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 2.2.1
August, 2004

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.

The emphasis of the software is on methods and techniques appropriate for problems in which the associated matrices are sparse, for example, those arising after the discretization of partial differential equations. Therefore, most of the methods offered by the library are projection methods or other methods with similar properties. Examples of these methods are Arnoldi, Lanczos and Subspace Iteration, to name a few. SLEPc implements these basic methods as well as more sophisticated algorithms. It also provides built-in support for spectral transformations such as shift-and-invert.

SLEPc is a general library in the sense that it covers standard and generalized eigenvalue problems, both Hermitian and non-Hermitian, with either real or complex arithmetic.

SLEPc is built on top of PETSc, the Portable, Extensible Toolkit for Scientific Computation [Balay *et al.*, 2004]. 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 which are not acquainted with PETSc are highly recommended to familiarize with it before proceeding with SLEPc.

This manual provides a general description of the package. In addition, manual pages for individual routines are included in the distribution in hypertext format.

SLEPc interfaces to some external software packages such as:

- ARPACK, `http://www.caam.rice.edu/software/ARPACK`.
- BLZPACK, `http://www.nersc.gov/~osni/#Software`.
- PLANSO, `http://www.nersc.gov/~kewu/planso.html`.
- TRLAN, `http://www.nersc.gov/~kewu/trlan.html`.

These are all optional packages and do not need to be installed to use SLEPc. See section B.4 for details.

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 this site.

Acknowledgements

We thank all the PETSc team for their help. We also thank Barry Smith, David Keyes, Osni Marques, Tony Drummond and Rich Lehoucq for supporting this project.

The development of the library has been partially funded by the Science and Technology Office of the Valencian Regional Government under grant number CTIDB/2002/54.

Contents

Chapter 1

Introduction

SLEPc, the *Scalable Library for Eigenvalue Problem Computations*, is a software package 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. There is a strong demand for solving problems associated with stability and vibrational analysis in practical applications, which 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, scientists keep on using traditional well-established techniques.

The reasons for this situation are manifold. 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.

A further obstacle appears when these methods have to be applied in a large software project developed by inter-disciplinary teams. In this scenery, libraries must be able to interoperate with already existing software and with other libraries, possibly written in a different programming language. In order to cope with the complexity associated with such large software projects, libraries must be designed carefully in order to overcome hurdles such as different storage formats. 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.

The SLEPc library is an attempt to address this complexity and provides a set of tools that can be used to obtain a solution in many applications. SLEPc is based on PETSc, the Portable, Extensible Toolkit for Scientific Computation [Balay *et al.*, 2004], and, therefore, a large percentage of the complexity is avoided since SLEPc relies on PETSc for all low level implementation details. SLEPc focuses on high level features structured around a few types of objects. It offers a growing number of solution methods as well as interfaces to integrate well-established eigenvalue packages such as ARPACK.

1.1 Getting Started

SLEPc is a general library for the solution of eigenvalue problems, in the sense that it covers standard and generalized eigenvalue problems, both Hermitian and non-Hermitian, with either real or complex arithmetic. This manual assumes that the reader is familiar with eigenvalue problems, their basic mathematical

properties and the basic techniques and methods to solve them. A brief introduction to the topic is included in section A.1. A nice introduction of eigenvalue problems and an overview of methods can be found in [Golub and van der Vorst, 2000].

The emphasis of SLEPc is on methods and techniques appropriate for problems in which the associated matrices are sparse, for example, those arising after the discretisation of partial differential equations. Therefore, most of the methods offered by the library are projection methods or other methods with similar properties. Examples of these methods are Arnoldi, Lanczos and Subspace Iteration, to name a few. A comprehensive description of state-of-the-art methods of this kind can be found in [Bai *et al.*, 2000]. SLEPc contains implementations of the basic methods as well as a growing number of more sophisticated algorithms.

The Portable, Extensible Toolkit for Scientific Computation (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. SLEPc is based on PETSc, and 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 which are not acquainted with PETSc are highly recommended to familiarize with it before proceeding with SLEPc. An introduction to PETSc is included in section A.2.

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.*, 2004].

The complete SLEPc distribution, users manual, manual pages, and additional information are available via the SLEPc home page at

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

The SLEPc home page also contains details regarding installation, new features and changes in recent versions of SLEPc, and more information.

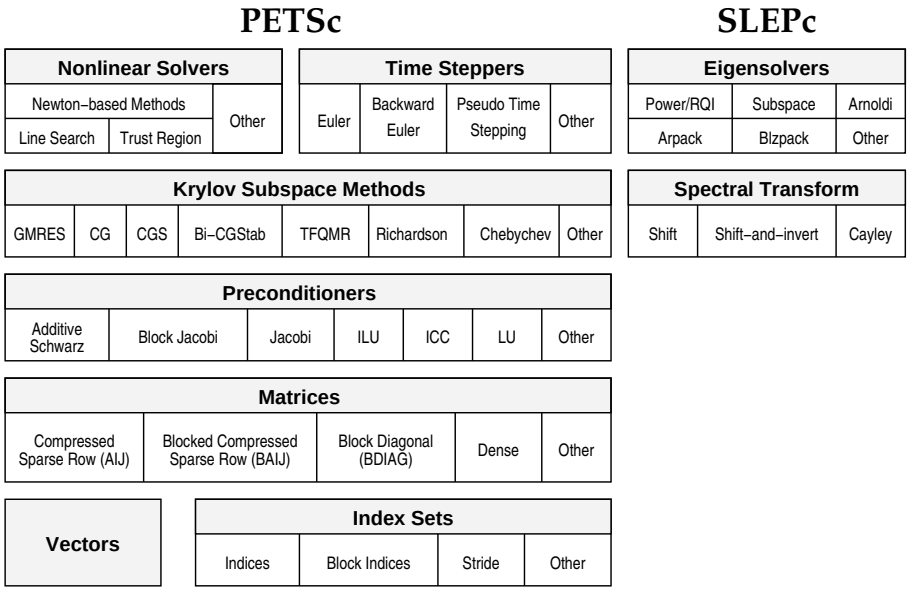

Figure 1.1: Numerical components of PETSc and SLEPc.

Within the SLEPc distribution, the directory `${SLEPC_DIR}/docs` contains all the documentation of the library. Manual pages for all SLEPc functions can be accessed on-line at `http://www.grycap.upv.es/slepc/document.htm`. These manual pages provide hyperlinked indices (organized by both concepts and routine names) to the source code and enable easy movement among related topics. The file `slepc.ps` contains the Postscript form of the SLEPc Users Manual (this document). A PDF version is also available.

Note to Fortran Programmers: As in the case of PETSc, 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. Section 4.3 provides a discussion of the differences between using SLEPc from Fortran and C, as well as complete Fortran examples.

1.2 Installation

This section gives an overview of the installation procedure. For full 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 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.

SLEPc version	PETSc version	Major	Status	Release date
2.1.0	2.1.0	⋆	Not released	-
2.1.1	2.1.1		Released	Dec 2002
	2.1.2			
	2.1.3			
2.1.5	2.1.5		Released	May 2003
	2.1.6			
2.2.0	2.2.0	⋆	Released	Apr 2004
2.2.1	2.2.1	⋆	Released	Aug 2004

Table 1.1: Correspondence between SLEPc and PETSc releases.

There are two possible ways of installing PETSc: automatically (with configure scripts) or manually. SLEPc does not support automatic installation yet. For a manual installation of PETSc, the user simply sets the environment variables `PETSC_DIR` and `PETSC_ARCH` and types `make`. Apart of this, some customization may be necessary, see the PETSc documentation for details.

The installation process for SLEPc is very similar. The main steps are described next. Note that prior to this steps, optional packages must have been installed. If any of these packages is installed afterwards, recompilation is necessary. Refer to `http://www.grycap.upv.es/slepc/install.htm` or to section B.4 for details about installation of some of these packages.

