number of iterations required by the process can be obtained with the function

```
EPSGetIterationNumber(EPS eps,int *its);
```

which returns in argument `its` either the iteration number at which convergence was successfully reached, or the iteration at which a problem was detected.

The user specifies when a solution should be considered sufficiently accurate by means of a tolerance. An approximate eigenvalue is considered to be converged if the error estimate associated to it is below the specified tolerance. Note that the error estimates can be computed differently depending on the solution method. The default value of the tolerance is 10^{-7} and can be changed at run time with `-eps_tol <tol>` or inside the program with the function

```
EPSSetTolerances(EPS eps,PetscReal tol,int max_it);
```

The third parameter of this function allows the programmer to modify the maximum number of iterations allowed to the solution algorithm, which can also be set via `-eps_max_it <its>`.

Error estimates used internally by eigensolvers for checking convergence may be different from the error bounds provided by `EPSComputeRelativeError`. At the end of the solution process, error estimates are available via

```
EPSGetErrorEstimate(EPS eps,int j,PetscReal *errest);
```

Error estimates can also be displayed during execution of the solution algorithm, as a way of monitoring convergence. The user can activate this feature by using `-eps_monitor` within the options database. By default, the solvers run silently without displaying information about the iteration. When the option `-eps_monitor` is given, then the approximate eigenvalues together with the associated error estimates are printed in each iteration. Application programmers can provide their own routines to perform the monitoring by using the function `EPSMonitorSet`.

Graphical monitoring (in an X display) is also available with `-eps_monitor_draw`. Also, the options database key `-eps_plot_eigs` instructs SLEPc to plot the computed approximations of the eigenvalues at the end of the process. See Fig. 2.2 for an example.

2.6 Advanced Usage

This section includes the description of several advanced features of the eigensolver object. The default settings are appropriate for most applications and modification is not necessary for normal usage.

2.6.1 Initial Vectors

Most of the algorithms implemented in SLEPc iteratively build and refine a basis of a certain subspace. This basis is constructed starting from an initial vector, v_1. `EPS` initializes this

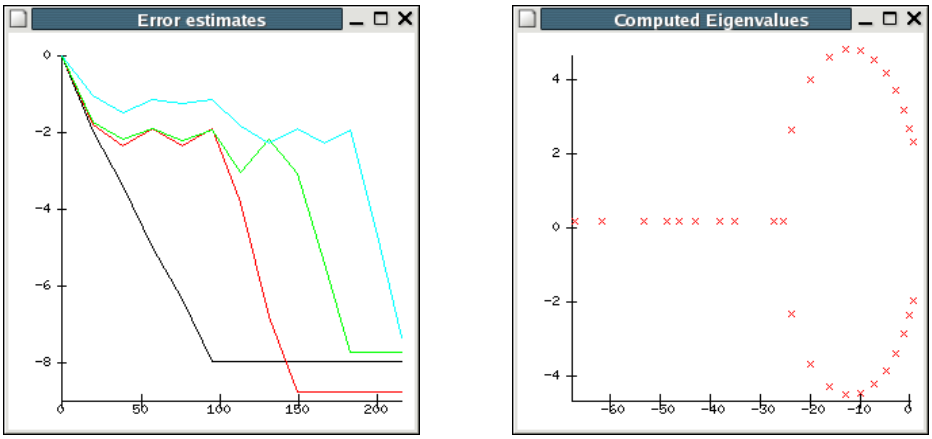

Figure 2.2: Graphical output in SLEPc: convergence monitor (left) and eigenvalue plot (right).

starting vector randomly. This default is a reasonable choice. However, it is also possible to supply the starting vector with the command

```
EPSSetInitialVector(EPS eps,Vec v0);
```

In some cases, a suitable starting vector can accelerate convergence. For this, the initial vector should be rich in the directions of wanted eigenvectors. This is the case for example when the eigenvalue calculation is one of a sequence of closely related problems and the starting vector is built by taking a linear combination of the eigenvectors computed in a previously converged eigenvalue calculation.

2.6.2 Dealing with Deflation Subspaces

In some applications, when solving an eigenvalue problem the user wishes to use a priori knowledge about the solution. This is the case when an invariant subspace has already been computed (e.g. in a previous `EPSSolve` call) or when a basis of the null-space is known.

Consider the following example. Given a graph G, with vertex set V and edges E, the Laplacian matrix of G is a sparse symmetric positive semidefinite matrix L such that

$$l_{ij} = \begin{cases} d(v_i) & \text{if } i = j \\ -1 & \text{if } e_{ij} \in E \\ 0 & \text{otherwise} \end{cases}$$

where $d(v_i)$ is the degree of vertex v_i. This matrix is singular since all row sums are equal to zero. The constant vector is an eigenvector with zero eigenvalue, and if the graph is connected then all other eigenvalues are positive. The so-called Fiedler vector is the eigenvector associated to the smallest nonzero eigenvalue and can be used in heuristics for a number of graph manipulations

such as partitioning. One possible way of computing this vector with SLEPc is to instruct the eigensolver to search for the smallest eigenvalue (with `EPSSetWhichEigenpairs` or by using a spectral transformation as described in next chapter) but preventing it from computing the already known eigenvalue. For this, the user must provide a basis for the invariant subspace (in this case just vector $[1, 1, \ldots, 1]^T$) so that the eigensolver can *deflate* this subspace. This process is very similar to what eigensolvers normally do with invariant subspaces associated to eigenvalues as they converge. In other words, when a deflation space has been specified, the eigensolver works with the restriction of the problem to the orthogonal complement of this subspace.

The following function can be used to provide the `EPS` object with some basis vectors corresponding to a subspace that should be deflated during the solution process.

```
EPSAttachDeflationSpace(EPS eps,int n,Vec *ds,PetscTruth ortho)
```

The value `n` indicates how many vectors are passed in argument `ds`. This function can be called several times. The last parameter indicates whether all the provided vectors are known to be mutually orthonormal or not. If not, they are explicitly orthonormalized internally.

The deflation space can be any subspace but typically it is most useful in the case of an invariant subspace or a null-space. In any case, SLEPc internally checks to see if all (or part of) the provided subspace is a null-space of the associated linear system (see section 3.4.1). In this case, this null-space is passed to the linear solver (see PETSc's function `KSPSetNullSpace`) to enable the solution of singular systems. In practice, this allows the computation of eigenvalues of singular pencils (i.e. when A and B share a common null-space).

2.6.3 Orthogonalization

Internally, eigensolvers in `EPS` often need to orthogonalize a vector against a set of vectors (for instance, when building an orthonormal basis of a Krylov subspace). This operation is carried out typically by a Gram-Schmidt orthogonalization procedure. The user is able to adjust several options related to this algorithm, although the default behavior is good for most cases. This topic is covered in detail in [STR-1].

Chapter 3

ST: Spectral Transformation

The Spectral Transformation (ST) is the SLEPc object that encapsulates the functionality required for acceleration techniques based on the transformation of the spectrum. All the eigensolvers provided in EPS work by applying an operator to a set of vectors and this operator can adopt different forms. The ST object handles all the different possibilities in a uniform way, so that the solver can proceed without knowing which transformation has been selected. The type of spectral transformation can be specified at run time, as well as several parameters such as the value of the shift.

3.1 General Description

Spectral transformations are powerful tools for adjusting the way in which eigensolvers behave when coping with a problem. The general strategy consists in transforming the original problem into a new one in which eigenvalues are mapped to a new position while eigenvectors remain unchanged. These transformations can be used with several goals in mind:

- Compute internal eigenvalues. In some applications, the eigenpairs of interest are not the extreme ones (largest magnitude, smallest magnitude, rightmost, leftmost), but those contained in a certain interval or those closest to a certain value of the complex plane.
- Accelerate convergence. Convergence properties typically depend on how close the eigenvalues are from each other. With some spectral transformations, difficult eigenvalue distributions can be remapped in a more favorable way in terms of convergence.
- Handle some special situations. For instance, in generalized problems when matrix B is singular, it may be necessary to use a spectral transformation.

SLEPc separates spectral transformations from solution methods so that any combination of them can be specified by the user. To achieve this, all the eigensolvers contained in EPS must be implemented in such a way that they are independent of which transformation has been selected by the user. That is, the solver algorithm has to work with a generic operator, whose actual form depends on the transformation used. After convergence, eigenvalues are transformed back appropriately.

For technical details of the transformations described in this chapter, the interested user is referred to [Ericsson and Ruhe, 1980], [Scott, 1982], [Nour-Omid *et al.*, 1987], and [Meerbergen *et al.*, 1994].

3.2 Basic Usage

The ST module is the analogue to some PETSc modules such as PC. The user does not usually need to create a stand-alone ST object explicitly. Instead, every EPS object internally sets up an associated ST. Therefore, the usual object management methods such as STCreate, STDestroy, STView, STSetFromOptions, are not usually called by the user.

Although the ST context is hidden inside the EPS object, the user still has control over all the options, by means of the command line, or also inside the program. To allow application programmers to set any of the spectral transformation options directly within the code, the following routine is provided to extract the ST context from the EPS object,

```
EPSGetST(EPS eps,ST *st);
```

After this, one is able to set any options associated to the ST object. For example, to set the value of the shift, the following function is available

```
STSetShift(ST st,PetscScalar shift);
```

This can also be done with the command line option `-st_shift <shift>`. Note that the argument shift is defined as a PetscScalar, and this means that complex shifts are not allowed unless the complex version of SLEPc is used.

Other object operations are available, which are not usually called by the user. The most important of such functions are STApply, which applies the operator to a vector, and STSetUp, which prepares all the necessary data structures before the solution process starts. The term "operator" refers to one of A, $B^{-1}A$, $A + \sigma I$, ... depending on which kind of spectral transformation is being used.

