UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departament de Sistemes Informàtics i Computació
DSIC

Technical Report DSIC-II/24/02

SLEPc Users Manual

Scalable Library for Eigenvalue Problem Computations

https://slepc.upv.es

Jose E. Roman
Carmen Campos
Lisandro Dalcin
Eloy Romero
Andrés Tomás

To be used with SLEPc 3.26

September, 2026

Abstract

This manual 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 to solve various types of eigenvalue problems, including linear and nonlinear, as well as other related problems such as the singular value decomposition (see a summary of supported problem classes in table *SLEPc modules* (page i)). SLEPc is a general library in the sense that it covers both Hermitian and non-Hermitian problems, with either real or complex arithmetic.

The emphasis of the software is on methods and techniques appropriate for problems in which the associated matrices are large and sparse, for example, those arising after the discretization of partial differential equations. Thus, most of the methods offered by the library are projection methods, including different variants of Krylov and Davidson iterations. In addition to its own solvers, SLEPc provides transparent access to some external software packages such as ARPACK. Apart from the solvers, SLEPc also provides built-in support for some operations commonly used in the context of eigenvalue computations, such as preconditioning or the shift-and-invert spectral transformation.

SLEPc is built on top of PETSc, the Portable, Extensible Toolkit for Scientific Computation. SLEPc extends PETSc with all the functionality necessary for the solution of eigenvalue problems. This means that PETSc must be previously installed in order to use SLEPc. PETSc users will find SLEPc very easy to use, since it enforces the same programming paradigm. Those readers that are not acquainted with PETSc are highly recommended to familiarize with it before proceeding with SLEPc.

How to Get SLEPc

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

https://slepc.upv.es.

The distribution file is available for download at this site. Instructions for installing the software can be found in section *Installation* (page 3).

PETSc can be downloaded from https://petsc.org. PETSc is supported, and information on contacting support can be found at that site.

Additional Documentation

This manual provides a general description of SLEPc. In addition, manual pages for individual routines are available on-line at https://slepc.upv.es (both the C/Fortran API and the Python API). These manual pages provide hyperlinked access to the source code and enable easy movement among related topics. Finally, there are also several hands-on exercises available, which are intended for learning the basic concepts easily, see https://slepc.upv.es/documentation.

How to Read this Manual

Users that are already familiar with PETSc can read chapter *Getting Started* (page 1) very fast. Section *Eigenvalue Problems* (page 13) provides a brief overview of eigenproblems and the general concepts used by eigensolvers, so it can be skipped by experienced users. Chapters *EPS: Eigenvalue Problem Solver* (page 13) to *Auxiliary Classes* (page 89) describe the main SLEPc functionality. Some of them include an advanced usage section that can be skipped at a first reading. Finally, chapter *Additional Information* (page 95) contains less important, additional information.

SLEPc Technical Reports

The information contained in this manual is complemented by a set of Technical Reports available at https://slepc.upv.es/documentation. They provide technical details that normal users typically do not need to know but may be useful for experts in order to identify the particular method implemented in SLEPc.

Acknowledgments

The current version contains code contributed by several people. A list is given at https://slepc.upv.es/contact. That page also lists the funding sources that have made the development of SLEPc possible. We are thankful to all of them.

License and Copyright

Starting from version 3.8, SLEPc is released under a 2-clause BSD license (see `LICENSE` file).

Copyright 2002–2026 Universitat Politècnica de València, Spain

Supported Problem Classes

The following table provides an overview of the functionality offered by SLEPc, organized by problem classes.

Table 1: SLEPc modules

Problem class	Model equation	Module	Chapter
Linear eigenvalue problem	$Ax = \lambda x, \quad Ax = \lambda Bx$	EPS	*EPS: Eigenvalue Problem Solver* (page 13)
Polynomial eigenvalue problem	$(A_0 + \lambda A_1 + \cdots + \lambda^d A_d)x = 0$	PEP	*PEP: Polynomial Eigenvalue Problems* (page 61)
Nonlinear eigenvalue problem	$T(\lambda)x = 0$	NEP	*NEP: Nonlinear Eigenvalue Problems* (page 73)
Singular value decomposition	$Av = \sigma u$	SVD	*SVD: Singular Value Decomposition* (page 49)
Matrix function (action of)	$y = f(A)v$	MFN	*MFN: Matrix Function* (page 81)
Linear matrix equation	$AXE + DXB = C$	LME	*LME: Linear Matrix Equation* (page 85)

In order to solve a given problem, one should create a solver object corresponding to the solver class (module) that better fits the problem (the less general one; e.g., we do not recommend using NEP to solve a linear eigenproblem).

Notes

- Most users are typically interested in linear eigenproblems only.
- In each problem class there may exist several subclasses (problem types in SLEPc terminology), for instance symmetric-definite generalized eigenproblem in EPS.
- The solver class (module) is named after the problem class. For historical reasons, the one for linear eigenvalue problems is called EPS rather than LEP.
- In addition to the SVD shown in the table, the SVD module also supports other related problems such as the GSVD and the HSVD.
- For the action of a matrix function (MFN), in SLEPc we focus on methods that are closely related to methods for eigenvalue problems.

Contents

1 Getting Started **1**
- 1.1 SLEPc and PETSc . . . 2
- 1.2 Installation . . . 3
  - 1.2.1 Standard Installation . . . 4
  - 1.2.2 Configuration Options . . . 4
  - 1.2.3 Installing Multiple Configurations in a Single Directory Tree . . . 5
  - 1.2.4 Prefix-based Installation . . . 6
- 1.3 Running SLEPc Programs . . . 7
- 1.4 Writing SLEPc Programs . . . 7
  - 1.4.1 Simple SLEPc Example . . . 8
  - 1.4.2 Writing Application Codes with SLEPc . . . 10

2 EPS: Eigenvalue Problem Solver **13**
- 2.1 Eigenvalue Problems . . . 13
  - 2.1.1 Projection Methods . . . 14
  - 2.1.2 Preconditioned Eigensolvers . . . 14
  - 2.1.3 Related Problems . . . 15
- 2.2 Basic Usage . . . 15
- 2.3 Defining the Problem . . . 16
  - 2.3.1 Eigenvalues of Interest . . . 18
  - 2.3.2 Left Eigenvectors . . . 19
- 2.4 Selecting the Eigensolver . . . 19
- 2.5 Retrieving the Solution . . . 21
  - 2.5.1 The Computed Solution . . . 21
  - 2.5.2 Reliability of the Computed Solution . . . 22
  - 2.5.3 Controlling and Monitoring Convergence . . . 23
  - 2.5.4 Viewing the Solution . . . 26
- 2.6 Advanced Usage . . . 27
  - 2.6.1 Initial Guesses . . . 27
  - 2.6.2 Dealing with Deflation Subspaces . . . 27
  - 2.6.3 Orthogonalization . . . 28
  - 2.6.4 Specifying a Region for Filtering . . . 28
  - 2.6.5 Computing a Large Portion of the Spectrum . . . 28
  - 2.6.6 Computing Interior Eigenvalues with Harmonic Extraction . . . 29
  - 2.6.7 Balancing for Non-Hermitian Problems . . . 30
  - 2.6.8 Structured Eigenvalue Problems . . . 30

3 ST: Spectral Transformation **35**
- 3.1 General Description . . . 35
- 3.2 Basic Usage . . . 36
- 3.3 Available Transformations . . . 36
  - 3.3.1 Shift of Origin . . . 37
  - 3.3.2 Shift-and-invert . . . 38
  - 3.3.3 Cayley . . . 39
  - 3.3.4 Preconditioner . . . 39

3.3.5 Polynomial Filter . . . 40
3.4 Advanced Usage . . . 40
3.4.1 Solution of Linear Systems . . . 41
3.4.2 Explicit Computation of the Coefficient Matrix . . . 42
3.4.3 Preserving the Symmetry in Generalized Eigenproblems . . . 43
3.4.4 Purification of Eigenvectors . . . 44
3.4.5 Spectrum Slicing . . . 44
3.4.6 Spectrum Folding . . . 46

4 SVD: Singular Value Decomposition **49**
4.1 Mathematical Background . . . 49
4.1.1 The (Standard) Singular Value Decomposition (SVD) . . . 49
4.1.2 The Generalized Singular Value Decomposition (GSVD) . . . 51
4.1.3 The Hyperbolic Singular Value Decomposition (HSVD) . . . 52
4.2 Basic Usage . . . 53
4.3 Defining the Problem . . . 54
4.3.1 Using a threshold to specify wanted singular values . . . 55
4.4 Selecting the SVD Solver . . . 56
4.5 Retrieving the Solution . . . 58
4.5.1 Reliability of the Computed Solution . . . 59
4.5.2 Controlling and Monitoring Convergence . . . 59
4.5.3 Viewing the Solution . . . 60