1. Unbundle the distribution file `slepc.tgz` with a usual command such as `gunzip -c slepc.tgz | tar xvf -`. 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-2.2.x
   ```

   In addition to this variable, `PETSC_DIR` and `PETSC_ARCH` must also be set correctly, the first one pointing to the PETSc home directory and the other containing the selected architecture.

4. Edit the file `${SLEPC_DIR}/bmake/${PETSC_ARCH}/packages` to indicate the local installation of optional software packages such as ARPACK. If there exists no directory named `bmake/${PETSC_ARCH}` for the value of `${PETSC_ARCH}` you are using, then create it similar to the existing ones.

5. In the SLEPc home directory, type

   ```
   make BOPT=g
   ```

   to build a debugging version of SLEPc, or

   ```
   make BOPT=O
   ```

   to build an optimized version of the SLEPc libraries. The flag `BOPT` determines what type of libraries are built (i.e., specifies compiler options). Other available alternatives are `BOPT=[g_complex,O_complex]` for complex numbers versions (see section 4.4).

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

   ```
   make BOPT=g slepc_testexamples >& examples_log
   ```

   Examine the file `examples_log` for any obvious errors or problems.

7. The Fortran libraries are built automatically during the installation outlined above. To compile and test the Fortran examples, use the command

   ```
   make BOPT=g slepc_testfortran >& fortran_log
   ```

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 two environment variables required by PETSc, that is, `PETSC_DIR`, to indicate the full path of the PETSc installation, and `PETSC_ARCH` to specify the architecture (e.g., `rs6000`, `solaris`, `IRIX`, etc.) on which PETSc is being used. The utility `${PETSC_DIR}/bin/petscarch` can be used for this purpose. For example,

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

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

All PETSc programs use the MPI (Message Passing Interface) standard for message-passing communication [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 processors:

```
mpirun -np 8 slepc_program [arguments]
```

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`. In Fortran the initialization command has the form

```
SlepcInitialize(character file,integer ierr)
```

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 processors, though most users rarely need to use these. 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 routines return an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For the C/C++ interface, the error variable is the routine's return value, while for the Fortran version, each SLEPc routine has as its final argument an integer error variable.

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

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

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

1.5 Simple SLEPc Example

To help the user start using SLEPc immediately, a simple example is listed next which 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.

```
static char help[] = "Solves a standard symmetric eigenproblem corresponding to the Laplacian operator in 1 dimension.\n\n"
  "The command line options are:\n\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;
  int         n=30, nev, ierr, maxit, i, its, nconv,
              col[3], Istart, Iend, FirstBlock=0, LastBlock=0;
  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,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);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);
```

```
   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
55                Create the eigensolver and set various options
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   /*
      Create eigensolver context
60 */
   ierr = EPSCreate(PETSC_COMM_WORLD,&eps);CHKERRQ(ierr);

   /*
      Set operators. In this case, it is a standard eigenvalue problem
65 */
   ierr = EPSSetOperators(eps,A,PETSC_NULL);CHKERRQ(ierr);
   ierr = EPSSetProblemType(eps,EPS_HEP);CHKERRQ(ierr);

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

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

   ierr = EPSSolve(eps);CHKERRQ(ierr);
   ierr = EPSGetIterationNumber(eps, &its);CHKERRQ(ierr);
80 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
   */
85 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);
90 ierr = EPSGetTolerances(eps,&tol,&maxit);CHKERRQ(ierr);
   ierr = PetscPrintf(PETSC_COMM_WORLD," Stopping condition: tol=%.4g, maxit=%d\n",tol,maxit);
         CHKERRQ(ierr);


95 /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                  Display solution and clean up
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   /*
100    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);
105
   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,PETSC_NULL,PETSC_NULL);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 %12f\n",re,im,error);CHKERRQ(ierr);
      } else {
        ierr = PetscPrintf(PETSC_COMM_WORLD,"   %12f       %12f\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 = 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). 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 the routine `MatCreate`

```
MatCreate(MPI_Comm comm,int m,int n,int M,int N,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);
EPSGetIterationNumber(EPS eps,int *its);
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 which enables the user to customize the solution process at runtime by selecting the solution algorithm and also specifying the convergence tolerance, setting various monitoring routines, etc.

Spectral Transformation

In the example program 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 PETSc macro `CHKERRQ(ierr)` checks the value of `ierr` and calls the PETSc error handler upon error detection. `CHKERRQ(ierr)` should be used in all subroutines to enable a complete error traceback. See the PETSc manual for full details.

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 at the SLEPc web site.

2. Copy the SLEPc example that corresponds to the class of problem of interest (e.g., singular value decomposition).

3. Copy the corresponding makefile within the example directory; compile and run the example program.

4. Use the example program as a starting point for developing a custom code.

1.6 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 various architectures.

`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 visible to the user.

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

`lib` - Location of all the generated libraries for each combination of `BOPT` and architecture.

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

- `sys` - general system-related routines.
- `eps` - eigenvalue problem solver.
- `st` - spectral transformation.
- `fortran` - Fortran interface stubs.
- `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.

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 General Description

The `EPS` module can be used to solve eigenvalue problems. In the standard formulation, the 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 eigenvector, respectively. Note that they can be complex even when the matrix is real. SLEPc can also solve eigenvalue

problems in generalized form,

$$Ax = \lambda Bx \ , \tag{2.2}$$

where $B \in \mathbb{C}^{n\times n}$.

The methods provided by SLEPc are appropriate for large sparse eigenproblems and typically only use matrix A in matrix-vector products of the form $w = Av$, or $w = B^{-1}Av$ in the generalized case. In these two cases, the matrices A and $B^{-1}A$, respectively, will be referred to as the *operator* matrix. Therefore, the implemented methods apply the operator to a set of vectors repeatedly until the approximations to the eigenpairs are sufficiently accurate. The operator can adopt yet other different forms if spectral transformations are used, as explained in chapter 3.

SLEPc assumes that only a subset of the eigenvalues must be computed. The user specifies how many of them and also in which part of the spectrum they are to be sought.

2.2 Basic Usage

The `EPS` module is used in a similar way as other 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`). 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 requested tolerance and the maximum number of iterations allowed. The user can also specify other things such as the orthogonalization technique or the portion of the spectrum of interest.

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 which invokes the subroutine for the selected eigensolver. `EPSGetConverged` can be used afterwards to determine how many of the requested eigenpairs have converged to working precision. `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

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

  EPSCreate( PETSC_COMM_WORLD, &eps );
  EPSSetOperators( eps, A, PETSC_NULL );
10 EPSSetProblemType( eps, EPS_NHEP );
  EPSSetFromOptions( eps );
  EPSSolve( eps );
  EPSGetIterationNumber( eps, &its );
  EPSGetConverged( eps, &nconv );
15 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.

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 only necessary change to the example code in order to solve a generalized problem is 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 4.1.

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

All the data associated to the solution of the eigenproblem is kept internally. The function

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

retrieves in the parameter `its` either the iteration number at which convergence was successfully reached, or the *negative* of the iteration at which a problem was detected. And the function

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

queries how many eigenpairs have converged to working precision. The solution of the eigenproblem is retrieved in line 16 with serveral calls to the following function

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

This function returns the j-th solution of the eigenproblem. `kr` and `ki` receive the real and imaginary parts of the eigenvalue, while `xi` and `xr` receive the real and imaginary parts of the associated eigenvector. Therefore, the j-th eigenvalue is `kr`$+i\cdot$`ki` and the j-th eigenvector is stored in the `Vec` objects `xr` and `xi`. [Note: see section 4.4 for a detailed discussion of this issue.]

In line 17 of the example the relative residual error $\|Ax_j - \lambda_j Bx_j\|/\|\lambda_j x_j\|$ associated to the j-th eigenpair is computed with a call to

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

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);
```

Problem Type	`EPSProblemType`	Command line key
Hermitian	`EPS_HEP`	`-eps_hermitian`
Generalized Hermitian	`EPS_GHEP`	`-eps_gen_hermitian`
Non-Hermitian	`EPS_NHEP`	`-eps_non_hermitian`
Generalized Non-Hermitian	`EPS_GNHEP`	`-eps_gen_non_hermitian`

Table 2.1: Problem types considered in `EPS`.