3.3 Available Transformations

This section describes the spectral transformations that are provided in SLEPc. As in the case of eigensolvers, the spectral transformation to be used can be specified procedurally or via the command line. The application programmer can set it by means of the command

```
STSetType(ST st,STType type);
```

Spectral Transformation	STType	Options Name	Operator
Shift of Origin	STSHIFT	shift	$B^{-1}A + \sigma I$
Spectrum Folding	STFOLD	fold	$(B^{-1}A - \sigma I)^2$
Shift-and-invert	STSINV	sinvert	$(A - \sigma B)^{-1}B$
Cayley	STCAYLEY	cayley	$(A - \sigma B)^{-1}(A + \tau B)$
Shell Transformation	STSHELL	shell	*user-defined*

Table 3.1: Spectral transformations available in the ST package.

ST	Choice of σ, τ	Standard problem	Generalized problem
shift	$\sigma = 0$	A	$B^{-1}A$
	$\sigma \neq 0$	$A + \sigma I$	$B^{-1}A + \sigma I$
fold	$\sigma = 0$	A^2	$(B^{-1}A)^2$
	$\sigma \neq 0$	$(A - \sigma I)^2$	$(B^{-1}A - \sigma I)^2$
sinvert	$\sigma = 0$	A^{-1}	$A^{-1}B$
	$\sigma \neq 0$	$(A - \sigma I)^{-1}$	$(A - \sigma B)^{-1}B$
cayley	$\sigma \neq 0, \tau = 0$	$(A - \sigma I)^{-1}A$	$(A - \sigma B)^{-1}A$
	$\sigma = 0, \tau \neq 0$	$I + \tau A^{-1}$	$I + \tau A^{-1}B$
	$\sigma \neq 0, \tau \neq 0$	$(A - \sigma I)^{-1}(A + \tau I)$	$(A - \sigma B)^{-1}(A + \tau B)$

Table 3.2: Operators used in each spectral transformation mode.

where type can be one of STSHIFT, STFOLD, STSINV, STCAYLEY or STSHELL. The ST type can also be set with the command-line option -st_type followed by the name of the method (see table 3.1). The first four spectral transformations are described in detail in the rest of this section. The last possibility, STSHELL, uses a specific, application-provided spectral transformation. Section 5.3 describes how to implement one of these transformations.

The last column of Table 3.1 shows a general form of the operator used in each case. This generic operator can adopt different particular forms depending on whether the eigenproblem is standard or generalized, or whether the value of the shift (σ) and anti-shift (τ) is zero or not. All the possible combinations are illustrated in table 3.2.

The expressions shown in table 3.2 are not built explicitly. Instead, the appropriate operations are carried out when applying the operator to a certain vector. The inverses imply the solution of a linear system of equations that is managed by setting up an associated KSP object. The user can control the behavior of this object by adjusting the appropriate options, as will be illustrated with examples in section 3.4.1.

3.3.1 Shift of Origin

By default, no spectral transformation is performed. This is equivalent to a shift of origin (STSHIFT) with $\sigma = 0$, that is, the first line of table 3.2. The solver works with the original

expressions of the eigenvalue problems,

$$Ax = \lambda x \ , \tag{3.1}$$

for standard problems, and $Ax = \lambda Bx$ for generalized ones. Note that this last equation is actually treated internally as

$$B^{-1}Ax = \lambda x \ . \tag{3.2}$$

When the eigensolver in `EPS` requests the application of the operator to a vector, a matrix-vector multiplication by matrix A is carried out (in the standard case) or a matrix-vector multiplication by matrix A followed by a linear system solve with coefficient matrix B (in the generalized case). Note that in the last case, the operation will fail if matrix B is singular.

When the shift, σ, is given a value different from the default, 0, the effect is to move the whole spectrum by that exact quantity, σ, which is called *shift of origin*. To achieve this, the solver works with the shifted matrix, that is, the expressions it has to cope with are

$$(A + \sigma I)x = \theta x \ , \tag{3.3}$$

for standard problems, and

$$(B^{-1}A + \sigma I)x = \theta x \ , \tag{3.4}$$

for generalized ones. The important property that is used is that shifting does not alter the eigenvectors and that it does change the eigenvalues in a simple known way, it shifts them by σ. In both the standard and the generalized problems, the following relation holds

$$\theta = \lambda + \sigma \ . \tag{3.5}$$

This means that after the solution process, the value σ has to be subtracted from the computed eigenvalues, θ, in order to retrieve the solution of the original problem, λ. This is done by means of the function `STBackTransform`, which does not need to be called directly by the user.

3.3.2 Spectrum Folding

Spectrum folding refers to a spectral transformation that involves squaring in addition to shifting. The transformed problems to be addressed are the following

$$(A - \sigma I)^2 x = \theta x \ , \tag{3.6}$$

for standard problems, and

$$(B^{-1}A - \sigma I)^2 x = \theta x \ , \tag{3.7}$$

for generalized ones. In both cases, the following relation holds

$$\theta = (\lambda - \sigma)^2 \ . \tag{3.8}$$

The effect of this transformation is that the spectrum is folded around the value of σ. Thus, eigenvalues that are closest to the shift become the smallest eigenvalues in the folded spectrum,

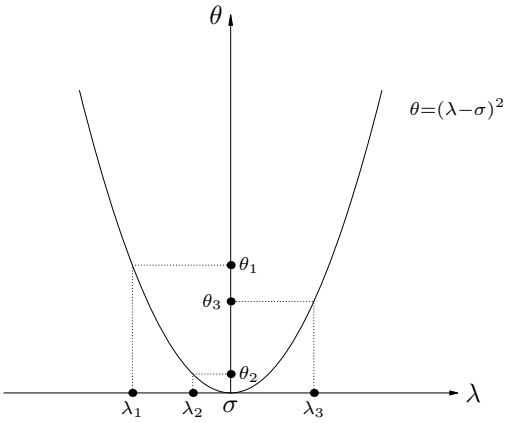

Figure 3.1: Illustration of the effect of spectrum folding.

as illustrated in Figure 3.1 for an example with real eigenvalues. For this reason, spectrum folding is commonly used in combination with eigensolvers that compute the smallest eigenvalues, for instance in the context of electronic structure calculations, [Canning *et al.*, 2000]. This transformation can be an effective, low-cost alternative to shift-and-invert (explained below).

Warning: It is possible that some eigenpairs have a very large associated error when using this transformation. There is a simple explanation for this: since the sign is lost when squaring, there is no way to determine if the original eigenvalue is located to the left or to the right of σ. As a consequence, the eigenvalue returned by `STBackTransform` is computed as $\lambda = \sqrt{\theta} + \sigma$, always taking the positive sign for the square root. This guess is wrong for values located on the left, which will have a large error even when the eigenvector is correct. This behavior can be changed by the user with the following function

```
STFoldSetLeftSide(ST st,PetscTruth left);
```

(or with `-st_fold_leftside`), so that the negative sign is taken for the square root. In this case those eigenvalues located on the left side of σ will be returned correctly but not the right ones.

3.3.3 Shift-and-invert

The shift-and-invert spectral transformation (`STSINV`) is used to enhance convergence of eigenvalues in the neighborhood of a given value. In this case, the solver deals with the expressions

$$(A - \sigma I)^{-1} x = \theta x \quad , \tag{3.9}$$

$$(A - \sigma B)^{-1} B x = \theta x \quad , \tag{3.10}$$

for standard and generalized problems, respectively. This transformation is effective for finding eigenvalues near σ since the eigenvalues θ of the operator that are largest in magnitude corre-

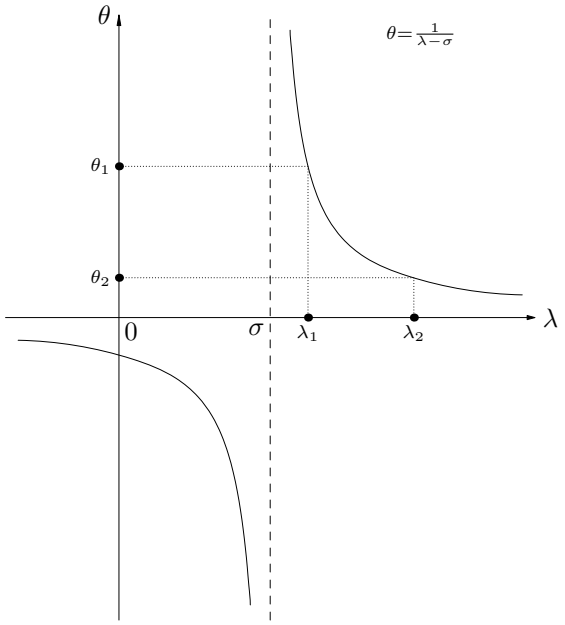

Figure 3.2: The shift-and-invert spectral transformation.

spond to the eigenvalues λ of the original problem that are closest to the shift σ in absolute value, as illustrated in Figure 3.2 for an example with real eigenvalues. Once the wanted eigenvalues have been found, they may be transformed back to eigenvalues of the original problem. Again, the eigenvectors remain unchanged. In this case, the relation between the eigenvalues of both problems is

$$\theta = 1/(\lambda - \sigma) \quad . \tag{3.11}$$

Therefore, after the solution process, the operation to be performed in function `STBackTransform` is $\lambda = \sigma + 1/\theta$ for each of the computed eigenvalues.