5 PEP: Polynomial Eigenvalue Problems **61**
5.1 Overview of Polynomial Eigenproblems . . . 61
5.1.1 Quadratic Eigenvalue Problems . . . 61
5.1.2 Polynomials of Arbitrary Degree . . . 63
5.2 Basic Usage . . . 64
5.3 Defining the Problem . . . 64
5.4 Selecting the Solver . . . 66
5.5 Spectral Transformation for PEP . . . 67
5.5.1 Spectrum Slicing for PEP . . . 68
5.6 Retrieving the Solution . . . 69
5.6.1 Reliability of the Computed Solution . . . 69
5.6.2 Scaling . . . 70
5.6.3 Extraction . . . 70
5.6.4 Iterative Refinement . . . 70

6 NEP: Nonlinear Eigenvalue Problems **73**
6.1 General Nonlinear Eigenproblems . . . 73
6.2 Defining the Problem . . . 74
6.2.1 Using Callback Functions . . . 74
6.2.2 Expressing the NEP in Split Form . . . 76
6.3 Selecting the Solver . . . 77
6.4 Retrieving the Solution . . . 79

7 MFN: Matrix Function **81**
7.1 The Problem $f(A)v$ . . . 81
7.2 Basic Usage . . . 81
7.2.1 Defining the Problem . . . 82
7.2.2 Selecting the Solver . . . 83
7.2.3 Accuracy and Monitors . . . 83

8 LME: Linear Matrix Equation **85**
8.1 Current Functionality . . . 85
8.1.1 Continuous-Time Lyapunov Equation . . . 86
8.1.2 Krylov Solver . . . 86
8.1.3 Projected Problem . . . 86
8.2 Basic Usage . . . 87

9 Auxiliary Classes **89**
9.1 FN: Mathematical Functions . . . 89
9.2 RG: Region . . . 90
9.3 DS: Direct Solver (or Dense System) . . . 93
9.4 BV: Basis Vectors . . . 93

10 Additional Information **95**
10.1 Supported PETSc Features . . . 95
10.2 Supported Matrix Types . . . 96
10.2.1 Shell Matrices . . . 96
10.3 GPU Computing . . . 96
10.4 Extending SLEPc . . . 97
10.5 Directory Structure . . . 98
10.6 Wrappers to External Libraries . . . 99
10.6.1 List of External Libraries . . . 99
10.7 Fortran Interface . . . 102

Bibliography **105**

CHAPTER 1

Getting Started

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

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

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

The reasons for this situation are diverse. First, new methods are increasingly complex and difficult to implement and therefore robust implementations must be provided by computational specialists, for example as software libraries. The development of such libraries requires investing 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 and this leads to an explosion of available algorithms from which the user has to choose. Not all these algorithms are available in the form of software libraries, even less frequently with parallel capabilities.

Another difficulty resides in how to represent the operator matrix. Unlike in dense methods, there is no widely accepted standard for basic sparse operations in the spirit of BLAS. This is due to the fact that sparse storage is more complicated, admitting of more variation, and therefore less standardized. For this reason, sparse libraries have an added level of complexity. This holds even more so in the case of parallel distributed-memory programming, where the data of the problem have to be distributed across the available processors.

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

A better solution for the matrix representation problem is the well-known reverse communication interface, a technique that allows the development of iterative methods disregarding the implementation details of various operations. Whenever the iterative method subroutine needs the results of one of the operations, it returns control

to the user's subroutine that called it. The user's subroutine then invokes the module that performs the operation. The iterative method subroutine is invoked again with the results of the operation.

Several libraries with any of the interface schemes mentioned above are publicly available. For a survey of such software see the SLEPc Technical Report [Hernandez *et al.*, 2009] and references therein. Some of the most recent libraries are even prepared for parallel execution (some of them can be used from within SLEPc, see section *Wrappers to External Libraries* (page 99)). However, they still lack some flexibility or require too much programming effort from the user, especially in the case that the eigensolution requires to employ advanced techniques such as spectral transformations or preconditioning.

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

1.1 SLEPc and PETSc

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

SLEPc is based on PETSc, the Portable, Extensible Toolkit for Scientific Computation [Balay *et al.*, 2026], and, therefore, a large percentage of the software complexity is avoided since many PETSc developments are leveraged, including matrix storage formats and linear solvers, to name a few. SLEPc focuses on high level features for eigenproblems, structured around a few object classes as described below.

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

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

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

SLEPc can be considered an extension of PETSc providing all the functionality necessary for the solution of eigenvalue problems. Figure *Numerical components of PETSc and SLEPc* (page 3) 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.*, 2026].

Each of these components consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. Both PETSc and SLEPc are written in C, which lacks direct support for object-oriented programming. However, it is still possible to take advantage of the three basic principles of object-oriented programming to manage the complexity of such large packages. PETSc uses data *encapsulation* in

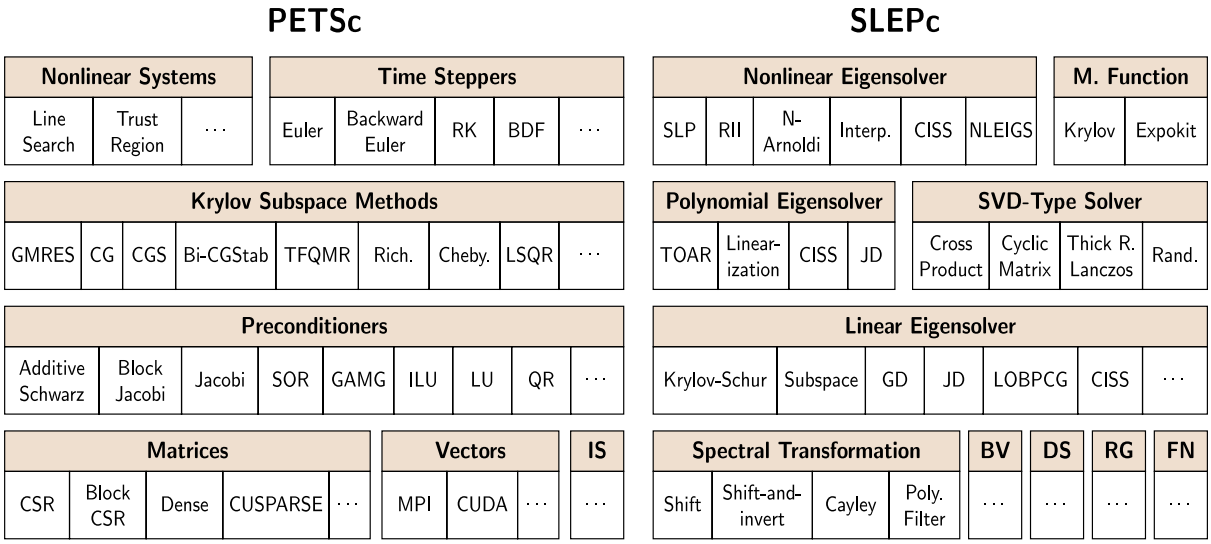


Fig. 1: Numerical components of PETSc and SLEPc

both vector and matrix data objects. Application code accesses data through function calls. Also, all the operations are supported through *polymorphism*. The user calls a generic interface routine, which then selects the underlying routine that handles the particular data structure. Finally, PETSc also uses *inheritance* in its design. All the objects are derived from an abstract base object. From this fundamental object, an abstract base object is defined for each PETSc object (Mat, Vec and so on), which in turn has a variety of instantiations that, for example, implement different matrix storage formats.

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

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

1.2 Installation

This section describes SLEPc's installation procedure. Previously to the installation of SLEPc, the system must have an appropriate version of PETSc installed. Compatible versions of PETSc and SLEPc are those with coincident major and minor version number, the third number (patch level) being irrelevant for this. For instance, SLEPc 3.26.1 may be built with PETSc 3.26.1. Also note that, if using git repositories, both PETSc and SLEPc must be either release versions or development versions, so make sure you select the appropriate branch in both repositories (`git checkout release` or `git checkout main`).

The installation process for SLEPc is very similar to PETSc, with two stages: configuration and compilation. SLEPc's configuration is much simpler because most of the configuration information is taken from PETSc, including compiler options and scalar type (real or complex). See section *Configuration Options* (page 4) for a discussion of options that are most relevant for SLEPc. Several configurations can coexist in the same directory tree, so that for instance one can have SLEPc libraries compiled with real scalars as well as with complex scalars. This is explained in section *Installing Multiple Configurations in a Single Directory Tree* (page 5). Also, system-based

installation is also possible with the `--prefix` option, as discussed in section *Prefix-based Installation* (page 6).

1.2.1 Standard Installation

The basic steps for the installation are described next. Note that prior to these steps, optional packages must have been installed. If any of these packages is installed afterwards, reconfiguration and recompilation is necessary. Refer to sections *Configuration Options* (page 4) and *Wrappers to External Libraries* (page 99) for details about installation of some of these packages.

1. Unbundle the distribution file with

   ```
   tar xzf slepc-3.26.1.tar.gz
   ```

   or an equivalent command. This will create a directory and unpack the software there.