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.

The options database key `-eps_plot_eigs` instructs SLEPc to plot the computed approximations of the eigenvalues in an X display after the solution process.

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., $A = A^H$) or, equivalently in the case of real matrices, if matrix A is symmetric (i.e., $A = A^T$). A generalized eigenproblem is Hermitian if if matrix A is Hermitian (symmetric) and matrix B is Hermitian (symmetric) positive definite.

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

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

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. 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 which eigenvalues to compute. The default is to compute only one eigenvalue (and eigenvector), in particular, the dominant one (largest in magnitude). 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`, 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 al least equal to `nev`, although in general it is recommended (depending on the method) to work with a larger subspace, for instance $\texttt{ncv} \geq 2*\texttt{nev}$ 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. To compute eigenvalues located in the interior part of the spectrum,

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.

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.
- Subspace Iteration with Rayleigh-Ritz projection and locking.
- Arnoldi method with explicit restart and deflation.

The default solver is Arnoldi. A detailed description of the implemented algorithms is included in appendix B of this manual.

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.

In addition to these methods, SLEPc also provides wrappers to external packages such as ARPACK, BLZPACK, PLANSO, or TRLAN. A complete list of these interfaces can be found in section B.4.

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

Method	EPSType	Options Database Name
LAPACK solver	EPSLAPACK	lapack
Power / Inverse / RQI	EPSPOWER	power
Subspace Iteration	EPSSUBSPACE	subspace
Arnoldi Method	EPSARNOLDI	arnoldi
Wrapper to ARPACK	EPSARPACK	arpack
Wrapper to BLZPACK	EPSBLZPACK	blzpack
Wrapper to PLANSO	EPSPLANSO	planso
Wrapper to TRLAN	EPSTRLAN	trlan

Table 2.3: Eigenvalue solvers available in the EPS module.

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);
```

where method can be one of EPSLAPACK, EPSPOWER, EPSSUBSPACE, EPSARNOLDI, EPSARPACK, EPSBLZPACK, EPSPLANSO, or EPSTRLAN. The EPS method can also be set with the options database command -eps_type followed by the name of the method (see table 2.3).

2.5 Controlling the Solution Process

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_0. EPS initializes this 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 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.

It is possible to specify the tolerance of the solution. An approximate eigenvalue is considered to be converged if the error estimate associated to it is lower than the specified tolerance. Note that the error estimates can be computed differently depending on the solution method. The tolerance can be specified 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 permitted to the solution algorithm, which can also be set via `-eps_max_it <its>`. Note that the default values for these and other parameters can be algorithm dependent. See appendix B for reference.

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 `EPSSetMonitor`.

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 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 in carried out typically by a Gram-Schmidt orthogonalization procedure.

It has been acknowledged that the classical Gram-Schmidt (CGS) algorithm may produce vectors which are far from orthogonal. The method known as modified Gram-Schmidt (MGS) is numerically to be preferred, since the achieved orthogonality is of the order of machine precision times condition number of the matrix whose columns are the vectors to orthogonalize. This may still be insufficient for matrices that are ill conditioned, such as the case of Krylov subspaces. To overcome this difficulty, the MGS process can be applied iteratively (a single reorthogonalization step is sufficient in practice). A simple test has been devised to assess when a second orthogonalization is required, see [Daniel *et al.*, 1976]. On the other hand, the same idea is applicable to the CGS process and it has been shown that the same accuracy can be attained in the same number of iterations, see [Hoffmann, 1989].

Algorithm 2.1 (Classical Gram-Schmidt with Iterative Refinement)

Input: Vector v to orthogonalize against the m columns of Q
Output: Orthogonalized vector q

$h = Q^H v$
$\tilde{q} = v - Qh$
If $||\tilde{q}||_2 < \eta||h||_2$
  $s = Q^H \tilde{q}$
  $\tilde{q} = \tilde{q} - Qs$
  $h = h + s$
end
$q = \tilde{q}/||\tilde{q}||_2$

Algorithm 2.2 (Modified Gram-Schmidt with Iterative Refinement)

Input: Vector v to orthogonalize against the m columns of Q
Output: Orthogonalized vector q

$\tilde{q} = v$
For $i = 1, \ldots, m$
  $h_i = q_i^H \tilde{q}$
  $\tilde{q} = \tilde{q} - q_i h_i$
End
If $||\tilde{q}||_2 < \eta||h||_2$
  For $i = 1, \ldots, m$

$s_i = q_i^H \tilde{q}$
$\tilde{q} = \tilde{q} - q_i s_i$
End
$h = h + s$
end
$q = \tilde{q}/||\tilde{q}||_2$

SLEPc provides implementations of both CGS and MGS with iterative refinement (see algorithms 2.1 and 2.2 above). The default is CGS since it is better suited for parallel architectures. The user is able to select the orthogonalization technique to be used. Again, this can be done procedurally or via the command line. The following function provides all the possibilities

```
EPSSetOrthogonalization(EPS eps,EPSOrthogonalizationType type,
   EPSOrthogonalizationRefinementType refinement, PetscReal eta);
```

The argument `type` can be used to choose between CGS and MGS. The argument `refinement` specifies if refinement should be performed always (thus carrying out unnecessary work), never (i.e. the non-iterative algorithms) or if needed (according to the condition established in the algorithms above). In the last case, the value of η can be provided via the last argument, `eta`. The default is to do refinement if needed with a value of η equal to $1/\sqrt{2}$, as suggested in [Reichel and Gragg, 1990]. Alternatively, all these options can be specified in the command line with `-eps_orthog_type [cgs|mgs]` for the algorithm, `-eps_orthog_refinement [never|ifneeded|always]` for the refinement strategy, and `-eps_orthog_eta` for setting the value of η.

Note that in some situations, the orthogonalization subroutines of SLEPc will carry out a B-orthogonalization instead of an orthogonalization. In other words, the resulting vectors satisfy $Q^H B Q = I$ instead of $Q^H Q = I$ (to working accuracy). This is related to what is presented in section 3.4.4.

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 nullspace 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 nullspace. In any case, SLEPc internally checks to see if all (or part of) the provided subspace is a nullspace of the associated linear system (see section 3.4.1). In this case, this nullspace is passed to the linear solver (see PETSc's function `KSPSetNullSpace`) to enable the solution of

singular systems. In practice, this allows to compute eigenvalues of singular pencils (i.e. when A and B share a common nullspace).

Chapter 3

ST: Spectral Transformation

The other main SLEPc object is the Spectral Transformation (`ST`), which encapsulates the functionality required for acceleration techniques based on the transformation of the spectrum. As explained in chapter 2, the implemented eigensolvers 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 manipulating 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:

- Avoid convergence problems. For instance, simple methods such as the Power Iteration can fail to obtain the solution under certain conditions, and sometimes this situation can be avoided by simply shifting the spectrum.

- 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.

3.2 Basic Usage

The `ST` module is the analogue to other 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: 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 — see section 4.4 for a detailed discussion of this issue.]

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, `STApplyB`, which applies matrix B to a vector, and `STSetUp` which prepares all the necessary data structures before the solution process starts. The 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 which 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);
```

where `type` can be one of `STSHIFT`, `STSINV`, `STCAYLEY` or `STSHELL`. The `ST` type can also be set with the options database command `-st_type` followed by the name of the method (see table 3.1).

The first three spectral transformations are described in detail in the rest of this section. The last possibility, `STSHELL`, uses a specific, application-provided spectral transformation. Section 3.4.3 describes how to implement one of this transformations.

Spectral Transformation	`STType`	Options Name	Operator
Shift of Origin	`STSHIFT`	`shift`	$B^{-1}A + \sigma I$
Shift-and-invert	`STSINV`	`sinvert`	$(A - \sigma B)^{-1}B$
Cayley	`STCAYLEY`	`cayley`	$(A - \sigma B)^{-1}(A + \tau B)$
Shell Transformation	`STSHELL`	`shell`	–

Table 3.1: Spectral transformations available in the `ST` package.