3.3.4 Cayley

The generalized Cayley transform (`STCAYLEY`) is defined from the expressions

$$(A - \sigma I)^{-1}(A + \tau I)x = \theta x \quad , \tag{3.12}$$

$$(A - \sigma B)^{-1}(A + \tau B)x = \theta x \quad , \tag{3.13}$$

for standard and generalized problems, respectively. Sometimes, the term Cayley transform is applied for the particular case in which $\tau = \sigma$. This is the default if τ is not given a value explicitly. The value of τ (the anti-shift) can be set with the following function

```
STCayleySetAntishift(ST st,PetscScalar tau);
```

or in the command line with `-st_antishift`.

This transformation is mathematically equivalent to shift-and-invert and, therefore, it is effective for finding eigenvalues near σ as well. However, in some situations it is numerically advantageous with respect to shift-and-invert (see [Bai *et al.*, 2000, §11.2], [Lehoucq and Salinger, 2001]).

In this case, the relation between the eigenvalues of both problems is

$$\theta = (\lambda + \tau)/(\lambda - \sigma) \quad . \tag{3.14}$$

Therefore, after the solution process, the operation to be performed in function `STBackTransform` is $\lambda = (\theta\sigma + \tau)/(\theta - 1)$ for each of the computed eigenvalues.

3.4 Advanced Usage

Using the `ST` object is very straightforward. However, when using spectral transformations many things are happening behind the scenes, mainly the solution of linear systems of equations. The user must be aware of what is going on in each case, so that it is possible to guide the solution process in the most beneficial way. This section describes several advanced aspects that can have a considerable impact on efficiency.

3.4.1 Solution of Linear Systems

In many of the cases shown in table 3.2, the operator contains an inverted matrix, which means that a linear system of equations must be solved whenever the application of the operator to a vector is required. These cases are handled internally by means of a `KSP` object.

In the simplest case, a generalized problem is to be solved with a zero shift. A sample command line could be

```
$ program -eps_type subspace -eps_tol 1e-6 -eps_monitor
```

In this case, assuming that the program solves a generalized problem, the `ST` object associated to the `EPS` solver creates a `KSP` object whose coefficient matrix is B. This `KSP` object will be set with the default values, that is, GMRES with ILU preconditioning (see the PETSc documentation for details).

The default values corresponding to the `KSP` object can be modified via the command line. For instance,

```
$ program -eps_type subspace -eps_tol 1e-6 -eps_monitor
          -st_ksp_type cg -st_pc_type jacobi -st_ksp_rtol 1e-5
```

specifies some additional options for the solution of this linear system. In particular, this example selects the CG solver with Jacobi preconditioning and a relative tolerance of 10^{-5}. The `-st_` prefix signifies that the option corresponds to the linear solver within `ST`.

If an iterative method is used for the linear system solves, usually a slightly more stringent tolerance must be required of the linear solves relative to the desired accuracy of the eigenvalue calculation. It is also possible to select any of the direct linear solvers available in PETSc. In this case, the factorization is only carried out at the beginning of the eigenvalue calculation and this cost is amortized in each subsequent application of the operator. This is also the case for iterative methods with preconditioners with high-cost set-up such as ILU.

The application programmer is able to set the desired linear systems solver options also from within the code. In order to do this, first the context of the `KSP` object must be retrieved with the following function

```
STGetKSP(ST st,KSP *ksp);
```

The above functionality is also applicable to the other spectral transformations. In this other example, the spectrum is shifted by $\sigma = 0.5$ and several options are specified for the linear systems

```
$ program -st_type shift -st_shift 0.5 -st_ksp_type cgs -st_pc_factor_levels 1
```

Similarly, for the shift-and-invert technique with $\sigma = 10$:

```
$ program -st_type sinvert -st_shift 10 -st_pc_type jacobi
```

The shift-and-invert and Cayley transformations deserve special consideration. In these cases, the coefficient matrix is not a simple matrix but an expression that can be explicitly constructed or not, depending on the user's choice. This issue is examined in detail in section 3.4.2 below.

In many cases, especially if a shift-and-invert or Cayley transformation is being used, iterative methods may not be well suited for solving linear systems (because of the properties of the coefficient matrix that can be indefinite and badly conditioned). In such cases, during the execution of the application, the user may get the following message:

```
[0]PETSC ERROR: Warning: KSP did not converge (-3)!
```

If this happens, chances are that the `EPS` object fails to compute the eigensolution or that the retrieved solution is wrong whatsoever. In that situation, it is necessary to use a direct method for solving the linear systems. See the PETSc documentation for a list of available possibilities. The simplest one is to use the LU decomposition as in the following example:

```
$ program -st_type sinvert -st_ksp_type preonly -st_pc_type lu
```

3.4.2 Explicit Computation of Coefficient Matrix

Three possibilities can be distinguished regarding the form of the coefficient matrix of the linear systems of equations associated to the different spectral transformations. The possible coefficient matrices are:

- Simple: B.

- Shifted: $A - \sigma I$.
- Axpy: $A - \sigma B$.

The first case has already been described and presents no difficulty. In the other two cases, there are three possible approaches:

"`shell`" To work with the corresponding expression without forming the matrix explicitly. This is achieved by internally setting a matrix-free matrix with `MatCreateShell`.

"`inplace`" To build the coefficient matrix explicitly. This is done by means of a `MatShift` or a `MatAXPY` operation, which overwrites matrix A with the corresponding expression. This alteration of matrix A is reversed after the eigensolution process has finished.

"`copy`" To build the matrix explicitly, as in the previous option, but using a working copy of the matrix, that is, without modifying the original matrix A.

The default behavior is to build the coefficient matrix explicitly in a copy of A (option "`copy`"). The user can change this as in the following example

```
$ program -st_type sinvert -st_shift 10 -st_pc_type jacobi -st_matmode shell
```

As always, the procedural equivalent is also available for specifying this option in the code of the program:

```
STSetMatMode(ST st,STMatMode mode);
```

The user must consider which approach is the most appropriate for the particular application. The different options have advantages and drawbacks. The first approach is the simplest one but severely restricts the number of possibilities available for solving the system, in particular most of the PETSc preconditioners would not be available, including direct methods. The only preconditioners that can be used in this case are Jacobi (only if matrices A and B have the operation `MATOP_GET_DIAGONAL`) or a user-defined one.

The second approach ("`inplace`") can be much faster, specially in the generalized case. A more important advantage of this approach is that, in this case, the linear system solver can be combined with any of the preconditioners available in PETSc, including those which need to access internal matrix data-structures such as ILU. The main drawback is that, in the generalized problem, this approach probably makes sense only in the case that A and B have the same sparse pattern, because otherwise the function `MatAXPY` can be very inefficient. If the user knows that the pattern is the same (or a subset), then this can be specified with the function

```
STSetMatStructure(ST st,MatStructure str);
```

Note that when the value of the shift σ is very close to an eigenvalue, then the linear system will be ill-conditioned and using iterative methods may be problematic. On the other hand, in symmetric definite problems, the coefficient matrix will be indefinite whenever σ is a point

in the interior of the spectrum and in that case it is not possible to use a symmetric definite factorization (Cholesky or ICC).

The third approach ("`copy`") uses more memory but avoids a potential problem that could appear in the "`inplace`" approach: the recovered matrix might be slightly different from the original one (due to roundoff).

3.4.3 Preserving the Symmetry in Generalized Eigenproblems

As mentioned in section 2.3, some eigensolvers can exploit symmetry and compute a solution for Hermitian problems with less storage and/or computational cost than other methods. Also, symmetric solvers can be more accurate in some cases. However, in the case of generalized eigenvalue problems in which both A and B are symmetric, it happens that, due to the spectral transformation, symmetry is lost since none of the transformed operators $B^{-1}A + \sigma I$, $(A - \sigma B)^{-1}B$, etc. is symmetric (the same applies in the Hermitian case for complex matrices).

The solution proposed in SLEPc is based on selecting different kinds of inner products. Currently, we have the following choice of inner products:

- Standard Hermitian inner product: $\langle x, y\rangle = x^* y$.
- B-inner product: $\langle x, y\rangle_B = x^* B\, y$.

The second one can be used for preserving the symmetry in symmetric definite generalized problems, as described below. Note that $\langle x, y\rangle_B$ is a genuine inner product only if B is symmetric positive definite (for the case of symmetric positive semi-definite B see subsection 3.4.4).

It can be shown that $\mathbb{R}^n$ with the $\langle x, y\rangle_B$ inner product is isomorphic to the Euclidean n-space $\mathbb{R}^n$ with the standard Hermitian inner product. This means that if we use $\langle x, y\rangle_B$ instead of the standard inner product, we are just changing the way lengths and angles are measured, but otherwise all the algebraic properties are maintained and therefore algorithms remain correct. What is interesting to observe is that the transformed operators such as $B^{-1}A$ or $(A - \sigma B)^{-1}B$ are self-adjoint with respect to $\langle x, y\rangle_B$.

Internally, SLEPc operates with the abstraction illustrated in figure 3.3. The operations indicated by dashed arrows are implemented as virtual functions: `IPInnerProduct` and `STApply`. From the user point of view, all the above explanation is transparent. The only thing he/she has to care about is to set the problem type appropriately with `EPSSetProblemType` (see section 2.3). In the case of the Cayley transform, SLEPc is using $\langle x, y\rangle_{A+\tau B}$ as the inner product for preserving symmetry.

Using the B-inner product may be attractive also in the non-symmetric case (A non-symmetric) as described in the next subsection.

3.4.4 Purification of Eigenvectors

In generalized eigenproblems, the case of singular B deserves especial consideration. Note that in this case the default spectral transformation (`STSHIFT`) cannot be used since B^{-1} does not exist.