2. Set the environment variable `SLEPC_DIR` to the full path of the SLEPc home directory. For example, under the `bash` shell:

   ```
   export SLEPC_DIR=/home/username/slepc-3.26.1
   ```

   In addition, the variables `PETSC_DIR` and `PETSC_ARCH` must also be set appropriately, e.g.

   ```
   export PETSC_DIR=/home/username/petsc-3.26.1
   export PETSC_ARCH=arch-darwin-c-debug
   ```

   The rationale for `PETSC_ARCH` is explained in section *Installing Multiple Configurations in a Single Directory Tree* (page 5) (see section *Prefix-based Installation* (page 6) for a case in which `PETSC_ARCH` is not required).

3. Change to the SLEPc directory and run the configuration script:

   ```
   $ cd $SLEPC_DIR
   $ ./configure
   ```

4. If the configuration was successful, build the libraries:

   ```
   $ make
   ```

5. After the compilation, try running some test examples with

   ```
   $ make check
   ```

   Examine the output for any obvious errors or problems.

1.2.2 Configuration Options

Several options are available in SLEPc's configuration script. To see all available options, type `./configure --help`.

In SLEPc, configure options have the following purposes:

- Specify a directory for prefix-based installation, as explained in section *Prefix-based Installation* (page 6).
- Enable compilation of Python bindings (slepc4py).
- Enable external eigensolver packages. For example, to use ARPACK, specify the following options (with the appropriate paths):

  ```
  $ ./configure --with-arpack-dir=/usr/software/ARPACK
  ```

  Some of the external packages also support the `--download-xxxx` option. Section *Wrappers to External Libraries* (page 99) provides more details related to use of external libraries.

Additionally, PETSc's configuration script provides a very long list of options that are relevant to SLEPc. Here is a list of options that may be useful. Note that these are options of PETSc that apply to both PETSc and SLEPc, in such a way that it is not possible to, e.g., build PETSc without debugging and SLEPc with debugging.

- Add `--with-scalar-type=complex` to build complex scalar versions of all libraries. See below a note related to complex scalars.

- Build single precision versions with `--with-precision=single`. In most applications, this can achieve a significant reduction of memory requirements, and a moderate reduction of computing time. Also, quadruple precision (128-bit floating-point representation) is also available using `--with-precision=__float128` on systems with GNU compilers (`gcc-4.6` or later).
- Enable use from Fortran. By default, PETSc's configure looks for an appropriate Fortran compiler. If not required, this can be disabled: `--with-fc=0`. If required but not correctly detected, the compiler to be used can be specified with a configure option. It is also possible to configure with a Fortran compiler but do not build Fortran interfaces of PETSc and SLEPc, with `--with-fortran-bindings=0`.
- If not detected, use `--with-blas-lapack-lib` to specify the location of BLAS and LAPACK. If SLEPc's configure complains about some missing LAPACK subroutines, reconfigure PETSc with option `--download-f2cblaslapack`.
- Enable external libraries that provide direct linear solvers or preconditioners, such as MUMPS, hypre, or SuperLU; for example, `--download-mumps`. These are especially relevant for SLEPc in the case that a spectral transformation is used, see chapter *ST: Spectral Transformation* (page 35).
- Add `--with-64-bit-indices=1` to use 8 byte integers (`long long`) for indexing in vectors and matrices. This is only needed when working with over roughly 2 billion unknowns.
- Build static libraries, `--with-shared-libraries=0`. This is generally not recommended, since shared libraries produce smaller executables and the run time overhead is small.
- Error-checking code can be disabled with `--with-debugging=0`, but this is only recommended in production runs of well-tested applications.
- Enable GPU computing setting `--with-cuda=1` or `--with-hip=1`; see section *GPU Computing* (page 96) for details.
- The option `--with-mpi=0` allows building PETSc and SLEPc without MPI support (only sequential).

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

1.2.3 Installing Multiple Configurations in a Single Directory Tree

Often, it is necessary to build two (or more) versions of the libraries that differ in a few configuration options. For instance, versions for real and complex scalars, or versions for double and single precision, or versions with debugging and optimized. In a standard installation, this is handled by building all versions in the same directory tree, as explained below, so that source code is not replicated unnecessarily. In contrast, in prefix-based installation where source code is not present, the issue of multiple configurations is handled differently, as explained in section *Prefix-based Installation* (page 6).

In a standard installation, the different configurations are identified by a unique name that is assigned to the environment variable `PETSC_ARCH`. Let us illustrate how to set up PETSc with two configurations. First, set a value of `PETSC_ARCH` and proceed with the installation of the first one:

```
$ cd $PETSC_DIR
$ export PETSC_ARCH=arch-linux-gnu-c-debug-real
$ ./configure --with-scalar-type=real
$ make all
```

Note that if `PETSC_ARCH` is not given a value, PETSc suggests one for us. After this, a subdirectory named `$PETSC_ARCH` is created within `$PETSC_DIR`, that stores all information associated with that configuration, including the built libraries, configuration files, automatically generated source files, and log files. For the second configuration, proceed similarly:

```
$ cd $PETSC_DIR
$ export PETSC_ARCH=arch-linux-gnu-c-debug-complex
$ ./configure --with-scalar-type=complex
$ make all
```

The value of PETSC_ARCH in this case must be different than the previous one. It is better to set the value of PETSC_ARCH explicitly, because the name suggested by `configure` may coincide with an existing value, thus overwriting a previous configuration. After successful installation of the second configuration, two $PETSC_ARCH directories exist within $PETSC_DIR, and the user can easily choose to build his/her application with either configuration by simply changing the value of PETSC_ARCH.

The configuration of two versions of SLEPc in the same directory tree is very similar. The only important restriction is that the value of PETSC_ARCH used in SLEPc must exactly match an existing PETSc configuration, that is, a directory $PETSC_DIR/$PETSC_ARCH must exist.

1.2.4 Prefix-based Installation

Both PETSc and SLEPc allow for prefix-based installation. This consists in specifying a directory to which the files generated during the building process are to be copied.

In PETSc, if an installation directory has been specified during configuration (with option `--prefix` in step *configuration* (page 4) of section *Standard Installation* (page 4)), then after building the libraries the relevant files are copied to that directory by typing

```
$ make install
```

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

To be more precise, suppose that the configuration was done with `--prefix=/opt/petsc-x.x-linux-gnu-c-debug`. Then, `make install` will create directory `/opt/petsc-x.x-linux-gnu-c-debug` if it does not exist, and several subdirectories containing the libraries, the configuration files, and the header files. Note that the source code files are not copied, nor the documentation, so the size of the installed directory will be much smaller than the original one. For that reason, it is no longer necessary to allow for several configurations to share a directory tree. In other words, in a prefix-based installation, variable PETSC_ARCH loses significance and must be unset. To maintain several configurations, one should specify different prefix directories, typically with a name that informs about the configuration options used.

In order to prepare a prefix-based installation of SLEPc that uses a prefix-based installation of PETSc, start by setting the appropriate value of PETSC_DIR. Then, run SLEPc's configure with a prefix directory.

```
export PETSC_DIR=/opt/petsc-3.26.1-linux-gnu-c-debug
unset PETSC_ARCH
cd $SLEPC_DIR
./configure --prefix=/opt/slepc-3.26.1-linux-gnu-c-debug
make
make install
export SLEPC_DIR=/opt/slepc-3.26.1-linux-gnu-c-debug
```

Note that the variable PETSC_ARCH has been unset before SLEPc's configure. SLEPc will use a temporary arch name during the build (this temporary arch is named `installed-arch-xxx`, where the `arch-xxx` string represents the configuration of the installed PETSc version). Although it is not a common case, it is also possible to configure SLEPc without prefix, in which case the PETSC_ARCH variable must still be empty and the arch directory `installed-xxx` is picked automatically (it is hardwired in file `$SLEPC_DIR/lib/slepc/conf/slepcvariables`). The combination PETSc without prefix and SLEPc with prefix is also allowed, in which case PETSC_ARCH should not be unset.

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 containing SLEPc. For example, under the bash shell, a command of the form

```
export SLEPC_DIR=/software/slepc-3.26.1
```

can be placed in the user's .bashrc file. The SLEPC_DIR directory can be either a standard installation SLEPc directory, or a prefix-based installation directory, see section *Prefix-based Installation* (page 6). In addition, the user must set the environment variables required by PETSc, that is, PETSC_DIR, to indicate the full path of the PETSc directory, and PETSC_ARCH to specify a particular architecture and set of options. Note that PETSC_ARCH should not be set in the case of prefix-based installations.

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). Usually, the mpiexec command can be used to initiate a program as in the following example that uses eight processes:

```
$ mpiexec -n 8 slepc_program [command-line options]
```

Note that MPI may be deactivated during configuration of PETSc, if one wants to run only serial programs in a laptop, for example.

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

1.4 Writing SLEPc Programs

Most SLEPc programs begin with a call to SlepcInitialize()

```
SlepcInitialize(int *argc,char ***args,const char file[],const char help[]);
```

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

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

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

```
SlepcFinalize();
```

This routine handles operations to be executed at the conclusion of the program, and calls PetscFinalize() if SlepcInitialize() began PETSc.