`ST`	Choice of σ, τ	Standard problem	Generalized problem
`shift`	$\sigma = 0$	A	$B^{-1}A$
	$\sigma \neq 0$	$A + \sigma I$	$B^{-1}A + \sigma I$
`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.

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 antishift (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 which 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.

In the table, the value σ represents the shift, whereas τ is called the antishift (used only in the Cayley transformation). As explained above, the shift can be specified via the `STSetShift` function or in the command line. The antishift can be given in a similar way (see 3.3.4 below).

3.3.1 Default Behavior

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,

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

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

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

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.

3.3.2 Shift of Origin

The purpose of this spectral transformation (`STSHIFT`) is to shift the whole spectrum by a certain quantity, σ, which is called *shift of origin*. To achieve this, the solver has to work with the shifted matrix, that is, the expressions it has to cope with are

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

for standard problems, and

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

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

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

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.

3.3.3 Shift-and-invert

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

$$(A - \sigma I)^{-1} x = \theta x \quad , \tag{3.6}$$

for standard problems, and

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

for generalized ones. This transformation is effective for finding eigenvalues near σ since the eigenvalues θ of the operator that are largest in magnitude correspond to the eigenvalues λ of the original problem that are closest to the shift σ in absolute value. Once they are 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

$$\theta = 1/(\lambda - \sigma) \quad . \tag{3.8}$$

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

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

for standard problems, and

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

for generalized ones. 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 advantatgeous with respect to shift-and-invert (see [Bai *et al.*, 2000, §11.2]).

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

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

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 to the most beneficial way. This section describes several advanced aspects which 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 system 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_ilu_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 which can be explicitly constructed or not, depending on the user's choice. This issue is examined in detail next.

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 advantatges 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 Shell Transformations

The `ST` package allows the user to define new spectral transformations by means of the `shell` type, in a similar way as *shell* preconditioners or *shell* matrices.

This tool is intended for simple spectral transformations. For more sophisticated transformations, the user should register a new `ST` type (see section 4.2 for details).

At least, user-defined spectral transformations have to define how the operator is to be applied to a vector. Optionally, it 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 which 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`).

3.4.4 Preserving the Symmetry

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 are typically more accurate. However, in the case of generalized eigenvalue problems in which both A and B are symmetric, we have that due to the spectral transformation, symmetry is lost because none of the operators $B^{-1}A+\sigma I$, $(A-\sigma B)^{-1}B$ or $(A-\sigma B)^{-1}(A+\tau B)$ 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:

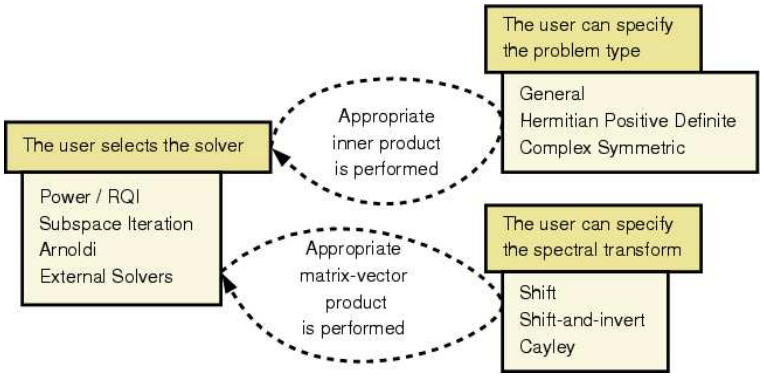

Figure 3.1: Abstraction used by SLEPc solvers.

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

The second one will be used for preserving the symmetry in symmetric definite generalized problems. Note that $\langle x, y\rangle_B$ is a genuine inner product only if B is symmetric positive definite.

$\mathbb{R}^n$ with $\langle x, y\rangle_B$ 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 $B^{-1}A$ is auto-adjoint with respect to $\langle x, y\rangle_B$, and similarly with the rest of spectral transformations.

Internally, SLEPc operates with the abstraction illustrated in figure 3.1. The operations indicated by dashed arrows are implemented as virtual functions: `STInnerProduct` 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).

Chapter 4

Relation with PETSc

SLEPc relies on PETSc for all the features which 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.
- 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.
  - Profile application codes with user-defined events.

  - Direct wall-clock timing with `PetscGetTime`.
  - Display detailed profile information and trace of events.
  - Graphical visualization of events with MPE.

- 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 graphics viewers.
- Interface to external software such as MATLAB®.
- Easy handling of runtime options.

This chapter discusses several issues related to the interaction between SLEPc and PETSc which can be important for the user.

4.1 Supported Matrix Objects

Methods implemented in the `EPS` module merely require vector operations and matrix-vector products. In PETSc, mathematical objects such as vectors and matrices have an interface which 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,

$$\begin{bmatrix} A & C \\ C^H & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} = \lambda \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} \quad , \tag{4.1}$$

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 equation (A.9), or the singular value decomposition (A.12).

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 which 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 which 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).

4.2 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. This is done by using the `STSHELL` spectral transformation in which 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 3.4.3 for details).

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 the Subspace Iteration method with symmetric projection, 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`. In a similar way, `STRegister` and `STRegisterDynamic` can be used to register new spectral transformation types.

4.3 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 1.5 and can be found in `${SLEPC_DIR}/src/examples/ex1f.F`.

```
!
!  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
!
!/*T
!  Concepts: SLEPc - Basic functionality
!  Routines: SlepcInitialize(); SlepcFinalize();
!  Routines: EPSCreate(); EPSSetFromOptions();
!  Routines: EPSSolve(); EPSDestroy();
!T*/
!
! -----------------------------------------------------------------------
!
      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
!  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      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)

      if (rank .eq. 0) then
        write(*,100) n
      endif
  100 format ('1-D Laplacian Eigenproblem, n =',i6)

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

      call MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,A,
     &               ierr)
      call MatSetFromOptions(A,ierr)

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

115 ! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    !     Solve the eigensystem
    ! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call EPSSolve(eps,ierr)
120   call EPSGetIterationNumber(eps,its,ierr)
      if (rank .eq. 0) then
        write(*,*)
        write(*,140) its
      endif
125  140  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
130     write(*,110) type
      endif
     110  format (' Solution method: ',A)
      call EPSGetDimensions(eps,nev,PETSC_NULL_INTEGER,ierr)
      if (rank .eq. 0) then
135     write(*,120) nev
      endif
     120  format (' Number of requested eigenvalues:',I2)
      call EPSGetTolerances(eps,tol,maxit,ierr)
      if (rank .eq. 0) then
140     write(*,130) tol, maxit
      endif
     130  format (' Stopping condition: tol=',1PE10.4,', maxit=',I6)

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

    !     ** Get number of converged eigenpairs
      call EPSGetConverged(eps,nconv,ierr)
150   if (rank .eq. 0) then
        write(*,150) nconv
```

```
      endif
 150  format (' Number of converged approximate eigenpairs:',I2)

!     ** Display eigenvalues and relative errors
      if (nconv.gt.0 .and. rank.eq.0) then
        write(*,*)
        write(*,*) '         k          ||Ax-kx||/||kx||'
        write(*,*) ' ----------------- ------------------'
        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 (ki.ne.0.D0) then
            write(*,180) kr, ki, error
          else
            write(*,190) kr, error
          endif
        enddo
        write(*,*)
      endif
 180  format (1P,E11.4,E11.4,' j ',E12.4)
 190  format (1P,'   ',E12.4,'       ',E12.4)

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

      call SlepcFinalize(ierr)
      end
```

4.4 Complex Numbers

PETSc supports the use of complex numbers in application programs written in C, C++ and Fortran. Currently, this is done 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. [Note: this may change in future versions of PETSc.]