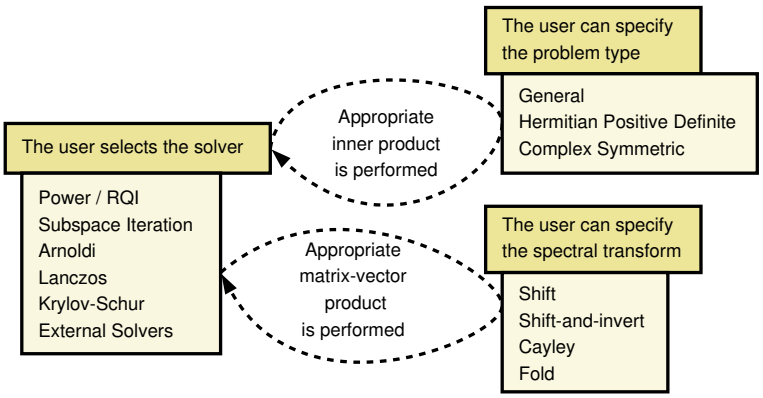

Figure 3.3: Abstraction used by SLEPc solvers.

In shift-and-invert with operator matrix $T = (A - \sigma B)^{-1}B$, when B is singular all the eigenvectors that belong to finite eigenvalues are also eigenvectors of T and belong to the range of T, $\mathcal{R}(T)$. In this case, the bilinear function $\langle x, y\rangle_B$ introduced in subsection 3.4.3 is a semi-inner product and $\|x\|_B = \sqrt{\langle x, x\rangle_B}$ is a semi-norm. As before, T is self-adjoint with respect to this inner product since $BT = T^*B$. Also, $\langle x, y\rangle_B$ is a true inner product on $\mathcal{R}(T)$.

The implication of all this is that, for singular B, if the B-inner product is used throughout the eigensolver then, assuming that the initial vector has been forced to lie in $\mathcal{R}(T)$, the computed eigenvectors should be correct, i.e. they should belong to $\mathcal{R}(T)$ as well. Nevertheless, finite precision arithmetic spoils this nice picture, and computed eigenvectors are easily corrupted by components of vectors in the null-space of B. Additional computation is required for achieving the desired property. This is usually referred to as *eigenvector purification*.

Although more elaborate purification strategies have been proposed (usually trying to reduce the computational effort, see [Nour-Omid *et al.*, 1987] and [Meerbergen and Spence, 1997]), the approach in SLEPc is simply to explicitly force the initial vector in the range of B, with $v_0 = Tv_0$, as well as the computed eigenvectors at the end, $x_i = Tx_i$.

A final comment is that eigenvector corruption happens also in the non-symmetric case. If A is non-symmetric but B is symmetric positive semi-definite, then the scheme presented above (B-inner product together with purification) can still be applied and is generally more successful than the straightforward approach with the standard inner product. For using this scheme in SLEPc, the user has to specify the special problem type `EPS_PGNHEP`, see table 2.1.

Chapter 4

SVD: Singular Value Decomposition

The Singular Value Decomposition (`SVD`) solver object can be used for computing a partial SVD of a rectangular matrix. It provides uniform and efficient access to several specific SVD solvers included in SLEPc, and also gives the possibility to compute the decomposition via the eigensolvers provided in the `EPS` package.

In many aspects, the user interface of `SVD` resembles that of `EPS`. For this reason, this chapter and chapter 2 have a very similar structure.

4.1 The Singular Value Decomposition

In this section, some basic concepts about the singular value decomposition are presented. The objective is to set up the notation and also to justify some of the solution approaches, particularly those based on the `EPS` object. As in the case of eigensolvers, some of the implemented methods are described in detail in the SLEPc technical reports.

For background material about the SVD, see for instance [Bai *et al.*, 2000, ch. 6]. Many other books such as [Björck, 1996] or [Hansen, 1998] present the SVD from the perspective of its application to the solution of least squares problems and other related linear algebra problems.

The singular value decomposition (SVD) of an $m \times n$ matrix A can be written as

$$A = U\Sigma V^*, \tag{4.1}$$

where $U = [u_1, \ldots, u_m]$ is an $m \times m$ unitary matrix ($U^*U = I$), $V = [v_1, \ldots, v_n]$ is an $n \times n$ unitary matrix ($V^*V = I$), and Σ is an $m \times n$ diagonal matrix with diagonal entries $\Sigma_{ii} = \sigma_i$ for $i = 1, \ldots, \min\{m, n\}$. If A is real, U and V are real and orthogonal. The vectors u_i are called the left singular vectors, the v_i are the right singular vectors, and the σ_i are the singular values.

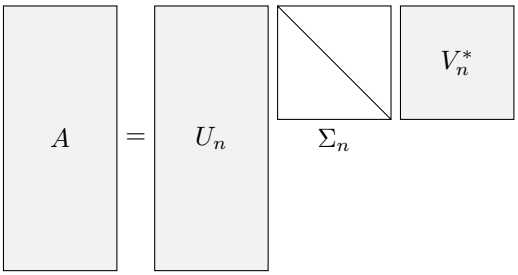

Figure 4.1: Scheme of the thin SVD of a rectangular matrix A.

In the following, we will assume that $m \geq n$. If $m < n$ then A should be replaced by A^* (note that in SLEPc this is done transparently as described later in this chapter and the user need not worry about this). In the case that $m \geq n$, the top n rows of Σ contain $\text{diag}(\sigma_1, \ldots, \sigma_n)$ and its bottom $m - n$ rows are zero. The relation 4.1 may also be written as $AV = U\Sigma$, or

$$Av_i = u_i\sigma_i \,, \quad i = 1, \ldots, n \,, \tag{4.2}$$

and also as $A^*U = V\Sigma^*$, or

$$A^*u_i = v_i\sigma_i \,, \quad i = 1, \ldots, n \,, \tag{4.3}$$

$$A^*u_i = 0 \,, \quad i = n+1, \ldots, m \,. \tag{4.4}$$

The last left singular vectors corresponding to Eq. 4.4 are often not computed, especially if $m \gg n$. In that case, the resulting factorization is sometimes called the *thin* SVD, $A = U_n\Sigma_n V_n^*$, and is depicted in Figure 4.1. This factorization can also be written as

$$A = \sum_{i=1}^{n} \sigma_i u_i v_i^* . \tag{4.5}$$

Each (σ_i, u_i, v_i) is called a singular triplet.

The singular values are real and nonnegative, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_n = 0$, where $r = \text{rank}(A)$. It can be shown that $\{u_1, \ldots, u_r\}$ span the range of A, $\mathcal{R}(A)$, whereas $\{v_{r+1}, \ldots, v_n\}$ span the null space of A, $\mathcal{N}(A)$.

If the zero singular values are dropped from the sum in Eq. 4.5, the resulting factorization, $A = \sum_{i=1}^{r} \sigma_i u_i v_i^*$, is called the *compact* SVD, $A = U_r\Sigma_r V_r^*$.

In the case of a very large and sparse A, it is usual to compute only a subset of $k \leq r$ singular triplets. We will refer to this decomposition as the *truncated* SVD of A. It can be shown that the matrix $A_k = U_k\Sigma_k V_k^*$ is the best rank-k approximation to matrix A, in the least squares sense.

In general, one can take an arbitrary subset of the summands in Eq. 4.5, and the resulting factorization is called the *partial* SVD of A. As described later in this chapter, SLEPc allows the computation of a partial SVD corresponding to either the k largest or smallest singular triplets.

Equivalent Eigenvalue Problems. It is possible to formulate the problem of computing the singular triplets of a matrix A as an eigenvalue problem involving a Hermitian matrix related to A. There are two possible ways of achieving this:

1. With the *cross product* matrix, either A^*A or AA^*.
2. With the *cyclic* matrix, $H(A) = \left[\begin{smallmatrix} 0 & A \\ A^* & 0 \end{smallmatrix}\right]$.

In SLEPc, the computation of the SVD is always based on one of these two alternatives, either by passing one of these matrices to an `EPS` object or by performing the computation implicitly.

By pre-multiplying Eq. 4.2 by A^* and then using Eq. 4.3, the following relation results

$$A^*Av_i = \sigma_i^2 v_i \ , \tag{4.6}$$

that is, the v_i are the eigenvectors of matrix A^*A with corresponding eigenvalues equal to σ_i^2. Note that after computing v_i the corresponding left singular vector, u_i, is readily available through Eq. 4.2 with just a matrix-vector product, $u_i = \frac{1}{\sigma_i}Av_i$.

Alternatively, one could compute first the left vectors and then the right ones. For this, pre-multiply Eq. 4.3 by A and then use Eq. 4.2 to get

$$AA^*u_i = \sigma_i^2 u_i \ . \tag{4.7}$$

In this case, the right singular vectors are obtained as $v_i = \frac{1}{\sigma_i}A^*u_i$.

The two approaches represented in Eqs. 4.6 and 4.7 are very similar. Note however that A^*A is a square matrix of order n whereas AA^* is of order m. In cases where $m \gg n$, the computational effort will favor the A^*A approach. On the other hand, the eigenproblem 4.6 has $n - r$ zero eigenvalues and the eigenproblem 4.7 has $m - r$ zero eigenvalues. Therefore, continuing with the assumption that $m \geq n$, even in the full rank case the AA^* approach may have a large null space resulting in difficulties if the smallest singular values are sought. In SLEPc, this will be referred to as the cross product approach and will use whichever matrix is smaller, either A^*A or AA^*.