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

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

Section *Fortran Interface* (page 102) provides a summary of the differences between using SLEPc from Fortran and C/C++, as well as a complete Fortran example.

1.4.1 Simple SLEPc Example

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

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

#include <slepceps.h>

int main(int argc,char **argv)
{
  Mat            A;           /* problem matrix */
  EPS            eps;         /* eigenproblem solver context */
  EPSType        type;
  PetscReal      error,tol,re,im;
  PetscScalar    kr,ki;
  Vec            xr,xi;
  PetscInt       n=30,i,Istart,Iend,nev,maxit,its,nconv;

  PetscFunctionBeginUser;
  PetscCall(SlepcInitialize(&argc,&argv,NULL,help));

  PetscCall(PetscOptionsGetInt(NULL,NULL,"-n",&n,NULL));
  PetscCall(PetscPrintf(PETSC_COMM_WORLD,"\n1-D Laplacian Eigenproblem, n=%" PetscInt_FMT "\n\n",n));

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

  PetscCall(MatCreate(PETSC_COMM_WORLD,&A));
  PetscCall(MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n));
  PetscCall(MatSetFromOptions(A));

  PetscCall(MatGetOwnershipRange(A,&Istart,&Iend));
  for (i=Istart;i<Iend;i++) {
    if (i>0) PetscCall(MatSetValue(A,i,i-1,-1.0,INSERT_VALUES));
    if (i<n-1) PetscCall(MatSetValue(A,i,i+1,-1.0,INSERT_VALUES));
    PetscCall(MatSetValue(A,i,i,2.0,INSERT_VALUES));
  }
  PetscCall(MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY));
  PetscCall(MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY));

  PetscCall(MatCreateVecs(A,NULL,&xr));
  PetscCall(MatCreateVecs(A,NULL,&xi));

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

  /*
     Set operators. In this case, it is a standard eigenvalue problem
  */
  PetscCall(EPSSetOperators(eps,A,NULL));
  PetscCall(EPSSetProblemType(eps,EPS_HEP));

  /*
     Set solver parameters at runtime
  */
  PetscCall(EPSSetFromOptions(eps));

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

(continues on next page)

(continued from previous page)

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

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

  if (nconv>0) {
    /*
       Display eigenvalues and relative errors
    */
    PetscCall(PetscPrintf(PETSC_COMM_WORLD,
         "           k          ||Ax-kx||/||kx||\n"
         "   ----------------- ------------------\n"));

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

#if PetscDefined(USE_COMPLEX)
      re = PetscRealPart(kr);
      im = PetscImaginaryPart(kr);
#else
      re = kr;
      im = ki;
#endif
      if (im!=0.0) PetscCall(PetscPrintf(PETSC_COMM_WORLD," %9f%+9fi %12g\n",(double)re,(double)im,
↪(double)error));
      else PetscCall(PetscPrintf(PETSC_COMM_WORLD,"   %12f       %12g\n",(double)re,(double)error));
    }
    PetscCall(PetscPrintf(PETSC_COMM_WORLD,"\n"));
  }

  /*
     Free work space
  */
  PetscCall(EPSDestroy(&eps));
  PetscCall(MatDestroy(&A));
  PetscCall(VecDestroy(&xr));
  PetscCall(VecDestroy(&xi));
  PetscCall(SlepcFinalize());
  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 `slepcsys.h` (base SLEPc file). Some PETSc header files are included as well, such as `petscksp.h`. The SLEPc include files are located in the directory `${SLEPC_DIR}/include`.

The Options Database. All the PETSc functionality related to the options database is available in SLEPc. This allows the user to input control data at run time very easily. In this example, the call PetscOptionsGetInt(`NULL,NULL,"-n",&n,NULL`) checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged.

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

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

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

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

```
EPSCreate(MPI_Comm comm,EPS *eps);
EPSSetOperators(EPS eps,Mat A,Mat B);
EPSSetProblemType(EPS eps,EPSProblemType type);
EPSSetFromOptions(EPS eps);
EPSSolve(EPS eps);
EPSGetConverged(EPS eps,PetscInt *nconv);
EPSGetEigenpair(EPS eps,PetscInt 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 *EPS: Eigenvalue Problem Solver* (page 13) describes in detail the `EPS` package, including the options database that enables the user to customize the solution process at runtime by selecting the solution algorithm and also specifying the convergence tolerance, the number of eigenvalues, the dimension of the subspace, etc.

Spectral Transformation. In the example program shown above there is no explicit reference to spectral transformations. However, an `ST` object is handled internally so that the user is able to request different transformations such as shift-and-invert. Chapter *ST: Spectral Transformation* (page 35) describes the `ST` package in detail.

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

1.4.2 Writing Application Codes with SLEPc

Several example programs demonstrate the software usage and can serve as templates for developing custom applications. They are scattered throughout the SLEPc directory tree, in particular in the `tutorials` directories under each class subdirectory.

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

1. Install and test SLEPc according to the instructions given in *Installation* (page 3).
2. Copy the SLEPc example that corresponds to the class of problem of interest (e.g., singular value decomposition).

3. Create a makefile as explained below, compile and run the example program.
4. Use the example program as a starting point for developing a custom code.

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

```
default: ex1

include ${SLEPC_DIR}/lib/slepc/conf/slepc_common

ex1: ex1.o
        -${CLINKER} -o ex1 ex1.o ${SLEPC_EPS_LIB}
        ${RM} ex1.o

ex1f: ex1f.o
        -${FLINKER} -o ex1f ex1f.o ${SLEPC_EPS_LIB}
        ${RM} ex1f.o
```

Replace EPS in ${SLEPC_EPS_LIB} with the highest level module you are using in your program.

CHAPTER 2

EPS: Eigenvalue Problem Solver

The Eigenvalue Problem Solver (EPS) is the main object provided by SLEPc. It is used to specify a linear eigenvalue problem, either in standard or generalized form, and provides uniform and efficient access to all of the linear 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 systems of linear equations.

2.1 Eigenvalue Problems

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

In the standard formulation, the linear eigenvalue problem consists in determining $\lambda \in \mathbb{C}$ 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 $x \neq 0$ are called eigenvalue and (right) eigenvector, respectively. Note that they can be complex even when the matrix is real. If λ is an eigenvalue of A then $\bar{\lambda}$ is an eigenvalue of its conjugate transpose, A^*, or equivalently

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

where $y \neq 0$ is called the left eigenvector.

In many applications, the problem is formulated as

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

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

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

In practical analyses, from the n possible solutions, typically only a few eigenpairs (λ, x) are considered relevant, either in the extremities of the spectrum, in an interval, or in a region of the complex plane. Depending on the

application, either eigenvalues or eigenvectors (or both) are required. In some cases, left eigenvectors are also of interest.

2.1.1 Projection Methods

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

$$\tilde{\lambda}_i = \theta_i, \qquad \tilde{x}_i = V_j s_i. \tag{2.4}$$

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

The subspace

$$\mathcal{K}_m(A, v) \equiv \text{span}\left\{v, Av, A^2 v, \ldots, A^{m-1} v\right\}, \tag{2.5}$$

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

$$h_{j+1,j} v_{j+1} = w_j := A v_j - \sum_{i=1}^{j} h_{i,j} v_i, \tag{2.6}$$

where $h_{i,j}$ are the scalar coefficients obtained in the Gram-Schmidt orthogonalization of Av_j with respect to v_i, $i = 1, 2, \ldots, j$, and $h_{j+1,j} = \|w_j\|_2$. Then, the columns of V_j span the Krylov subspace $\mathcal{K}_j(A, v_1)$ and $Ax = \lambda x$ is projected onto $H_j s = \theta s$, where H_j is an upper Hessenberg matrix with elements $h_{i,j}$, which are 0 for $i \geq j+2$. The related Lanczos algorithms obtain a projected matrix that is tridiagonal.

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

It is generally assumed (and observed) that the Lanczos and Arnoldi algorithms find solutions at the extremities of the spectrum. Their convergence pattern, however, is strongly related to the eigenvalue distribution. Slow convergence may be experienced in the presence of tightly clustered eigenvalues. The maximum allowable j may be reached without having achieved convergence for all desired solutions. Then, restarting is usually a useful technique and different strategies exist for that purpose. However, convergence can still be very slow and acceleration strategies must be applied. Usually, these techniques consist 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 make it possible to obtain eigenvalues other than those lying in the periphery of the spectrum. On the other hand, the separation of the eigenvalues of interest is improved in the transformed spectrum thus leading to faster convergence. The most commonly used spectral transformation is called shift-and-invert, which works with operator $(A - \sigma I)^{-1}$. It allows the computation of 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.

2.1.2 Preconditioned Eigensolvers

In many applications, Krylov eigensolvers perform very well because Krylov subspaces are optimal in a certain theoretical sense. However, these methods may not be appropriate in some situations such as the computation of interior eigenvalues. The spectral transformation mentioned above may not be a viable solution or it may be too costly. For these reasons, other types of eigensolvers such as Davidson and Jacobi-Davidson rely on a different way of expanding the subspace. Instead of satisfying the Krylov relation, these methods compute the new basis vector by the so-called correction equation. The resulting subspace may be richer in the direction of the desired eigenvectors. These solvers may be competitive especially for computing interior eigenvalues. From a practical point of view, the

correction equation may be seen as a cheap replacement for the shift-and-invert system of equations, $(A-\sigma I)y = x$. By cheap we mean that it may be solved inaccurately without compromising robustness, via a preconditioned iterative linear solver. For this reason, these are known as *preconditioned* eigensolvers.

2.1.3 Related Problems

In many applications such as the analysis of damped vibrating systems the problem to be solved is a *polynomial eigenvalue problem* (PEP), or more generally a *nonlinear eigenvalue problem* (NEP). For these, the reader is referred to chapters *PEP: Polynomial Eigenvalue Problems* (page 61) and *NEP: Nonlinear Eigenvalue Problems* (page 73). Another linear algebra problem that is very closely related to the eigenvalue problem is the *singular value decomposition* (SVD), see chapter *SVD: Singular Value Decomposition* (page 49).

2.2 Basic Usage

The EPS module in SLEPc is used in a similar way as PETSc modules such as KSP. All the information related to an eigenvalue problem is handled via a context variable. The usual object management functions are available (`EPSCreate()`, `EPSDestroy()`, `EPSView()`, `EPSSetFromOptions()`). In addition, the EPS object provides functions for setting several parameters such as the number of eigenvalues to compute, the dimension of the subspace, the portion of the spectrum of interest, the requested tolerance or the maximum number of iterations allowed.

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

As an illustration of basic usage, see listing *Example code for basic solution with EPS* (page 15), that implements the solution of a simple standard eigenvalue problem. Code for setting up the matrix `A` is not shown and error-checking code is omitted.

Listing 1: Example code for basic solution with EPS