SLEPc inherits this property. To build the real version of the SLEPc libraries, the flag `BOPT` must be set to `g` or `O` (debug or optimized flavors, respectively). To build the complex version, one of `BOPT=[g_complex,O_complex]` must be

used. Application programs must be compiled also specifying the appropriate `BOPT` value to link with the desired libraries.

In SLEPc it is not possible to completely separate real numbers and complex numbers because the solution of non-symmetric real-valued eigenvalue problems can be complex. SLEPc has been designed trying to provide a uniform interface to manage all the possible cases. This section clarifies the differences between the interface in each of the two versions, mainly in the format of the computed solution and the shifts.

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. The sign of the imaginary part is returned correctly in any case by function `EPSGetEigenpair`.

Complex SLEPc. In this case, all `Mat` and `Vec` objects are complex. The computed approximate solution returned by the 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).

Shifts. Some packages such as ARPACK support the use of complex shifts even when working with real arithmetic. Currently, this is not supported in SLEPc. The shifts in the `ST` package are defined as `PetscScalar` variables and, therefore,

the complex version of the libraries must be used in order to be able to specify complex shifts.

4.5 Makefiles

SLEPc uses a makefile system very similar to that of PETSc. All platform specific setting are taken directly from the PETSc installation. During installation of the SLEPc libraries, only the file `${SLEPC_DIR}/bmake/${PETSC_ARCH}/packages` must be edited to indicate the presence of optional software packages such as ARPACK.