Computing the SVD via the cross product approach may be adequate for determining the largest singular triplets of A, but the loss of accuracy can be severe for the smallest singular triplets. The cyclic matrix approach is an alternative that avoids this problem, but at the expense of significantly increasing the cost of the computation. Consider the eigendecomposition of

$$H(A) = \begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix} \ , \tag{4.8}$$

which is a Hermitian matrix of order $(m+n)$. It can be shown that $\pm\sigma_i$ is a pair of eigenvalues of $H(A)$ for $i = 1, \ldots, r$ and the other $m + n - 2r$ eigenvalues are zero. The unit eigenvectors associated to $\pm\sigma_i$ are $\frac{1}{\sqrt{2}}\left[\begin{smallmatrix} \pm u_i \\ v_i \end{smallmatrix}\right]$. Thus it is possible to extract the singular values and the left and right singular vectors of A directly from the eigenvalues and eigenvectors of $H(A)$. Note that in this case singular values are not squared, and therefore the computed values will be more accurate. The drawback in this case is that small eigenvalues are located in the interior of the spectrum.

```
SVD       svd;       /*  SVD solver context  */
Mat       A;         /*  problem matrix      */
Vec       u, v;      /*  singular vectors    */
PetscReal sigma;     /*  singular value      */
int       j, nconv;
PetscReal error;

SVDCreate( PETSC_COMM_WORLD, &svd );
SVDSetOperator( svd, A );
SVDSetFromOptions( svd );
SVDSolve( svd );
SVDGetConverged( svd, &nconv );
for (j=0; j<nconv; j++) {
  SVDGetSingularTriplet( svd, j, &sigma, u, v );
  SVDComputeRelativeError( svd, j, &error );
}
SVDDestroy( svd );
```

Figure 4.2: Example code for basic solution with `SVD`.

4.2 Basic Usage

From the perspective of the user interface, the `SVD` package is very similar to `EPS`, with some differences that will be highlighted shortly.

The basic steps for computing a partial SVD with SLEPc are illustrated in Figure 4.2. The steps are more or less the same as those described in chapter 2 for the eigenvalue problem. First, the solver context is created with `SVDCreate`. Then the problem matrix has to be specified with `SVDSetOperator`. Then, a call to `SVDSolve` invokes the actual solver. After that, `SVDGetConverged` is used to determine how many solutions have been computed, which are retrieved with `SVDGetSingularTriplet`. Finally, `SVDDestroy` cleans up everything.

If one compares this example code with the `EPS` example in Figure 2.1, the most outstanding differences are the following:

- The singular value is a `PetscReal`, not a `PetscScalar`.
- Each singular vector is defined with a single `Vec` object, not two as was the case for eigenvectors.
- Function `SVDSetOperator` only admits one `Mat` argument.
- There is no equivalent to `EPSSetProblemType`.

The reason for the last two differences is that SLEPc does not currently support different kinds of SVD problems. This may change in future versions if some generalization of the SVD such as the GSVD is added.

4.3 Defining the Problem

Defining the problem consists in specifying the problem matrix, A, and the portion of the spectrum to be computed. In the case of the SVD, the number of possibilities will be much more limited than in the case of eigenproblems.

The problem matrix is provided with the following function

```
SVDSetOperator(SVD svd,Mat A);
```

where `A` can be any matrix, not necessarily square, stored in any allowed PETSc format including the matrix-free mechanism (see section 5.2 for a detailed discussion).

It is important to note that all SVD solvers in SLEPc make use of both A and A^*, as suggested by the description in section 4.1. A^* is not explicitly passed as an argument to `SVDSetOperator`, therefore it will have to stem from A. There are two possibilities for this: either A is transposed explicitly and A^* is created as a distinct matrix, or A^* is handled implicitly via `MatMultTranspose` operations whenever a matrix-vector product is required in the algorithm. The default is to build A^* explicitly, but this behavior can be changed with

```
SVDSetTransposeMode(SVD svd,SVDTransposeMode mode);
```

In section 4.1, it was mentioned that in SLEPc the cross product approach chooses the smallest of the two possible cases A^*A or AA^*, that is, A^*A is used if A is a tall, thin matrix ($m \geq n$), and AA^* is used if A is a fat, short matrix ($m < n$). In fact, what SLEPc does internally is that if $m < n$ the roles of A and A^* are reversed. This is equivalent to transposing all the SVD factorization, so left singular vectors become right singular vectors and vice versa. This is actually done in all singular value solvers, not only the cross product approach. The objective is to simplify the number of cases to be treated internally by SLEPc, as well as to reduce the computational cost in some situations. Note that this is done transparently and the user need not worry about transposing the matrix, only to indicate how the transpose has to be handled, as explained above.

The user can specify how many singular values and vectors to compute. The default is to compute only one singular triplet. The function

```
SVDSetDimensions(EPS eps,int nsv,int ncv);
```

allows the specification of the number of singular values to compute, `nsv`. The last argument can be set to prescribe the number of column vectors to be used by the solution algorithm, `ncv`, that is, the largest dimension of the working subspace. These two parameters can also be set at run time with the options `-svd_nsv` and `-svd_ncv`. For example, the command line

```
$ program -svd_nsv 10 -svd_ncv 24
```

requests 10 singular values and instructs to use 24 column vectors. Note that `ncv` must be at least equal to `nsv`, although in general it is recommended (depending on the method) to work with a larger subspace, for instance $\texttt{ncv} \geq 2 \cdot \texttt{nsv}$ or even more.

SVDWhich	Command line key	Sorting criterion
SVD_LARGEST	-svd_largest	Largest σ
SVD_SMALLEST	-svd_smallest	Smallest σ

Table 4.1: Available possibilities for selection of the singular values of interest.

For the selection of the portion of the spectrum of interest, there are only two possibilities in the case of SVD: largest and smallest singular values, see Table 4.1. The default is to compute the largest ones, but this can be changed with

```
SVDSetWhichSingularTriplets(SVD svd,SVDWhich which);
```

which can also be specified at the command line. This criterion is used both for configuring how the algorithm seeks singular values and also for sorting the computed values. In contrast to the case of EPS, computing singular values located in the interior part of the spectrum is difficult, the only possibility is to use an EPS object combined with a spectral transformation (this possibility is explained in detail in the next section). Note that in this case, the value of which applies to the transformed spectrum.

4.4 Selecting the SVD Solver

The available methods for computing the partial SVD are shown in Table 4.2. These methods can be classified in the following three groups:

- Solvers based on EPS. These solvers set up an EPS object internally, thus using the available eigensolvers for solving the SVD problem. The two possible approaches in this case are the cross product matrix and the cyclic matrix, as described in section 4.1.
- Specific SVD solvers. These are typically eigensolvers that have been adapted algorithmically to exploit the structure of the SVD problem. There are currently two solvers in this category: Lanczos and thick-restart Lanczos. A detailed description of these methods can be found in the SLEPc Technical Reports.
- The LAPACK solver. This is an interface to some LAPACK routines, analog to those in the case of eigenproblems. These routines operate in dense mode with only one processor and therefore are suitable only for moderate size problems. This solver should be used only for debugging purposes.

The default solver is the one that uses the cross product matrix (cross), usually the fastest and most memory-efficient approach. See a more detailed explanation below.

The solution method can be specified procedurally or via the command line. The application programmer can set it by means of the command

```
SVDSetType(SVD svd,SVDType method);
```

Method	SVDType	Options Database Name
Cross Product	SVDCROSS	cross
Cyclic Matrix	SVDCYCLIC	cyclic
Lanczos	SVDLANCZOS	lanczos
Thick-restart Lanczos	SVDTRLANCZOS	trlanczos
LAPACK solver	SVDLAPACK	lapack

Table 4.2: List of solvers available in the SVD module.

while the user writes the options database command `-svd_type` followed by the name of the method (see table 4.2).

The EPS-based solvers deserve some additional comments. These SVD solvers work by creating an EPS object internally and setting up an eigenproblem of type EPS_HEP. These solvers implement the cross product matrix approach, Eq. 4.6, and the cyclic matrix approach, Eq. 4.8. Therefore, the operator matrix associated to the EPS object will be A^*A in the case of the cross solver and $H(A)$ in the case of the cyclic solver.

In the case of the cross solver, the matrix A^*A is not built explicitly, since sparsity would be lost. Instead, a shell matrix is created internally in the SVD object and passed to the EPS object. In the case of the cyclic solver, the situation is different since $H(A)$ is still a sparse matrix. SLEPc gives the possibility to handle it implicitly as a shell matrix (the default), or to create $H(A)$ explicitly, that is, storing its elements in a distinct matrix. The function for setting this option is

```
SVDCyclicSetExplicitMatrix(SVD svd,PetscTruth explicit);
```

The EPS object associated to the cross and cyclic SVD solvers is created with a set of reasonable default parameters. However, it may sometimes be necessary to change some of the EPS options such as the eigensolver. To allow application programmers to set any of the EPS options directly within the code, the following routines are provided to extract the EPS context from the SVD object,

```
SVDCrossGetEPS(SVD svd,EPS *eps);
SVDCyclicGetEPS(SVD svd,EPS *eps);
```

A more convenient way of changing EPS options is through the command-line. This is achieved simply by prefixing the EPS options with `-svd_` as in the following example:

```
$ program -svd_type cross -svd_eps_type lanczos
```

At this point, one may consider changing also the options of the ST object associated to the EPS object in cross and cyclic SVD solvers, for example to compute singular values located at the interior of the spectrum via a shift-and-invert transformation. This is indeed possible, but some considerations have to be taken into account. When A^*A or $H(A)$ are managed as shell matrices, then the potential of the spectral transformation is limited seriously, because

some of the required operations will not be defined (this is discussed briefly in sections 5.2 and 3.4.2). Therefore, computing interior singular values is more likely to be successful if using the `cyclic` solver with explicit $H(A)$ matrix. To illustrate this, here is a complicated command-line example for computing singular values close to 12.0:

```
$ program -svd_type cyclic -svd_cyclic_explicitmatrix -svd_st_type sinvert
          -svd_st_shift 12.0 -svd_st_ksp_type preonly -svd_st_pc_type lu
```

4.5 Retrieving the Solution

Once the call to `SVDSolve` is complete, all the data associated to the computed partial SVD is kept internally in the `SVD` object. This information can be obtained by the calling program by means of a set of functions described below.