```
1  EPS         eps;       /*  eigensolver context  */
2  Mat         A;         /*  matrix of Ax=kx      */
3  Vec         xr, xi;    /*  eigenvector, x       */
4  PetscScalar kr, ki;    /*  eigenvalue, k        */
5  PetscInt    j, nconv;
6  PetscReal   error;
7
8  EPSCreate(PETSC_COMM_WORLD, &eps);
9  EPSSetOperators(eps, A, NULL);
10 EPSSetProblemType(eps, EPS_NHEP);
11 EPSSetFromOptions(eps);
12 EPSSolve(eps);
13 EPSGetConverged(eps, &nconv);
14 for (j=0;j<nconv;j++) {
15   EPSGetEigenpair(eps, j, &kr, &ki, xr, xi);
16   EPSComputeError(eps, j, EPS_ERROR_RELATIVE, &error);
17 }
18 EPSDestroy(&eps);
```

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

```
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 with the problem, as in line 9, with the following function:

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

The example specifies a standard eigenproblem. In the case of a generalized problem, it would be necessary also to provide matrix B as the third argument to the call. The matrices specified in this call can be in any PETSc format. In particular, EPS allows the user to solve matrix-free problems by specifying matrices created via MatCreateShell(). A more detailed discussion of this issue is given in section *Supported Matrix Types* (page 96).

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 *Defining the Problem* (page 16). 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 of *Example code for basic solution with EPS* (page 15):

```
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:

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

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

After the call to `EPSSolve()` has finished, all the data associated with the solution of the eigenproblem are kept internally. This information can be retrieved with different function calls, as in lines 13 to 17 of *Example code for basic solution with EPS* (page 15). This part is described in detail in section *Retrieving the Solution* (page 21).

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

```
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()` 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 *ST: Spectral Transformation* (page 35)). 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 function:

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

it is possible to examine the actual values of the different settings of the EPS object, including also those related to the associated ST object. This is useful for making sure that the solver is using the settings that the user wants.

2.3 Defining the Problem

SLEPc is able to cope with different kinds of problems. Currently supported problem types are listed in table *Problem types considered in EPS* (page 17). An eigenproblem is generalized ($Ax = \lambda Bx$) if the user has specified two matrices (see `EPSSetOperators()` discussed above), otherwise it is standard ($Ax = \lambda x$). A standard eigenproblem is Hermitian if matrix A is Hermitian (i.e., $A = A^*$) or, equivalently in the case of real matrices, if matrix A is symmetric (i.e., $A = A^T$). A generalized eigenproblem is Hermitian if matrix A is Hermitian (symmetric) and B is Hermitian (symmetric) and positive (semi-)definite. If B is not positive (semi-)definite then the problem cannot be considered Hermitian but symmetry can still be exploited to some extent in some solvers (problem type EPS_GHIEP). A special case of generalized non-Hermitian problem is when A is non-Hermitian but B is Hermitian and positive (semi-)definite, see section *Preserving the Symmetry in Generalized Eigenproblems* (page 43) and *Purification of Eigenvectors* (page 44) for discussion. The remaining entries in table *Problem types considered in*

EPS (page 17) correspond to structured eigenvalue problems, which are discussed in section *Structured Eigenvalue Problems* (page 30).

Table 1: Problem types considered in EPS

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
GNHEP with positive (semi-)definite B	EPS_PGNHEP	-eps_pos_gen_non_hermitian
Generalized Hermitian indefinite	EPS_GHIEP	-eps_gen_indefinite
Bethe-Salpeter	EPS_BSE	-eps_bse
Hamiltonian	EPS_HAMILT	-eps_hamiltonian
Linear Response	EPS_LREP	-eps_lrep

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

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

By default, SLEPc assumes that the problem is non-Hermitian. Some eigensolvers are able to exploit symmetry, that is, they compute a solution for Hermitian problems with less storage and/or computational cost than other methods that ignore this property. Also, symmetric solvers may be more accurate. On the other hand, some eigensolvers in SLEPc only have a symmetric version and will abort if the problem is non-Hermitian. In the case of generalized eigenproblems some considerations apply regarding symmetry, especially in the case of singular B. This topic is covered in section *Preserving the Symmetry in Generalized Eigenproblems* (page 43) and *Purification of Eigenvectors* (page 44). Similarly, if your eigenproblem has a particular algebraic structure listed in table *Problem types considered in EPS* (page 17), solving it with a structured eigensolver as discussed in section *Structured Eigenvalue Problems* (page 30) will result in more accuracy and better efficiency. For all these reasons, the user is strongly recommended to always specify the problem type in the source code.

The characteristics of the problem can be determined with the functions `EPSIsGeneralized()`, `EPSIsHermitian()`, `EPSIsPositive()`, and `EPSIsStructured()`.

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

```
EPSSetDimensions(EPS eps,PetscInt nev,PetscInt ncv,PetscInt mpd);
```

allows the specification of the number of eigenvalues to compute, `nev`. The next 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. The last argument has to do with a more advanced usage, as explained in section *Computing a Large Portion of the Spectrum* (page 28). These parameters can also be set at run time with the options `-eps_nev`, `-eps_ncv` and `-eps_mpd`. For example, the command line

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

requests 10 eigenvalues and instructs to use 24 column vectors. Note that `ncv` must be at least equal to `nev`, although in general it is recommended (depending on the method) to work with a larger subspace, for instance $\texttt{ncv} \geq 2 \cdot \texttt{nev}$ or even more. The case that the user requests a relatively large number of eigenpairs is discussed in section *Computing a Large Portion of the Spectrum* (page 28).

Instead of specifying the number of wanted eigenvalues `nev`, it is also possible to specify a threshold with:

```
EPSSetThreshold(EPS eps,PetscReal thres,PetscBool rel);
```

This usage is discussed for the case of `SVD` in section *Using a threshold to specify wanted singular values* (page 55). For details about the differences in case of `EPS`, we refer to the manual page of `EPSSetThreshold()`.

2.3.1 Eigenvalues of Interest

For the selection of the portion of the spectrum of interest, there are several alternatives. In real symmetric or complex Hermitian problems, one may want to compute the largest or smallest eigenvalues in magnitude, or the leftmost or rightmost ones, or even all eigenvalues in a given interval. 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. Sometimes the eigenvalues of interest are those closest to a given target value, τ, measuring the distance either in the ordinary way or along the real (or imaginary) axis. In some other cases, wanted eigenvalues must be found in a given region of the complex plane. Table *Available possibilities for selection of the eigenvalues of interest* (page 18) summarizes all the available possibilities to be selected with the following function:

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

This setting is used both for configuring how the eigensolver seeks eigenvalues (note that not all these possibilities are available for all the solvers) and also for sorting the computed values. The default is to compute the largest magnitude eigenvalues, except for those solvers in which this option is not available. There is another exception related to the use of some spectral transformations, see chapter *ST: Spectral Transformation* (page 35).