With respect to the application program makefiles, they are very easy to set up 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:

```
include ${SLEPC_DIR}/bmake/slepc_common

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

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

Appendix A

Background Material

A.1 The Eigenvalue Problem

The eigenvalue problem is a central topic in numerical linear algebra. In the standard formulation, the problem consists in the determination of $\lambda \in \mathbb{C}$ for which the equation

$$Ax = \lambda x \tag{A.1}$$

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 eigenvector, respectively.

In many applications, the problem is formulated as $Ax = \lambda Bx$, which is known as the generalized eigenvalue problem. Usually, this problem is solved by reformulating it in standard form, as discussed in section A.1.3.

Similarity transformations preserve eigenvalues. Two matrices A and $\tilde{A}$ are similar if a non-singular matrix X exists such that $A = X\tilde{A}X^{-1}$. Many methods for eigenvalue problems rely on such transformations in order to reduce the matrix to a canonical form from which it is easier to retrieve eigenpairs. Among these methods are the ones considered to be the fastest and most accurate methods, such as Divide and Conquer, QR Iteration and Jacobi methods. For

an up-to-date survey on methods for eigenvalue problems see [Golub and van der Vorst, 2000].

However, these methods are not appropriate for large sparse matrices because similarity transformations destroy sparsity. Moreover, most applications require only to know a few selected eigenvalues and not the entire spectrum. For these reasons, other methods have become popular for sparse problems.

A.1.1 Basic Methods

Methods for sparse eigenproblems obtain the solution from the information generated by the application of the operator to various vectors. That is, the matrix is only used in matrix-vector products. This not only maintains sparsity but allows to solve problems in which matrices are not available explicitly. A catalog of such methods can also be found in [Golub and van der Vorst, 2000]. For a more comprehensive description see [Bai *et al.*, 2000].

The most basic method of this kind is the Power Iteration, in which an initial vector is repeatedly premultiplied by the matrix A and conveniently normalized. After a certain number of iterations, this vector converges to the dominant eigenvector, which is the one associated to the eigenvalue with largest module. In many situations, the particular properties of the spectrum can prevent the Power Method from converging. Also, it is usual to require more than just one eigenvalue. For these reasons, more powerful methods are required.

The Simultaneous Iteration or Subspace Iteration is the generalization of the Power Method. In this method, the matrix is applied to a set of m vectors simultaneously, and orthogonality is enforced explicitly in order to avoid the convergence of all these vectors to the dominant eigenvector.

In some sense, the power method throws away potentially useful spectral information during the course of the iteration. At the k-th iteration, the algorithm overwrites the vector $A^{k-1}x^{(0)}$ with $A^k x^{(0)}$, where $x^{(0)}$ is the initial vector. However, it turns out to be useful to keep the previous vector instead of overwriting it, and by extension to keep the whole set of previous vectors. The subspace

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

is called the m-th Krylov subspace corresponding to A and v. Methods which use linear combinations of vectors in this space to extract spectral information

are called Krylov subspace methods. The most basic methods of this kind are the Lanczos, non-symmetric Lanczos and Arnoldi algorithms.

The basic idea of these methods is to construct approximate eigenvectors in the Krylov subspace $\mathcal{K}_m(A, v)$. A Ritz pair is any pair (λ_i, x_i) that satisfies the Galerkin condition,

$$(Ax_i - \lambda_i x_i,\, v) = 0 \;\;, \quad \forall v \in \mathcal{K}_m(A, v) \;\; . \tag{A.3}$$

That is, the Ritz pair satisfies the eigenvalue-eigenvector relationship in the projection onto a smaller space. If the component orthogonal to this space is sufficiently small then the Ritz pair is a good approximation to an eigenpair of A. The procedure for constructing approximate eigenpairs in this way is called Rayleigh-Ritz projection.

The following is the Lanczos method:

Select an initial vector v_1 of norm 1
Initialize $\beta_1 = 0$, $v_0 = 0$
For $j = 1, 2, \ldots, k$
    $w_j = Av_j - \beta_j v_{j-1}$
    $\alpha_j = v_j^H w_j$
    $w_j = w_j - \alpha_j v_j$
    $\beta_{j+1} = \|w_j\|_2$
    $v_{j+1} = w_j / \beta_{j+1}$
end

This algorithm builds an orthonormal basis $V_k = [v_1, \ldots, v_k]$ and computes a tridiagonal matrix T_k, where α_j and β_j form the diagonal and sub-diagonal elements, respectively, so that $T_k = V_k^T A V_k$. Let (λ, y) be an eigenpair of T_k, i.e., $T_k y = \lambda y$, then λ is a Ritz value of A and the corresponding Ritz vector is $x = V_k y$. In practice, the Lanczos vectors v_j may lose orthogonality when the above algorithm is carried out in floating-point arithmetic. Some strategies can be used to avoid this problem, including partial or selective re-orthogonalization and elimination of spurious eigenvalues.

In order to be able to solve non-symmetric eigenproblems, the non-symmetric Lanczos method uses two bi-orthonormal basis to construct a non-symmetric tridiagonal matrix.

The Arnoldi method, which is also intended for the non-symmetric case, builds a k-step Arnoldi factorization,

$$AV_k = V_k H_k + f_k e_k^T \ , \tag{A.4}$$

where the columns of V_k are orthonormal, $V_k^H f_k = 0$, and H_k is an upper Hessenberg matrix of order k. As in the Lanczos method, eigenpairs of H_k can be used for building Ritz pairs. The Arnoldi algorithm can be written as follows.

Select an initial vector v_1 of norm 1
For $j = 1, 2, \ldots, k$
  $h_{ij} = v_j^H A v_i, \ i = 1, 2, \ldots, j$
  $w_j = A v_j - \sum_{i=1}^{j} h_{ij} v_i$
  $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ Stop
  $v_{j+1} = w_j / h_{j+1,j}$
end
$f_k = h_{k+1,k} v_{k+1}$

The above algorithm uses the classical Gram-Schmidt orthogonalization scheme when constructing the basis. Other schemes are usually preferred in order to avoid problems with round-off errors.

A.1.2 Convergence

The Power Method is used to compute a single eigenvector. A simple modification can be done to find the k dominant eigenvectors: once the eigenpair (λ_1, x_1) is computed, a transformation is applied to the matrix A to move λ_1 to the interior of the spectrum, so that the second largest eigenvalue λ_2 becomes the dominant eigenvalue of the transformed matrix. This process is repeated until the k dominant eigenvalues have been found. This technique is called *deflation* and it appears implicitly in many other algorithms.

In methods such as Subspace Iteration or Lanczos, the convergence rate is different from one eigenpair to another. Sometimes the cost of the algorithm can be reduced by *locking* eigenvectors once they have already converged to desired accuracy. This technique is another form of deflation.

Convergence problems can arise in the presence of multiple or clustered eigenvalues. Selecting a sufficiently large number of basis vectors can usually avoid

the problem. However, convergence can still be very slow and acceleration techniques must be used. 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 eigenvalues of interest are well separated in the transformed spectrum thus leading to fast convergence. Sometimes, the transformation can also be constructed to explicitly damp unwanted eigenvalues.

The simplest transformation is to use the shifted matrix $A + \sigma I$. Other transforms are shift-and-invert $(A - \sigma I)^{-1}$, Cayley $(A - \sigma I)^{-1}(A + \sigma I)$ and, in general, polynomial $p(A)$ or even rational $p(A)q(A)^{-1}$ transformations. The most commonly used one is the shift-and-invert transformation, which 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.

A.1.3 Non-standard Problems

Although there are specific methods for the generalized eigenvalue problem, $Ax = \lambda Bx$, it is usually solved by reducing it to standard form. There are several possibilities for doing this. If B is non-singular, the problem can be written as

$$B^{-1}Ax = \lambda x \quad . \tag{A.5}$$

If B is singular or ill-conditioned, the roles of A and B can be reversed. On the other hand, if B is symmetric positive definite then the problem is equivalent to

$$L^{-1}AL^{-T}y = \lambda y \quad , \tag{A.6}$$

where $y = L^T x$ and L is lower triangular such that $B = LL^T$.

In any case, a system of linear equations must be solved in each iteration of the eigensolver. For this reason, using the shift-and-invert technique does not add extra complexity. In this case, after solving the transformed problem

$$(A - \sigma B)^{-1}Bx = \theta x \quad , \tag{A.7}$$

the eigenvalues of the original problem can be recovered as $\lambda = \frac{1}{\theta} + \sigma$ while the eigenvectors stay the same. Note that this transformation is valid regardless of the regularity of B.

When using equations (A.5) or (A.7) symmetry is lost. In order to be able to use methods such as Lanczos which assume a symmetric operator, Euclidean products and norms must be replaced by B-inner products and B-norms.

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

$$(A\lambda^2 + B\lambda + C)x = 0 \ . \tag{A.8}$$

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

$$\begin{bmatrix} -B & -C \\ I & 0 \end{bmatrix} v = \lambda \begin{bmatrix} A & 0 \\ 0 & I \end{bmatrix} v \ . \tag{A.9}$$

Other linear algebra problems are very closely related to eigenproblems. One of them is the *singular value decomposition* (SVD). Let A be a real $m \times n$ matrix, then there exist two orthogonal matrices $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ such that

$$U^T A\, V = \mathrm{diag}(\sigma_1, \ldots, \sigma_p) \ , \tag{A.10}$$

with $p = \min\{m, n\}$ and $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$. The values σ_i are called singular values. The relation (A.10) can be expressed as an eigenproblem in several ways, for example

$$A^T A\, v_i = \sigma_i^2 v_i \ , \tag{A.11}$$

or

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \sigma_i \begin{bmatrix} u_i \\ v_i \end{bmatrix} \ . \tag{A.12}$$

A.1.4 State-of-the-art Methods

One drawback of the Arnoldi algorithm with respect to Lanczos is the increment of cost and storage as the size of the subspace increases. The solution to this is to restart the Arnoldi reduction when a certain size is reached. The Implicit

Restart technique allows to filter away unwanted information from the process. This leads to a reduced subspace with a basis for which the matrix still has a Hessenberg form, so that Arnoldi's process can be continued with a subspace (rather than with a single vector as with more classical restart techniques). This Implicitly Restarted Arnoldi method is the one implemented in ARPACK. Another strategy called thick-restarting has been proposed to achieve similar effectiveness while somewhat reducing the complexity.

Krylov subspace methods are projection methods because they project the original matrix onto a certain subspace. Other projection methods use the same idea but with a different type of subspace. One of these techniques, called Rational Krylov Sequence (RKS), combines a spectral transformation such as shift-and-invert with a modification of the shift in each step. The result is the generation of the so-called rational Krylov subspace

$$\text{span}\left\{v, T_1^{SI}v, \ldots, \left(T_{m-1}^{SI}\cdots T_1^{SI}\right)v\right\} \quad , \tag{A.13}$$

where $T_j^{SI} = (A - \sigma_j B)^{-1}B$. Since the matrix $A - \sigma_j B$ changes at every step, it is not feasible to compute a factorization at the beginning and then reuse the factors to solve the system at each iteration. Iterative linear solvers may be preferred in this case.

Another approach is Davidson's method which expands the subspace by orthogonalizing $\tilde{v}$ with respect to the previous basis vectors. This vector $\tilde{v}$ is the solution of the linear system of equations $(D_A - \theta I)\tilde{v} = r$, where D_A is the diagonal of matrix A, r is the defect $r = Au - \theta u$, and (θ, u) is a Ritz pair. This method was enhanced later by introducing a correction Δu for u in the subspace orthogonal to u. In this case, the linear system to be solved is

$$(I - uu^H)(A - \theta I)(I - uu^H)\Delta u = -r \quad , \tag{A.14}$$

and the whole process constitutes the Jacobi-Davidson method.

Finally, the block-oriented analogues of some of the methods described above can be very effective in some situations. An example of this is the ABLE method, a block Lanczos algorithm which adaptively adjusts the block size.

A.1.5 Available Software

The development of high-quality software for eigenvalue problems starts with the book edited by Wilkinson and Reinsch [1971] which contained implementations