As in the case of eigenproblems, the number of computed singular triplets depends on the convergence and, therefore, it may be different from the number of solutions requested by the user. So the first task is to find out how many solutions are available, with

```
SVDGetConverged(SVD svd,int *nconv);
```

Usually, the number of converged solutions, `nconv`, will be equal to `nsv`, but in general it will be a number ranging from 0 to `ncv` (here, `nsv` and `ncv` are the arguments of function `SVDSetDimensions`).

Normally, the user is interested in the singular values only, or the complete singular triplets. The function

```
SVDGetSingularTriplet(SVD svd, int j, PetscReal *sigma, Vec u, Vec v);
```

returns the j-th computed singular triplet, (σ_j, u_j, v_j), where both u_j and v_j are normalized to have unit norm. Typically, this function is called inside a loop for each value of `j` from 0 to `nconv`–1. Note that singular values are ordered according to the same criterion specified with function `SVDSetWhichSingularTriplets` for selecting the portion of the spectrum of interest.

In some applications, it may be enough to compute only the right singular vectors. This is especially important in cases in which memory requirements are critical (remember that both U_k and V_k are dense matrices, and U_k may require much more storage than V_k, see Figure 4.1). In SLEPc, there is no general option for specifying this, but the default behavior of some solvers is to compute only right vectors and allocate/compute left vectors only in the case that the user requests them. This is done in the `cross` solver and in some special variants of other solvers such as one-sided Lanczos (consult the SLEPc technical reports for specific solver options).

Reliability of the Computed Solution. In SVD computations, a-posteriori error bounds are much the same as in the case of Hermitian eigenproblems, due to the equivalence discussed in section 4.1. The residual vector is defined in terms of the cyclic matrix, $H(A)$, so its norm is

$$\|r\|_2 = \left(\|A\tilde{v} - \tilde{\sigma}\tilde{u}\|_2^2 + \|A^*\tilde{u} - \tilde{\sigma}\tilde{v}\|_2^2\right)^{\frac{1}{2}} / \left(\|\tilde{u}\|_2^2 + \|\tilde{v}\|_2^2\right)^{\frac{1}{2}} \quad , \tag{4.9}$$

where $\tilde{\sigma}$, $\tilde{u}$ and $\tilde{v}$ represent any of the `nconv` computed singular triplets delivered by `SVDGetSingularTriplet`.

Given the above definition, the following relation holds

$$|\sigma - \tilde{\sigma}| \leq \|r\|_2 \quad , \tag{4.10}$$

where σ is an exact singular value. The following SLEPc function

```
SVDComputeResidualNorms(SVD svd,int j,PetscReal *norm1,PetscReal *norm2)
```

computes the two partial 2-norms separately, $\|A\tilde{v} - \tilde{\sigma}\tilde{u}\|_2$ and $\|A^*\tilde{u} - \tilde{\sigma}\tilde{v}\|_2$. The relative error can be obtained with the following function:

```
SVDComputeRelativeError(SVD svd,int j,PetscReal *error);
```

Controlling and Monitoring Convergence. Similarly to the case of eigensolvers, in `SVD` the number of iterations carried out by the solver can be determined with `SVDGetIterationNumber`, and the tolerance and maximum number of iterations can be set with `SVDSetTolerances`. Also, convergence can be monitored with command-line keys `-svd_monitor` or `-svd_monitor_draw`. See section 2.5.3 for additional details.

Chapter 5

Additional Information

This chapter contains miscellaneous information as a complement to the previous chapters, which can be regarded as less important.

5.1 Supported PETSc Features

SLEPc relies on PETSc for all the features that are not directly related to eigenvalue problems. All the functionality associated to vectors and matrices as well as linear systems of equations is provided by PETSc. Also, low level details are inherited directly from PETSc. In particular, the parallelism within SLEPc methods is handled completely by PETSc's vector and matrix modules.

SLEPc only contains high level objects, as depicted in figure 1.1. These object classes have been designed and implemented following the philosophy of other high level objects in PETSc. In this way, SLEPc benefits from a number of PETSc's good properties such as the following (see PETSc users guide for details):

- Portability and scalability in a wide range of platforms. Different architecture builds can coexist in the same installation. Where available, dynamic libraries are used to reduce disk space of executable files.
- Support for profiling of programs:
  - Display performance statistics with `-log_summary`, including also SLEPc's objects. The collected data are *flops* and execution times as well as information about parallel performance, for individual subroutines and the possibility of user-defined stages.
  - Event logging, including user-defined events.
  - Direct wall-clock timing with `PetscGetTime`.

  - Display detailed profile information and trace of events.

- Convergence monitoring, both textual and graphical.
- Support for debugging of programs:
  - Debugger startup and attachment of parallel processes.
  - Automatic generation of back-traces of the call stack.
  - Detection of memory leaks.
- A number of viewers for visualization of data, including built-in graphics capabilities that allow for sparse pattern visualization, graphic convergence monitoring, operator's spectrum visualization and other user-defined operations.
- Easy handling of runtime options.

5.2 Supported Matrix Types

Methods implemented in `EPS` merely require vector operations and matrix-vector products. In PETSc, mathematical objects such as vectors and matrices have an interface that is independent of the underlying data structures. SLEPc manipulates vectors and matrices via this interface and, therefore, it can be used with any of the matrix representations provided by PETSc, including dense, sparse, block-diagonal and symmetric formats, either sequential or parallel.

The above statement must be reconsidered when using `EPS` in combination with `ST`. As explained in chapter 3, in many cases the operator associated to a spectral transformation not only consists in pure matrix-vector products but also other operations may be required as well, most notably a linear system solve (see table 3.2). In this case, the limitation is that there must be support for the requested operation for the selected matrix representation. For instance, if one wants to use `cholesky` for the solution of the linear systems, then it may be necessary to work with a symmetric matrix format such as `MATSEQSBAIJ`.

Shell Matrices. In many applications, the matrices that define the eigenvalue problem are not available explicitly. Instead, the user knows a way of applying these matrices to a vector.

An intermediate case is when the matrices have some block structure and the different blocks are stored separately. There are numerous situations in which this occurs, such as the discretization of equations with a mixed finite-element scheme. An example is the eigenproblem arising in the stability analysis associated with Stokes problems,

$$\begin{bmatrix} A & C \\ C^* & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} = \lambda \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} \quad , \tag{5.1}$$

where x and p denote the velocity and pressure fields. Similar formulations also appear in many other situations, such as the quadratic eigenvalue problem, see Eq. 2.9.

Many of these problems can be solved by reformulating them as a reduced-order standard or generalized eigensystem, in which the matrices are equal to certain operations of the blocks. These matrices are not computed explicitly to avoid losing sparsity.

All these cases can be easily handled in SLEPc by means of shell matrices. These are matrices that do not require explicit storage of the component values. Instead, the user must provide subroutines for all the necessary matrix operations, typically only the application of the linear operator to a vector.

Shell matrices, also called matrix-free matrices, are created in PETSc with the command `MatCreateShell`. Then, the function `MatShellSetOperation` is used to provide any user-defined shell matrix operations (see the PETSc documentation for additional details). Several examples are available in SLEPc that illustrate how to solve a matrix-free eigenvalue problem.

In the simplest case, defining matrix-vector product operations (`MATOP_MULT`) is enough for using `EPS` with shell matrices. However, in the case of generalized problems, if matrix B is also a shell matrix then it may be necessary to define other operations in order to be able to solve the linear system successfully, for example `MATOP_GET_DIAGONAL` to use Jacobi preconditioning. On the other hand, if the shift-and-invert `ST` is to be used, then in addition it may also be necessary to define `MATOP_SHIFT` or `MATOP_AXPY` (see section 3.4.2 for discussion).

In the case of `SVD`, both A and A^* are required to solve the problem. So when computing the SVD, the shell matrix needs to have the `MATOP_MULT_TRANSPOSE` operation in addition to `MATOP_MULT`. Alternatively, if A^* is to be built explicitly, `MATOP_TRANSPOSE` is then the required operation (for details see the manual page for `SVDSetTransposeMode`).

5.3 Extending SLEPc

Shell matrices are a simple mechanism of extensibility, in the sense that the package is extended with new user-defined matrix objects. Once the new matrix has been defined, it can be used by SLEPc in the same way as the rest of the matrices as long as the required operations are provided.

A similar mechanism is available in SLEPc also for extending the system incorporating new spectral transformations (`ST`). This is done by using the `STSHELL` spectral transformation type, in a similar way as shell matrices or shell preconditioners. In this case, the user defines how the operator is applied to a vector and optionally how the computed eigenvalues are transformed back to the solution of the original problem (see section 5.3 for details). This tool is intended for simple spectral transformations. For more sophisticated transformations, the user should register a new `ST` type (see below).

At least, user-defined spectral transformations have to define how the operator is to be applied to a vector. Optionally, they can also specify the way in which computed eigenvalues must be transformed back to the solution of the original eigenproblem. An example program is provided in the SLEPc distribution in order to illustrate the use of shell transformations.

The function

```
STShellSetApply(ST,int(*)(void*,Vec,Vec),void*);
```

has to be invoked after the creation of the ST object in order to provide a routine that applies the operator to a vector. And the function

```
STShellSetBackTransform(ST,int(*)(void*,PetscScalar*,PetscScalar*));
```

can be used optionally to specify the routine for the back-transformation of eigenvalues. The two functions provided by the user receive a pointer to a user-defined context that can contain any useful information. This context must be passed as the last argument in the call to STShellSetApply.