Table 2: Available possibilities for selection of the eigenvalues of interest

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]
EPS_TARGET_MAGNITUDE	-eps_target_magnitude	Smallest $\|\lambda - \tau\|$
EPS_TARGET_REAL	-eps_target_real	Smallest $\|\mathrm{Re}(\lambda - \tau)\|$
EPS_TARGET_IMAGINARY	-eps_target_imaginary	Smallest $\|\mathrm{Im}(\lambda - \tau)\|$
EPS_ALL	-eps_all	All $\lambda \in [a, b]$ or $\lambda \in \Omega$
EPS_WHICH_USER	-eps_which_user	*user-defined*

For the sorting criteria relative to a target value τ, the following function must be called in order to specify such value:

```
EPSSetTarget(EPS eps,PetscScalar target);
```

or, alternatively, with the command-line key `-eps_target`. Note that, since the target is defined as a PetscScalar, complex values of τ are allowed only in the case of complex scalar builds of the SLEPc library.

The use of a target value makes sense especially when the eigenvalues of interest are located in the interior of the spectrum. Since these eigenvalues are usually more difficult to compute, the eigensolver by itself may not be able to obtain them, and additional tools are normally required. There are two possibilities for this:

- To use harmonic extraction (see section *Computing Interior Eigenvalues with Harmonic Extraction* (page 29)), a variant of some solvers that allows a better approximation of interior eigenvalues without changing the way the subspace is built.
- To use a spectral transformation such as shift-and-invert (see chapter *ST: Spectral Transformation* (page 35)), where the subspace is built from a transformed problem (usually much more costly).

The special case of computing all eigenvalues in an interval is discussed in the next chapter (sections *Polynomial Filter* (page 40) and *Spectrum Slicing* (page 44)), since it is related also to spectral transformations. In this case, instead of a target value the user has to specify the computational interval with:

```
EPSSetInterval(EPS eps,PetscReal a,PetscReal b);
```

which is equivalent to `-eps_interval a,b`.

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

There is also support for specifying a region of the complex plane so that the eigensolver finds eigenvalues within that region only. This possibility is described in section *Specifying a Region for Filtering* (page 28). If *all* eigenvalues inside the region are required, then a contour-integral method must be used, as described in [Maeda *et al.*, 2016].

Finally, we mention the possibility of defining an arbitrary sorting criterion by means of `EPS_WHICH_USER` in combination with `EPSSetEigenvalueComparison()`.

The selection criteria discussed above are based solely on the eigenvalue. In some special situations, it is necessary to establish a user-defined criterion that also makes use of the eigenvector when deciding which are the most wanted eigenpairs. For these cases, use `EPSSetArbitrarySelection()`.

2.3.2 Left Eigenvectors

In addition to right eigenvectors, some solvers are able to compute also left eigenvectors, as defined in equation (2.2). The algorithmic variants that compute both left and right eigenvectors are usually called *two-sided*. By default, SLEPc computes right eigenvectors only. To compute also left eigenvectors, the user should set a flag by calling the following function before `EPSSolve()`:

```
EPSSetTwoSided(EPS eps,PetscBool twosided);
```

Note that in some problems such as Hermitian, generalized Hermitian, or some structured eigenproblems, the left eigenvector can be obtained trivially from the right eigenvector. Otherwise, the user must set the two-sided flag *before* the solver starts to compute, and this is restricted to just a few solvers, see table *Supported problem types for all eigensolvers available in SLEPc* (page 21).

2.4 Selecting the Eigensolver

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

- Basic methods (not recommended except for simple problems):
  - Power Iteration with deflation. When combined with shift-and-invert (see chapter *ST: Spectral Transformation* (page 35)), it is equivalent to the inverse iteration. Also, this solver embeds the Rayleigh Quotient iteration (RQI) by allowing variable shifts. Additionally, it provides the nonlinear inverse iteration method for the case that the problem matrix is a nonlinear operator (for this advanced usage, see `EPSPowerSetNonlinear()`).
  - Subspace Iteration with Rayleigh-Ritz projection and locking.
  - Arnoldi method with explicit restart and deflation.
  - Lanczos with explicit restart, deflation, and different reorthogonalization strategies.
- Krylov-Schur, a variation of Arnoldi with a very effective restarting technique. In the case of symmetric problems, this is equivalent to the thick-restart Lanczos method.
- Generalized Davidson, a simple iteration based on subspace expansion with the preconditioned residual.
- Jacobi-Davidson, a preconditioned eigensolver with an effective correction equation.
- RQCG, a basic conjugate gradient iteration for the minimization of the Rayleigh quotient.
- LOBPCG, the locally-optimal block preconditioned conjugate gradient.
- CISS, a contour-integral solver that allows computing all eigenvalues in a given region.
- Lyapunov inverse iteration, to compute rightmost eigenvalues.

Table 3: Eigenvalue solvers available in the EPS module

Method	EPSType	Options Database	Default
Power / Inverse / RQI	EPSPOWER	power	
Subspace Iteration	EPSSUBSPACE	subspace	
Arnoldi	EPSARNOLDI	arnoldi	
Lanczos	EPSLANCZOS	lanczos	
Krylov-Schur	EPSKRYLOVSCHUR	krylovschur	*
Generalized Davidson	EPSGD	gd	
Jacobi-Davidson	EPSJD	jd	
Rayleigh quotient CG	EPSRQCG	rqcg	
LOBPCG	EPSLOBPCG	lobpcg	
Contour integral SS	EPSCISS	ciss	
Lyapunov Inverse Iteration	EPSLYAPII	lyapii	
Wrapper to LAPACK	EPSLAPACK	lapack	
Wrapper to ARPACK	EPSARPACK	arpack	
Wrapper to BLOPEX	EPSBLOPEX	blopex	
Wrapper to PRIMME	EPSPRIMME	primme	
Wrapper to FEAST	EPSFEAST	feast	
Wrapper to ScaLAPACK	EPSSCALAPACK	scalapack	
Wrapper to ELPA	EPSELPA	elpa	
Wrapper to ELEMENTAL	EPSELEMENTAL	elemental	
Wrapper to EVSL	EPSEVSL	evsl	
Wrapper to CHASE	EPSCHASE	chase	

The default solver is Krylov-Schur. A detailed description of the implemented algorithms is provided in the SLEPc Technical Reports. In addition to these methods, SLEPc also provides wrappers to external packages such as ARPACK, or PRIMME. A complete list of these interfaces can be found in section *Wrappers to External Libraries* (page 99). Note that some of these packages (LAPACK, ScaLAPACK, ELPA, ELEMENTAL) perform *dense* computations and hence return the full eigendecomposition (furthermore, take into account that LAPACK is a sequential library so the corresponding solver should be used only for debugging purposes with small problem sizes).

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

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

while the user writes the options database command `-eps_type` followed by the name of the method (see table *Eigenvalue solvers available in the EPS module* (page 20)).

Not all the methods can be used for all problem types. Table *Supported problem types for all eigensolvers available in SLEPc* (page 21) summarizes the scope of each eigensolver by listing which portion of the spectrum can be selected (as defined in table *Available possibilities for selection of the eigenvalues of interest* (page 18)) and which problem types are supported (as defined in table *Problem types considered in EPS* (page 17)). Note that the structured problem types are not considered here, see section *Structured Eigenvalue Problems* (page 30). The table also indicates whether the solvers are available or not in the complex version of SLEPc, and if they have a two-sided variant.

Table 4: Supported problem types for all eigensolvers available in SLEPc

Method	Portion of spectrum	Problem type	Real/complex	Two-sided
power	Largest $\|\lambda\|$	any	both	yes
subspace	Largest $\|\lambda\|$	any	both	
arnoldi	any	any	both	
lanczos	any	EPS_HEP, EPS_GHEP	both	
krylovschur	any	any	both	yes
gd	any	any	both	
jd	any	any	both	
rqcg	Smallest $\mathrm{Re}(\lambda)$	EPS_HEP, EPS_GHEP	both	
lobpcg	Largest and smallest $\mathrm{Re}(\lambda)$	EPS_HEP, EPS_GHEP	both	
ciss	All λ in region	any	both	
lyapii	Largest $\mathrm{Re}(\lambda)$	any	both	
lapack	any	any	both	yes
arpack	any	any	both	
blopex	Smallest $\mathrm{Re}(\lambda)$	EPS_HEP, EPS_GHEP	both	
primme	Largest and smallest $\mathrm{Re}(\lambda)$	EPS_HEP, EPS_GHEP	both	
feast	All $\mathrm{Re}(\lambda)$ in an interval	any	both	
scalapack	All λ	EPS_HEP, EPS_GHEP	both	
elpa	All λ	EPS_HEP, EPS_GHEP	both	
elemental	All λ	EPS_HEP, EPS_GHEP	both	
evsl	All λ in interval	EPS_HEP	real	
chase	Smallest $\mathrm{Re}(\lambda)$	EPS_HEP	both	

2.5 Retrieving the Solution

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

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

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

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

2.5.1 The Computed Solution

The user may be interested in the eigenvalues, or the eigenvectors, or both. The next function:

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

returns the j-th computed eigenvalue/eigenvector pair. Typically, this function is called inside a loop for each value of j from 0 to nconv–1. Note that eigenvalues are ordered according to the same criterion specified with function EPSSetWhichEigenpairs() for selecting the portion of the spectrum of interest. The meaning of the last 4 arguments depends on whether SLEPc has been compiled for real or complex scalars, as detailed below. The eigenvectors are normalized so that they have a unit 2-norm, except for problem type EPS_GHEP in which case returned eigenvectors have a unit B-norm.