in Algol60 of many algorithms for the solution of linear systems of equations and eigenvalue problems. In the early 1970's, these algorithms gave way to the packages LINPACK and EISPACK, which were written in Fortran. EISPACK [Smith *et al.*, 1970], which concentrates on eigenvalue problems, is the first library approaching this kind of problems in a rigorous way. These libraries are the basement of more recent software such as the commercial packages IMSL and NAG, or even MATLAB®. In the 1990's, LINPACK and EISPACK were replaced by LAPACK [Anderson *et al.*, 1992], a completely rewritten library which already guarantees reliability, portability and efficiency in a systematic way. It implements block-oriented versions of the algorithms and makes extensive use of the level 3 Basic Linear Algebra Subprograms (BLAS). Most of the algorithms for eigenvalue problems included in LAPACK have their parallel counterpart in SCALAPACK [Blackford *et al.*, 1997], today the main source of parallel algorithms for dense linear algebra problems.

Unlike in dense methods, there are few standards 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 parallel case, where additional indexing information is needed to specify which matrix elements are on which processor.

The first implementations of algorithms for sparse matrices appeared in the journal *Transactions on Mathematical Software*, for example LOPSI [Stewart and Jennings, 1981] which implements the Subspace Iteration method. This subroutine 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. This is the option used in SRRIT [Bai and Stewart, 1997], which also implements Subspace Iteration, and ARNCHEB [Braconnier, 1993], which implements the Arnoldi method with explicit restart. A description and comparison of some of these packages can be found in [Lehoucq and Scott, 1996].

A good solution for the matrix representation problem is the well-known reverse communication interface, a technique which 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. This scheme is very useful when, for example, the method requires not only the product by the matrix but also the product by the transpose. This is the case of QMRPACK [Freund and Nachtigal, 1996], which implements the non-symmetric Lanczos method.

More recent software packages implement advanced techniques and are prepared for parallel execution. This is the case of ARPACK, BLZPACK, PLANSO, and TRLAN. All of them have been integrated in SLEPc by means of a wrapper. A description of these packages can be found in section B.4.

A.2 Review of PETSc

The solution to the problem of using distributed-memory computers efficiently is the combination of the message-passing programming model and carefully designed and implemented parallel numerical libraries. This approach is also appropriate for other architectures such as clusters and NUMA (non-uniform memory access) shared-memory computers. This combination allows to strike a balance between code performance and ease of use.

Since the advent of the Message Passing Interface (MPI), the message-passing model has been widely used and several parallel numerical libraries have been developed. One of them is PETSc [Balay *et al.*, 2004], whose 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 freely available 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 focuses on components required for the solution of partial differential

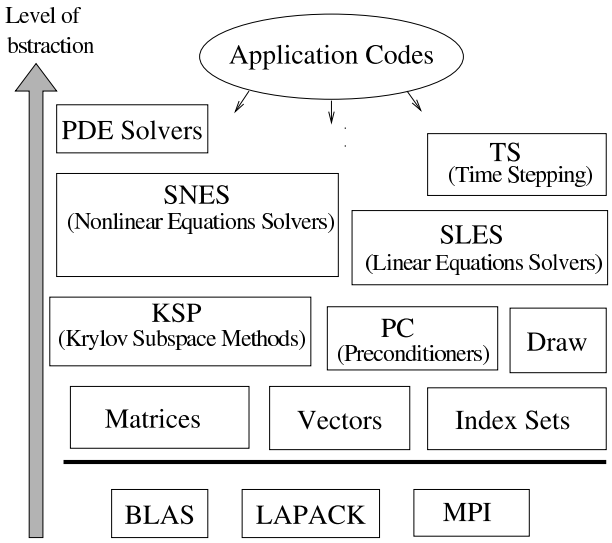

Figure A.1: Organization of the PETSc toolkit.

equations (PDEs) and related problems. In this kind of problems, a geometric decomposition of the solution domain among the processors is the most appropriate approach, leading to data locality and therefore to scalability (at least theoretically). PETSc is designed to provide efficient tools for handling problems arising from the discretization of PDEs by means of regular meshes (e.g. finite differences) as well as unstructured meshes (e.g. finite elements).

PETSc is built around a variety of data structures and algorithmic objects. Figure A.1 shows a diagram of some of these components, and illustrates the library's hierarchical organization. The application programmer works directly with these objects rather than concentrating on the underlying data structures. He has to consider the interrelationships among different pieces of PETSc, and employ the level of abstraction that is most appropriate for a particular problem.

PETSc components are discussed in detail in the users manual [Balay *et al.*, 2004]. 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. Some of the PETSc modules deal with

- index sets, including permutations, for indexing into vectors, renumbering, etc;
- vectors;
- matrices (generally sparse);
- distributed arrays (useful for parallelizing regular grid-based problems);
- Krylov subspace methods;
- preconditioners, including multigrid and sparse direct solvers;
- nonlinear solvers; and
- timesteppers for solving time-dependent (nonlinear) PDEs.

Each of these components consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. PETSc is 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 access 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 which handles the particular data structure. This is implemented by structures of function pointers. 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 provides clean and effective codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables

easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping.

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.

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. Other advantages of PETSc are the following:

- High portability due to an elaborate Makefile system. Different architecture builds can coexist in the same installation. Where available, dynamic libraries are used to reduce disk space of executable files.
- Support for debugging and profiling: attachment to external debuggers, event logging, subroutine timing, convergence monitoring, etc. PETSc also has built-in graphics capabilities which allow for sparse pattern visualization, graphic convergence monitoring, operator's spectrum visualization and other user-defined operations.
- Programming interface for Fortran.

Appendix B

Catalog of Solvers

This appendix provides a short description of all the eigensolvers which are available in SLEPc, including the interfaces to external libraries. In the case of "native" methods, that is, those methods that are implemented directly in SLEPc, the description includes a sketch of the algorithm which is actually implemented. A short description of wrappers to external libraries is provided in section B.4 of this appendix, including pointers to the respective websites from which the software can be downloaded. In both cases, 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.

Table B.1 summarizes all eigensolvers available in SLEPc, both native and external. This table shows the default values for some of the parameters that the user can adjust.

Table B.2 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.

Method	ncv	max_it	tol
lapack	nev	-	-
power	nev	$\max(2000, 100N)$	10^{-7}
subspace	$\max(2 \cdot nev, nev + 15)$	$\max(100, N)$	10^{-7}
arnoldi	$\max(2 \cdot nev, nev + 15)$	$\max(100, N)$	10^{-7}
arpack	$\min(\max(20, 2 \cdot nev+1), N)$	$\max(300, \lceil 2N/ncv \rceil)$	10^{-7}
blzpack	$\min(nev+10, 2 \cdot nev)$	$\max(100, N)$	10^{-7}
planso	nev	$\max(100, N)$	10^{-7}
trlan	nev	$\max(100, N)$	10^{-7}

Table B.1: Default parameter values for all eigensolvers available in SLEPc.

Method	Portion of spectrum	Problem type	Complex
lapack	all	all	yes
power	Largest $\|\lambda\|$	all	yes
subspace	Largest $\|\lambda\|$	all	yes
arnoldi	Largest $\|\lambda\|$	all	yes
arpack	all	all	yes
blzpack	Smallest $\mathrm{Re}(\lambda)$	EPS_HEP, EPS_GHEP	no
planso	Largest and smallest $\mathrm{Re}(\lambda)$	EPS_HEP	no
trlan	Largest and smallest $\mathrm{Re}(\lambda)$	EPS_HEP	no

Table B.2: Supported problem types for all eigesolvers available in SLEPc.

B.1 power

This eigensolver covers several well-known single-vector iteration methods such as the Power Iteration, the Inverse Iteration and the Rayleigh Quotient Iteration.

When using the default spectral transformation (`STSHIFT`), this solver provides an implementation of the Power Iteration. This is the simplest vector iteration method. It consists in premultiplying an initial vector by matrix A repeatedly. Under certain conditions, the iteration converges to the dominant eigenvector (the one associated to the largest eigenvalue in magnitude). Then the approximate eigenvalue can be obtained by computing the Rayleigh quotient. Once an eigenvector has converged, deflation can be used to reveal the next ones. Note that this method will fail in the case that there is no unique dominant eigenvalue. Convergence can be very slow if separation of the dominant eigenvalue with the rest is small.

Algorithm B.1 (Basic Power Method)

Input: Operator OP and initial vector v_0
Output: Approximate dominant eigenpairs (θ, v)

Set $y = v_0$
For $k = 1, 2, \ldots$
  Deflate previously converged eigenvectors
  $v = y/\|y\|_B$
  $y = OPv$
  $\theta = (y, v)_B$
  if $\|y - \theta v\|_2 < |\theta| \cdot \text{tol}$, accept eigenpair
end

In the algorithm above, matrix OP represents the operator, which can be any of the expressions in table 3.2. Thus, the algorithm can be converted to the Inverse Iteration by simply specifying the `sinvert` transformation.

The Inverse Iteration works with a constant shift, that is, the operator is $(A - \sigma B)^{-1}B$ throughout the iteration. However, it is possible to change this behaviour by requesting the use of *variable* shifts. There are two possibilities: Rayleigh shifts and Wilkinson shifts. This option can be changed with the command-line option `-eps_power_shift_type` or specified in the program source with the function `EPSPowerSetShiftType`.

The first alternative (`rayleigh`) results in the well-known Rayleigh Quotient Iteration (RQI), where the new shift ρ_k is computed as the Rayleigh quotient of the current approximate eigenvector v

$$\rho_k = R(v) = \frac{v^H A v}{v^H B v} \tag{B.1}$$

The rate of convergence of this strategy is quadratic in general and cubic when the problem is Hermitian. Note that changing the shift may imply refactorization of matrix $(A - \rho_k B)$.

The second alternative (`wilkinson`) uses a more sophisticated formula for the shift [Parlett, 1998].

B.2 subspace

The Subspace Iteration method is a generalization of the Power Method to m initial vectors. Ortogonality of vectors is enforced in order to avoid linear dependence.

The implementation currently available in SLEPc is based on SRRIT [Bai and Stewart, 1997]. It performs a Rayleigh-Ritz projection procedure in order to improve convergence. Deflation is handled by locking converged eigenvectors. For better performance, orthogonalization and projection are performed only when necessary.

Algorithm B.2 (Subspace Iteration)

Input: Operator *OP*
Output: m dominant Schur vectors V and corresponding eigenvalues
    Generate a set of initial orthonormal vectors $V \in \mathbb{C}^{n \times m}$
    For $k = 1, 2, \ldots$
        Perform a Rayleigh-Ritz Projection step (algorithm B.3)
        Check convergence of eigenvalues
        Orthogonalization loop
            Repeatedly compute $V = OP\,V$
            Orthogonalize columns of V
        end
    end

Algorithm B.3 (Rayleigh-Ritz Projection)

Input: Operator OP and set of vectors V
Output: Schur vectors V and quasi-triangular matrix T
$T = V^H OP\, V$
Reduce to Hessenberg form: $T = U_1^H T U_1$
Reduce to quasi-triangular form: $T = U_2^H T U_2$
$U = U_1 U_2$
$V = VU$

B.3 `arnoldi`

The version of the Arnoldi method implemented in SLEPc uses locking and explicit restart. The orthogonalization technique can be chosen as described in section 2.6.1.

The implemented method (see algorithm B.4) builds an Arnoldi factorization of order `ncv` by means of algorithm B.5. Converged eigenpairs are locked and the iteration is restarted with the rest of the columns being the active columns for the next Arnoldi factorization. Currently, no filtering is applied to the vector used for restarting.

Algorithm B.4 (Explicitly Restarted Arnoldi)

Input: Operator OP, initial vector v_1, and dimension of the subspace ncv
Output: A partial Schur decomposition $OP\, V = V H$
Normalize v_1
Restart loop
Compute an ncv-step Arnoldi factorization (algorithm B.5)
Reduce H to (quasi-)triangular form, $H \leftarrow U H U^T$
Compute eigenvectors of H, $H y_i = y_i \theta_i$
Compute residual norm estimates, $\tau_i = \beta |e_m^T y_i|$
Lock converged eigenpairs
$V = VU$
end

Algorithm B.5 (Basic Arnoldi Factorization)

Input: Operator OP, number of steps m, and V_k, H_k with $k < m$
Output: (V_m, H_m, f, β) so that $OP\,V_m - V_m H_m = f e_m^T$, $\beta = \|f\|_B$

For $j = k+1, \ldots, m-1$
  $w = OPv_j$
  Orthogonalize w with respect to V_j (obtaining $h_{1:j,j}$)
  $h_{j+1,j} = \|w\|_B$
  $v_{j+1} = w/h_{j+1,j}$
end
$f = OPv_m$
Orthogonalize f with respect to V_m (obtaining $h_{1:m,m}$)
$\beta = \|f\|_B$

B.4 Wrappers to External Libraries

SLEPc interfaces to several external libraries for the solution of eigenvalue problems. This section includes a short description of each of these packages as well as some hints for using them with SLEPc.

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

1. Install the external software.
2. Enable the utilization of the external software from SLEPc by editing the file `${SLEPC_DIR}/bmake/${PETSC_ARCH}/packages`. For example, to use ARPACK, one would specify the following variables with the appropriate paths:

```
ARPACK_INCLUDE    =
ARPACK_LIB        = -L/home/slepc/soft/ARPACK -lparpack -larpack
SLEPC_HAVE_ARPACK = -DSLEPC_HAVE_ARPACK
```

3. Build the SLEPc libraries.
4. Use the runtime option `-eps_type <type>` to select the solver.

An exception to the above is LAPACK, which should be enabled during the PETSc installation.

LAPACK

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

Website. `http://www.netlib.org/LAPACK`.

Version. 2.0 or later.

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

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.

Currently, only LAPACK drivers for standard eigenvalue problems are used. Generalized problems are transformed to standard ones.

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. In order to use ARPACK with SLEPc, both the sequential version and the parallel version (PARPACK) have to be installed. First, unbundle `arpack96.tar.gz`, then `parpack96.tar.gz`. Make sure you delete any `mpif.h` files that could exist in the directory tree. Also it is recommended to unpack the patch files `patch.tar.gz` and `ppatch.tar.gz`. After that, modify the `ARmake.inc` file and then compile the software with `make all`.

BLZPACK

References. [Marques, 1995].

Website. `http://www.nersc.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.

PLANSO

References. [Wu and Simon, 1997].

Website. `http://www.nersc.gov/research/SIMON/planso.html`.

Version. 1.0 (07/1997).

Summary. This package implements the Lanczos algorithm with partial reorthogonalization for symmetric generalized eigenvalue problems. It is based on the sequential package LANSO maintained by B. Parlett. PLANSO is implemented in Fortran 77 using MPI and the user must provide functions for matrix-vector products.

The current version uses the Omega-recurrence to simulate the loss of orthogonality among the Lanczos vectors and maintains semiorthogonality. This is sufficient to guarantee that eigenvalues are computed accurately, but under extreme conditions the eigenvectors may not be as accurate as the eigenvalues.

Installation. Change `Make.inc` in the top level directory to set appropriate compiler and flags to use. Then type `make lib plib`.

TRLAN

References. [Wu and Simon, 2001].

Website. `http://www.nersc.gov/~kewu/trlan.html`.

Version. 1.0 (03/1999).

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 which 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.

Bibliography

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1992). *LAPACK User's Guide*. SIAM Publications, 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*. SIAM, Philadelphia.

Bai, Z. and G. W. Stewart (1997). Algorithm 776: SRRIT: A FORTRAN Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix. *ACM Transactions on Mathematical Software*, 23(4):494–513.

Balay, S., K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang (2004). PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.2.1, 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. Birkhauser Press.

Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley (1997). *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Braconnier, T. (1993). The Arnoldi-Tchebycheff Algorithm for Solving Large Nonsymmetric Eigenproblems. Technical Report TR/PA/93/25, CERFACS, Toulouse, France.

Daniel, J. W., W. B. Gragg, L. Kaufman, and G. W. Stewart (1976). Reorthogonalization and Stable Algorithms for Updating the Gram–Schmidt QR Factorization. *Mathematics of Computation*, 30(136):772–795.

Freund, R. W. and N. M. Nachtigal (1996). QMRPACK: a Package of QMR Algorithms. *ACM Transactions on Mathematical Software*, 22:46–77.

Golub, G. H. and H. A. van der Vorst (2000). Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):35–65.

Hoffmann, W. (1989). Iterative algorithms for Gram-Schmidt orthogonalization. *Computing*, 41(4):335–348.

Lehoucq, R. B. and J. A. Scott (1996). An Evaluation of Software for Computing Eigenvalues of Sparse Nonsymmetric Matrices. Preprint MCS-P547-1195, Mathematics and Computer Science Division, Argonne National Laboratory.

Lehoucq, R. B., D. C. Sorensen, and C. Yang (1998). *ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. SIAM, 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. *Lecture Notes in Computer Science*, 1184:478–486.

MPI Forum (1994). MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416.

Parlett, B. N. (1998). *The Symmetric Eigenvalue Problem*. Prentice-Hall, Inc.

Reichel, L. and W. B. Gragg (1990). FORTRAN subroutines for updating the QR decomposition. *ACM Transactions on Mathematical Software*, 16:369–377.

Smith, B. T., J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler (1970). *Matrix Eigensystem Routines: EISPACK Guide*. New York, NY, USA, second edition.

Stewart, W. J. and A. Jennings (1981). Algorithm 570: LOPSI: A Simultaneous Iteration Method for Real Matrices [F2]. *ACM Transactions on Mathematical Software*, 7(2):230–232.

Wilkinson, J. H. and C. Reinsch (eds.) (1971). *Linear Algebra*, volume 2 of *Handbook for Automatic Computation*. New York, NY, USA.

Wu, K. and H. Simon (1997). A Parallel Lanczos Method for Symmetric Generalized Eigenvalue Problems. Technical Report LBNL-41284, Lawrence Berkeley National Laboratory.

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

Index