Finally, the application programmer can use the following function

```
STShellSetName(ST,char*);
```

to specify a name for the new shell transformation in order to identify it in the program's output (STView).

SLEPc further supports extensibility by allowing application programmers to code their own subroutines for unimplemented features such as new eigensolvers or new spectral transformations. It is possible to register these new methods to the system and use them as the rest of standard subroutines. For example, to implement a variant of the Subspace Iteration method, one could copy the SLEPc code associated to the subspace solver, modify it and register a new EPS type with the following line of code

```
EPSRegister("newsubspace",0,"EPSCreate_NEWSUB",EPSCreate_NEWSUB);
```

After this call, the new solver could be used in the same way as the rest of SLEPc solvers. For instance,

```
$ program -eps_type newsubspace
```

EPSRegister can be used to register new types whose code is linked into the executable. To register types in a dynamic library use EPSRegisterDynamic.

A similar mechanism is available for registering new types of classes ST and SVD.

5.4 Directory Structure

The directory structure of the SLEPc software is very similar to that in PETSc. The root directory of SLEPc contains the following directories:

bmake - Base SLEPc makefile directory. Includes subdirectories for each value of PETSC_ARCH.

config - SLEPc configuration files.

docs - All documentation for SLEPc, including this manual. The subdirectory manualpages contains the on-line manual pages of each SLEPc routine.

include - All include files for SLEPc that are public.

`include/finclude` - SLEPc include files for Fortran programmers using the .F suffix.

`lib` - Location of all the generated libraries for each value of `PETSC_ARCH`.

`src` - The source code for all SLEPc components, which currently includes:

- `sys` - general system-related routines.
- `eps` - eigenvalue problem solver.
- `st` - spectral transformation.
- `svd` - singular value decomposition solver.
- `ip` - inner product, for developer use only.
- `examples` - example programs.
- `mat/examples` - matrices used by some examples.

Each SLEPc source code component directory has the following subdirectories:

`interface` - The calling sequences for the abstract interface to the components. Code here does not know about particular implementations.

`impls` - Source code for one or more implementations.

5.5 Wrappers to External Libraries

SLEPc interfaces to several external libraries for the solution of eigenvalue problems. This section provides a short description of each of these packages as well as some hints for using them with SLEPc, including pointers to the respective websites from which the software can be downloaded. The description may also include method-specific parameters, that can be set in the same way as other SLEPc options, either procedurally or via the command-line.

SLEPc uses a configuration and makefile system very similar to that of PETSc. All platform specific setting are taken directly from the PETSc installation. In order to use SLEPc together with an external library such as ARPACK, this external software must be enabled during configuration of SLEPc.

To use these eigensolvers, one needs to do the following.

1. Install the external software, with the same compilers and MPI that will be used for PETSc/SLEPc.

2. Enable the utilization of the external software from SLEPc by adding specific command-line parameters when executing `config/configure.py`. For example, to use ARPACK, specify the following options (with the appropriate paths):

```
config/configure.py --with-arpack-dir=/usr/software/ARPACK
                    --with-arpack-flags=-lparpack,-larpack
```

3. Build the SLEPc libraries.

4. Use the runtime option `-eps_type <type>` to select the solver.

Exceptions to the above rule are LAPACK and BLOPEX, which should be enabled during PETSc's configuration.

LAPACK

References. [Anderson *et al.*, 1992].

Website. http://www.netlib.org/lapack.

Version. 3.0 or later.

Summary. LAPACK (Linear Algebra PACKage) is a software package for the solution of many different dense linear algebra problems, including various types of eigenvalue problems and singular value decompositions.

SLEPc explicitly creates the operator matrix in dense form and then the appropriate LAPACK driver routine is invoked. Therefore, this interface should be used only for testing and validation purposes and not in a production code. The operator matrix is created by applying the operator to the columns of the identity matrix.

Installation. The SLEPc interface to LAPACK can be used directly.

ARPACK

References. [Lehoucq *et al.*, 1998], [Maschhoff and Sorensen, 1996].

Website. http://www.caam.rice.edu/software/ARPACK.

Version. Release 2 (plus patches).

Summary. ARPACK (ARnoldi PACKage) is a software package for the computation of a few eigenvalues and corresponding eigenvectors of a general $n \times n$ matrix A. It is most appropriate for large sparse or structured matrices, where structured means that a matrix-vector product $w \leftarrow Av$ requires order n rather than the usual order n^2 floating point operations.

ARPACK is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). These variants may be viewed as a synthesis of the Arnoldi/Lanczos process with the Implicitly Shifted QR technique that is suitable for large scale problems.

It can be used for standard and generalized eigenvalue problems, both in real and complex arithmetic. It is implemented in Fortran 77 and it is based on the reverse communication interface. A parallel version, PARPACK, is available with support for both MPI and BLACS.

Installation. First of all, unpack `arpack96.tar.gz` and also the patch file `patch.tar.gz`. If ARPACK is to be used with more than one processor, then it is necessary to uncompress also the contents of the file `parpack96.tar.gz` together with the patches `ppatch.tar.gz`. Make sure you delete any `mpif.h` files that could exist in the directory tree. After setting all the directories, modify the `ARmake.inc` file and then compile the software with `make all`. It is recommended that ARPACK is installed with its own LAPACK version since it may give unexpected results with more recent versions of LAPACK.

PRIMME

References. [Stathopoulos, 2007].

Website. http://www.cs.wm.edu/~andreas/software.

Version. 1.1.

Summary. PRIMME (PReconditioned Iterative MultiMethod Eigensolver) is a C library for finding a number of eigenvalues and their corresponding eigenvectors of a real symmetric (or complex Hermitian) matrix. This library provides a multimethod eigensolver, based on Davidson/Jacobi-Davidson. Particular methods include GD+1, JDQMR, and LOBPCG. It supports preconditioning as well as the computation of interior eigenvalues.

Installation. Type `make lib` after customizing the file `Make_flags` appropriately.

Specific options. The SLEPc interface to this package allows the user to specify the maximum allowed block size with the function `EPSPRIMMESetBlockSize` or at run time with the option `-eps_primme_block_size <size>`.

For changing the particular algorithm within PRIMME, use the function `EPSPRIMMESetMethod`. Other options related to the method are the use of preconditioning (with function `EPSPRIMMESetPrecond`) and the restarting strategy (`EPSPRIMMESetRestart`).

BLZPACK

References. [Marques, 1995].

Website. http://crd.lbl.gov/~osni/#Software.

Version. 04/00.

Summary. BLZPACK (Block LancZos PACKage) is a standard Fortran 77 implementation of the block Lanczos algorithm intended for the solution of the standard eigenvalue problem $Ax = \mu x$ or the generalized eigenvalue problem $Ax = \mu Bx$, where A and B are real, sparse symmetric matrices. The development of this eigensolver was motivated by the need to solve large, sparse, generalized problems from free vibration analysis in structural engineering. Several upgrades were performed afterwards aiming at the solution of eigenvalue problems from a wider range of applications.

BLZPACK uses a combination of partial and selective re-orthogonalization strategies. It can be run in either sequential or parallel mode, by means of MPI or PVM interfaces, and it uses the reverse communication strategy.

Installation. For the compilation of the `libblzpack.a` library, first check the appropriate architecture file in the directory `sys/MACROS` and then type `creator -mpi`.

Specific options. The SLEPc interface to this package allows the user to specify the block size with the function `EPSBlzpackSetBlockSize` or at run time with the option `-eps_blzpack_block_size <size>`. Also, the function `EPSBlzpackSetNSteps` can be used to set the maximum number of steps per run (also with `-eps_blzpack_nsteps`).

For the spectrum slicing feature, SLEPc allows the programmer to provide the computational interval with the option `-eps_blzpack_interval`, or with the function `EPSBlzpackSetInterval` in the program source.

TRLAN

References. [Wu and Simon, 2001].

Website. http://crd.lbl.gov/~kewu/trlan.html.

Summary. This package provides a Fortran 90 implementation of the dynamic thick-restart Lanczos algorithm. This is a specialized version of Lanczos that targets only the case in which one wants both eigenvalues and eigenvectors of a large real symmetric eigenvalue problem that cannot use the shift-and-invert scheme. In this case the standard non-restarted Lanczos algorithm requires to store a large number of Lanczos vectors, what can cause storage problems and make each iteration of the method very expensive.

TRLAN requires the user to provide a matrix-vector multiplication routine. The parallel version uses MPI as the message passing layer.

Installation. To install this package, it is necessary to have access to a Fortran 90 compiler. The compiler name and the options used are specified in the file called `Make.inc`. To generate the library, type `make libtrlan_mpi.a` in the `TRLan` directory.

BLOPEX

References. [Knyazev, 2001].

Website. http://www-math.cudenver.edu/~aknyazev/software/BLOPEX.

Summary. BLOPEX is a package that implements the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method for computing several extreme eigenpairs of symmetric positive generalized eigenproblems. Numerical comparisons suggest that this method is a genuine analog for eigenproblems of the standard preconditioned conjugate gradient method for symmetric linear systems.

Installation. In order to use BLOPEX from SLEPc, it is a prior requirement that PETSc has been configured with `./config/configure.py --download-hypre --download-blopex`.

5.6 Fortran Interface

SLEPc provides an interface for Fortran 77 programmers, very much like PETSc. As in the case of PETSc, there are slight differences between the C and Fortran SLEPc interfaces, due to differences in Fortran syntax. For instance, the error checking variable is the final argument of all the routines in the Fortran interface, in contrast to the C convention of providing the error variable as the routine's return value.