In case they are available, the left eigenvectors can be extracted with:

```
EPSGetLeftEigenvector(EPS eps,PetscInt j,Vec yr,Vec yi);
```

Real vs Complex SLEPc

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

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

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

2.5.2 Reliability of the Computed Solution

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

$$r = A\tilde{x} - \tilde{\lambda}\tilde{x}, \tag{2.7}$$

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

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

$$|\lambda - \tilde{\lambda}| \leq \|r\|_2, \tag{2.8}$$

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

In the case of non-Hermitian problems, the situation is worse because no simple relation such as equation (2.8) is available. This means that in this case the residual norms may still give an indication of the actual error but the user should be aware that they may sometimes be completely wrong, especially in the case of highly non-normal matrices. A better bound would involve also the residual norm of the left eigenvector.

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

$$\sin\theta(x,\tilde{x}) \leq \frac{\|r\|_2}{\delta}, \tag{2.9}$$

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

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

```
EPSGetInvariantSubspace(EPS eps,Vec v[]);
```

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

Computation of Bounds

It is sometimes useful to compute error bounds based on the norm of the residual r_j, to assess the accuracy of the computed solution. The bound can be made in absolute terms, as in equation (2.8), or alternatively the error can be expressed relative to the eigenvalue or to the matrix norms. For this, the following function can be used:

```
EPSComputeError(EPS eps,PetscInt j,EPSErrorType type,PetscReal *error);
```

The types of errors that can be computed are summarized in table *Available expressions for computing error bounds* (page 23). The way in which the error is computed is unrelated to the error estimation used internally in the solver for convergence checking, as described below. Also note that in the case of two-sided eigensolvers, the error bounds are based on $\max\{\|\ell_j\|, \|r_j\|\}$, where the left residual ℓ_j is defined as $\ell_j = A^*\tilde{y} - \bar{\tilde{\lambda}}\tilde{y}$.

Table 5: Available expressions for computing error bounds

Error type	EPSErrorType	Command line key	Error bound
Absolute error	EPS_ERROR_ABSOLUTE	-eps_error_absolute	$\|r\|$
Relative error	EPS_ERROR_RELATIVE	-eps_error_relative	$\|r\|/\|\lambda\|$
Backward error	EPS_ERROR_BACKWARD	-eps_error_backward	$\|r\|/(\|A\| + \|\lambda\|\|B\|)$

2.5.3 Controlling and Monitoring Convergence

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

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

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

The user specifies when a solution should be considered sufficiently accurate by means of a tolerance. An approximate eigenvalue is considered to be converged if the error estimate associated with it is below the specified tolerance. The default value of the tolerance is 10^{-8} (in case of PETSc/SLEPc built for double precision) and can be changed at run time with `-eps_tol <tol>` or inside the program with the function:

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

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

Convergence Check

The error estimates used for the convergence test are based on the residual norm, as discussed in section *Reliability of the Computed Solution* (page 22). Most eigensolvers explicitly compute the residual of the relevant eigenpairs during the iteration, but Krylov solvers use a cheap formula instead, allowing to track many eigenpairs simultaneously. When using a spectral transformation, this formula may give too optimistic bounds (corresponding to the residual of the transformed problem, not the original problem). In such cases, the users can force the computation of the residual with `EPSSetTrueResidual()`.

Table 6: Available possibilities for the convergence criterion

Convergence criterion	EPSConv	Command line key	Error bound
Absolute	EPS_CONV_ABS	-eps_conv_abs	$\|r\|$
Relative to eigenvalue	EPS_CONV_REL	-eps_conv_rel	$\|r\|/\|\lambda\|$
Relative to matrix norms	EPS_CONV_NORM	-eps_conv_norm	$\|r\|/(\|A\| + \|\lambda\|\|B\|)$
User-defined	EPS_CONV_USER	-eps_conv_user	*user function*

From the residual norm, the error bound can be computed in different ways, see table *Available possibilities for the convergence criterion* (page 23). This can be set via the corresponding command-line switch or with:

```
EPSSetConvergenceTest(EPS eps,EPSConv conv);
```

The default is to use the criterion relative to the eigenvalue (note: for computing eigenvalues close to the origin this criterion will likely give very poor accuracy, so the user is advised to use EPS_CONV_ABS in that case). Finally, a custom convergence criterion may be established by specifying a user function (EPSSetConvergenceTestFunction()).

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

By default, the eigensolver will stop iterating when the current number of eigenpairs satisfying the convergence test is equal to (or greater than) the number of requested eigenpairs (or if the maximum number of iterations has been reached). However, it is also possible to provide a user-defined stopping test that may decide to quit earlier, see EPSSetStoppingTest().

Monitors

Error estimates can be displayed during execution of the solution algorithm, as a way of monitoring convergence. There are several such monitors available. The user can activate them via the options database (see examples below), or within the code with EPSMonitorSet(). By default, the solvers run silently without displaying information about the iteration. Also, application programmers can provide their own routines to perform the monitoring by using the function EPSMonitorSet().

The most basic monitor prints one approximate eigenvalue together with its associated error estimate in each iteration. The shown eigenvalue is the first unconverged one.

```
$ ./ex9 -eps_nev 1 -eps_tol 1e-6 -eps_monitor

    1 EPS nconv=0 first unconverged value (error) -0.0695109+2.10989i (2.38956768e-01)
    2 EPS nconv=0 first unconverged value (error) -0.0231046+2.14902i (1.09212525e-01)
    3 EPS nconv=0 first unconverged value (error) -0.000633399+2.14178i (2.67086904e-02)
    4 EPS nconv=0 first unconverged value (error) 9.89074e-05+2.13924i (6.62097793e-03)
    5 EPS nconv=0 first unconverged value (error) -0.000149404+2.13976i (1.53444214e-02)
    6 EPS nconv=0 first unconverged value (error) 0.000183676+2.13939i (2.85521004e-03)
    7 EPS nconv=0 first unconverged value (error) 0.000192479+2.13938i (9.97563492e-04)
    8 EPS nconv=0 first unconverged value (error) 0.000192534+2.13938i (1.77259863e-04)
    9 EPS nconv=0 first unconverged value (error) 0.000192557+2.13938i (2.82539990e-05)
   10 EPS nconv=0 first unconverged value (error) 0.000192559+2.13938i (2.51440008e-06)
   11 EPS nconv=2 first unconverged value (error) -0.671923+2.52712i (8.92724972e-05)
```

Graphical monitoring (in an X display) is also available with -eps_monitor draw::draw_lg. Figure *Default convergence monitor* (page 25) shows the result of the following sample command line:

```
$ ./ex9 -n 200 -eps_nev 12 -eps_tol 1e-12 -eps_monitor draw::draw_lg -draw_pause .2
```

Again, only the error estimate of one eigenvalue is drawn. The spikes in the last part of the plot indicate convergence of one eigenvalue and switching to the next.

The two previously mentioned monitors have an alternative version (*_all) that processes all eigenvalues instead of just the first one. Figure *Simultaneous convergence monitor for all eigenvalues* (page 25) corresponds to the same example but with -eps_monitor_all draw::draw_lg. Note that these variants have a side effect: they force the computation of all error estimates even if the method would not normally do so.

A less verbose monitor is -eps_monitor_conv, which simply displays the iteration number at which convergence takes place. Note that several monitors can be used at the same time.

```
$ ./ex9 -n 200 -eps_nev 8 -eps_tol 1e-12 -eps_monitor_conv

   79 EPS converged value (error) #0 4.64001e-06+2.13951i (8.26091148e-13)
   79 EPS converged value (error) #1 4.64001e-06-2.13951i (8.26091148e-13)
   93 EPS converged value (error) #2 -0.674926+2.52867i (6.85260521e-13)
   93 EPS converged value (error) #3 -0.674926-2.52867i (6.85260521e-13)
   94 EPS converged value (error) #4 -1.79963+3.03259i (5.48825386e-13)
   94 EPS converged value (error) #5 -1.79963-3.03259i (5.48825386e-13)
```

(continues on next page)