The following code is a sample program written in Fortran 77. It is the Fortran equivalent of the program given in section 1.4.1 and can be found in `${SLEPC_DIR}/src/examples/ex1f.F`.

```
!  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     SLEPc - Scalable Library for Eigenvalue Problem Computations
!     Copyright (c) 2002-2007, Universidad Politecnica de Valencia, Spain
!
!     This file is part of SLEPc. See the README file for conditions of use
!     and additional information.
!  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Program usage: mpirun -np n ex1f [-help] [-n <n>] [all SLEPc options]
!
!  Description: Simple example that solves an eigensystem with the EPS object.
!  The standard symmetric eigenvalue problem to be solved corresponds to the
!  Laplacian operator in 1 dimension.
!
!  The command line options are:
!    -n <n>, where <n> = number of grid points = matrix size
!
! ----------------------------------------------------------------------
!
      program main
      implicit none

#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscmat.h"
#include "finclude/slepc.h"
#include "finclude/slepceps.h"

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Declarations
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     A     operator matrix
!     eps   eigenproblem solver context

      Mat          A
      EPS          eps
      EPSType      type
      PetscReal    tol, error
      PetscScalar  kr, ki
      integer      rank, n, nev, ierr, maxit, i, its, nconv
      integer      col(3), Istart, Iend
      PetscTruth   flg
      PetscScalar  value(3)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
50
      call SlepcInitialize(PETSC_NULL_CHARACTER,ierr)
      call MPI_Comm_rank(PETSC_COMM_WORLD,rank,ierr)
      n = 30
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)
55
      if (rank .eq. 0) then
        write(*,100) n
      endif
 100  format (/'1-D Laplacian Eigenproblem, n =',I3,' (Fortran)')
60
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Compute the operator matrix that defines the eigensystem, Ax=kx
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

65    call MatCreate(PETSC_COMM_WORLD,A,ierr)
      call MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n,ierr)
      call MatSetFromOptions(A,ierr)

      call MatGetOwnershipRange(A,Istart,Iend,ierr)
70    if (Istart .eq. 0) then
        i = 0
        col(1) = 0
        col(2) = 1
        value(1) =  2.0
75      value(2) = -1.0
        call MatSetValues(A,1,i,2,col,value,INSERT_VALUES,ierr)
        Istart = Istart+1
      endif
      if (Iend .eq. n) then
80      i = n-1
        col(1) = n-2
        col(2) = n-1
        value(1) = -1.0
        value(2) =  2.0
85      call MatSetValues(A,1,i,2,col,value,INSERT_VALUES,ierr)
        Iend = Iend-1
      endif
      value(1) = -1.0
      value(2) =  2.0
90    value(3) = -1.0
      do i=Istart,Iend-1
        col(1) = i-1
        col(2) = i
        col(3) = i+1
95      call MatSetValues(A,1,i,3,col,value,INSERT_VALUES,ierr)
      enddo

      call MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY,ierr)
      call MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY,ierr)
100
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Create the eigensolver and display info
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

105 !     ** Create eigensolver context
      call EPSCreate(PETSC_COMM_WORLD,eps,ierr)

    !     ** Set operators. In this case, it is a standard eigenvalue problem
      call EPSSetOperators(eps,A,PETSC_NULL_OBJECT,ierr)
110   call EPSSetProblemType(eps,EPS_HEP,ierr)

    !     ** Set solver parameters at runtime
```

```
      call EPSSetFromOptions(eps,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Solve the eigensystem
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call EPSSolve(eps,ierr)
      call EPSGetIterationNumber(eps,its,ierr)
      if (rank .eq. 0) then
        write(*,110) its
      endif
 110  format (/' Number of iterations of the method:',I4)

!     ** Optional: Get some information from the solver and display it
      call EPSGetType(eps,type,ierr)
      if (rank .eq. 0) then
        write(*,120) type
      endif
 120  format (' Solution method: ',A)
      call EPSGetDimensions(eps,nev,PETSC_NULL_INTEGER,ierr)
      if (rank .eq. 0) then
        write(*,130) nev
      endif
 130  format (' Number of requested eigenvalues:',I2)
      call EPSGetTolerances(eps,tol,maxit,ierr)
      if (rank .eq. 0) then
        write(*,140) tol, maxit
      endif
 140  format (' Stopping condition: tol=',1P,E10.4,', maxit=',I4)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Display solution and clean up
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!     ** Get number of converged eigenpairs
      call EPSGetConverged(eps,nconv,ierr)
      if (rank .eq. 0) then
        write(*,150) nconv
      endif
 150  format (' Number of converged eigenpairs:',I2/)

!     ** Display eigenvalues and relative errors
      if (nconv.gt.0) then
        if (rank .eq. 0) then
          write(*,*) '         k          ||Ax-kx||/||kx||'
          write(*,*) ' ----------------- ------------------'
        endif
        do i=0,nconv-1
!         ** Get converged eigenpairs: i-th eigenvalue is stored in kr
!         ** (real part) and ki (imaginary part)
          call EPSGetEigenpair(eps,i,kr,ki,PETSC_NULL,PETSC_NULL,ierr)

!         ** Compute the relative error associated to each eigenpair
          call EPSComputeRelativeError(eps,i,error,ierr)
          if (rank .eq. 0) then
            write(*,160) PetscRealPart(kr), error
          endif
 160      format (1P,'   ',E12.4,'       ',E12.4)

        enddo
        if (rank .eq. 0) then
          write(*,*)
        endif
```

```
      endif

!     ** Free work space
      call EPSDestroy(eps,ierr)
      call MatDestroy(A,ierr)

      call SlepcFinalize(ierr)
      end
```

Bibliography

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1992). *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Bai, Z., J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst (eds.) (2000). *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Balay, S., K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang (2007). PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.3.3, Argonne National Laboratory.

Balay, S., W. D. Gropp, L. C. McInnes, and B. F. Smith (1997). Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing* (edited by E. Arge, A. M. Bruaset, and H. P. Langtangen), pp. 163–202. Birkhaüser.

Björck, Å. (1996). *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, Philadelphia.

Canning, A., L. W. Wang, A. Williamson, and A. Zunger (2000). Parallel Empirical Pseudopotential Electronic Structure Calculations for Million Atom Systems. *J. Comput. Phys.*, 160(1):29–41.

Ericsson, T. and A. Ruhe (1980). The Spectral Transformation Lanczos Method for the Numerical Solution of Large Sparse Generalized Symmetric Eigenvalue Problems. *Math. Comp.*, 35(152):1251–1268.

Golub, G. H. and H. A. van der Vorst (2000). Eigenvalue Computation in the 20th Century. *J. Comput. Appl. Math.*, 123(1-2):35–65.

Hansen, P. C. (1998). *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Knyazev, A. V. (2001). Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. *SIAM J. Sci. Statist. Comput.*, 23(2):517–541.

Lehoucq, R. B. and A. G. Salinger (2001). Large-Scale Eigenvalue Calculations for Stability Analysis of Steady Flows on Massively Parallel Computers. *International Journal for Numerical Methods in Fluids*, 36:309–327.

Lehoucq, R. B., D. C. Sorensen, and C. Yang (1998). *ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Marques, O. A. (1995). BLZPACK: Description and User's Guide. Technical Report TR/PA/95/30, CERFACS, Toulouse, France.

Maschhoff, K. J. and D. C. Sorensen (1996). PARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures. *Lect. Notes Comp. Sci.*, 1184:478–486.

Meerbergen, K. and A. Spence (1997). Implicitly Restarted Arnoldi with Purification for the Shift-Invert Transformation. *Math. Comp.*, 66(218):667–689.

Meerbergen, K., A. Spence, and D. Roose (1994). Shift-invert and Cayley Transforms for Detection of Rightmost Eigenvalues of Nonsymmetric Matrices. *BIT*, 34(3):409–423.

MPI Forum (1994). MPI: a Message-Passing Interface Standard. *Int. J. Supercomp. Applic. High Perf. Comp.*, 8(3/4):159–416.

Nour-Omid, B., B. N. Parlett, T. Ericsson, and P. S. Jensen (1987). How to Implement the Spectral Transformation. *Math. Comp.*, 48(178):663–673.

Parlett, B. N. (1980). *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, NJ. Reissued with revisions by SIAM, Philadelphia, 1998.

Saad, Y. (1992). *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. John Wiley and Sons, New York.

Scott, D. S. (1982). The Advantages of Inverted Operators in Rayleigh-Ritz Approximations. *SIAM J. Sci. Statist. Comput.*, 3(1):68–75.

Stathopoulos, A. (2007). Nearly Optimal Preconditioned Methods for Hermitian Eigenproblems under Limited Memory. Part I: Seeking One Eigenvalue. *SIAM J. Sci. Comput.*, 29(2):481–514.

Stewart, G. W. (2001). *Matrix Algorithms. Volume II: Eigensystems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Wu, K. and H. Simon (2001). Thick-Restart Lanczos Method for Large Symmetric Eigenvalue Problems. *SIAM J. Matrix Anal. Appl.*, 22(2):602–616.

SLEPc Technical Reports (Note: these reports are available through the SLEPc web site.)

[STR-1] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Orthogonalization Routines in SLEPc."

[STR-2] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Single Vector Iteration Methods in SLEPc."

[STR-3] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Subspace Iteration in SLEPc."

[STR-4] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Arnoldi Methods in SLEPc."

[STR-5] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Lanczos Methods in SLEPc."

[STR-6] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "A Survey of Software for Sparse Eigenvalue Problems."

[STR-7] V. Hernández, J. E. Román, A. Tomás, V. Vidal. "Krylov-Schur Methods in SLEPc."

[STR-8] V. Hernández, J. E. Román, A. Tomás. "Restarted Lanczos Bidiagonalization for the SVD in SLEPc."

Index