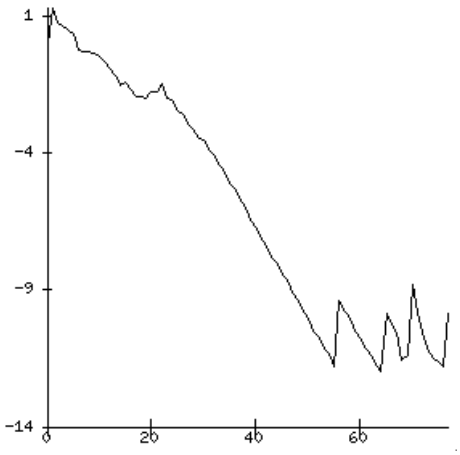


Fig. 1: Default convergence monitor

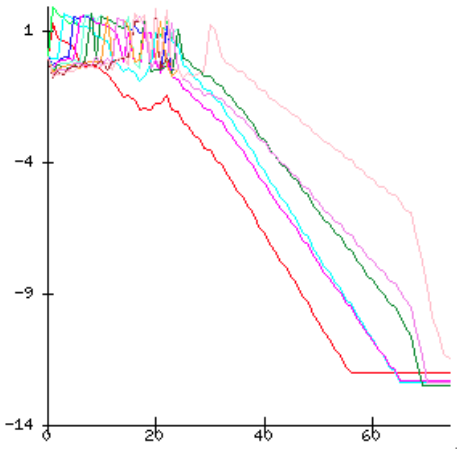


Fig. 2: Simultaneous convergence monitor for all eigenvalues

(continued from previous page)

```
    98 EPS converged value (error) #6 -3.37383+3.55626i (2.74909207e-13)
    98 EPS converged value (error) #7 -3.37383-3.55626i (2.74909207e-13)
```

2.5.4 Viewing the Solution

The computed solution (eigenvalues and eigenvectors) can be viewed in different ways, exploiting the flexibility of PetscViewers. The API functions for this are `EPSValuesView()` and `EPSVectorsView()`. We next illustrate their usage via the command line.

The command-line option `-eps_view_values` shows the computed eigenvalues on the standard output at the end of `EPSSolve()`. It admits an argument to specify PetscViewer options, for instance the following will create a Matlab command file `myeigenvalues.m` to load the eigenvalues in Matlab:

```
$ ./ex1 -n 120 -eps_nev 8 -eps_view_values :myeigenvalues.m:ascii_matlab
```

```
Lambda_EPS_0xb430f0_0 = [
3.9993259306070224e+00
3.9973041767976509e+00
3.9939361013742269e+00
3.9892239746533216e+00
3.9831709729353331e+00
3.9757811763634532e+00
3.9670595661733632e+00
3.9570120213355646e+00
];
```

One particular instance of this option is `-eps_view_values draw`, that will plot the computed approximations of the eigenvalues on an X window. See Figure *Eigenvalues plot* (page 26) for an example.

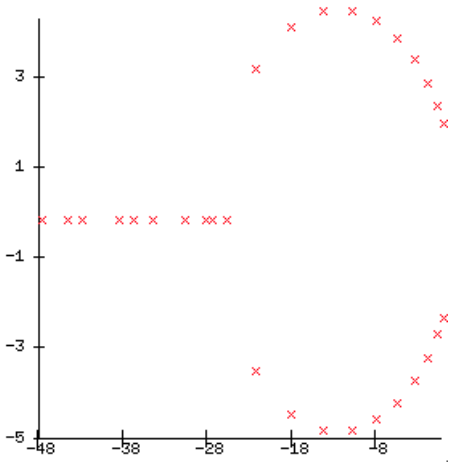


Fig. 3: Eigenvalues plot

Similarly, eigenvectors may be viewed with `-eps_view_vectors`, either in text form, in Matlab format, in binary format, or as a draw. All eigenvectors are viewed, one after the other. As an example, the next line will dump eigenvectors to the binary file `evec.bin`:

```
$ ./ex1 -n 120 -eps_nev 8 -eps_view_vectors binary:evec.bin
```

Two more related functions are available: `EPSErrorView()` and `EPSConvergedReasonView()`. These will show computed errors and the converged reason (plus number of iterations), respectively. Again, we illustrate its use via the command line. The option `-eps_error_relative` will show eigenvalues whose relative error are below the tolerance. The different types of errors have their corresponding options, see table *Available expressions for computing error bounds* (page 23). A more detailed output can be obtained as follows:

```
$ ./ex1 -n 120 -eps_nev 8 -eps_error_relative ::ascii_info_detail
     ---------------------- --------------------
                k              ||Ax-kx||/||kx||
     ---------------------- --------------------
            3.999326            1.26221e-09
            3.997304            3.82982e-10
            3.993936            2.76971e-09
            3.989224            4.94104e-10
            3.983171            6.19307e-10
            3.975781             5.9628e-10
            3.967060            2.32347e-09
            3.957012            6.12436e-09
     ---------------------- --------------------
```

Finally, the option for showing the converged reason is:

```
$ ./ex1 -n 120 -eps_nev 8 -eps_converged_reason
     Linear eigensolve converged (8 eigenpairs) due to CONVERGED_TOL; iterations 14
```

2.6 Advanced Usage

This section includes the description of advanced features of the eigensolver object. Default settings are appropriate for most applications and modification is unnecessary for normal usage.

2.6.1 Initial Guesses

In this subsection, we consider the possibility of providing initial guesses so that the eigensolver can exploit this information to get the answer faster.

Most of the algorithms implemented in EPS iteratively build and improve a basis of a certain subspace, which will eventually become an eigenspace corresponding to the wanted eigenvalues. In some solvers such as those of Krylov type, this basis is constructed starting from an initial vector, v_1, whereas in other solvers such as those of Davidson type, an arbitrary subspace can be used to start the method. By default, EPS initializes the starting vector or the initial subspace randomly. This default is a reasonable choice. However, it is also possible to supply an initial subspace with:

```
EPSSetInitialSpace(EPS eps,PetscInt n,Vec is[]);
```

In some cases, a suitable initial space can accelerate convergence significantly, for instance when the eigenvalue calculation is one of a sequence of closely related problems, where the eigenspace of one problem is fed as the initial guess for the next problem.

Note that if the eigensolver supports only a single initial vector, but several guesses are provided, then all except the first one will be discarded. One could still build a vector that is rich in the directions of all guesses, by taking a linear combination of them, but this is less effective than using a solver that considers all guesses as a subspace.

2.6.2 Dealing with Deflation Subspaces

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

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

$$l_{ij} = \begin{cases} d(v_i) & \text{if } i = j \\ -1 & \text{if } e_{ij} \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

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 with 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 with 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:

```
EPSSetDeflationSpace(EPS eps,PetscInt n,Vec defl[])
```

The value `n` indicates how many vectors are passed in argument `defl`.

The deflation space can be any subspace but typically it is most useful in the case of an invariant subspace or a null-space. In any case, SLEPc internally checks to see if all (or part of) the provided subspace is a null-space of the associated linear system (see section *Solution of Linear Systems* (page 41)). In this case, this null-space is attached to the coefficient matrix of the linear solver (see PETSc's function MatSetNullSpace()) to enable the solution of singular systems. In practice, this allows the computation of eigenvalues of singular pencils (i.e., when A and B share a common null-space).

2.6.3 Orthogonalization

Internally, eigensolvers in `EPS` often need to orthogonalize a vector against a set of vectors (for instance, when building an orthonormal basis of a Krylov subspace). This operation is carried out typically by a Gram-Schmidt orthogonalization procedure. The user is able to adjust several options related to this algorithm, although the default behavior is good for most cases, and we strongly suggest not to change any of these settings. This topic is covered in detail in [Hernandez *et al.*, 2007].

2.6.4 Specifying a Region for Filtering

Some solvers in `EPS` (and other solver classes as well) can take into consideration a user-defined region of the complex plane, e.g., an ellipse. A region can be used for two purposes:

- To instruct the eigensolver to compute all eigenvalues lying in that region. This is available only in eigensolvers based on the contour integral technique (`EPSCISS`).
- To filter out eigenvalues outside the region. In this way, eigenvalues lying inside the region get higher priority during the iteration and are more likely to be returned as computed solutions.

Regions are specified by means of an `RG` object. This object is handled internally in the `EPS` solver, as other auxiliary objects, and can be extracted with `EPSGetRG()` to set the options that define the region. These options can also be set in the command line. The following example computes largest magnitude eigenvalues, but restricting to an ellipse of radius 0.5 centered at the origin (with vertical scale 0.1):

```
$ ./ex1 -rg_type ellipse -rg_ellipse_center 0 -rg_ellipse_radius 0.5 -rg_ellipse_vscale 0.1
```

If one wants to use the region to specify where eigenvalues should *not* be computed, then the region must be the complement of the specified one. The next command line computes the smallest eigenvalues not contained in the ellipse:

```
$ ./ex1 -eps_smallest_magnitude -rg_type ellipse -rg_ellipse_center 0 -rg_ellipse_radius 0.5 -rg_ellipse_
↪vscale 0.1 -rg_complement
```

Additional details of the `RG` class can be found in section *RG: Region* (page 90).

2.6.5 Computing a Large Portion of the Spectrum

We now consider the case when the user requests a relatively large number of eigenpairs (the related case of computing all eigenvalues in a given interval is addressed in section *Spectrum Slicing* (page 44)). To fix ideas, suppose that the problem size (the dimension of the matrix, denoted as `n`), is in the order of 100,000's, and the

user wants nev to be approximately 5,000 (recall the notation of EPSSetDimensions() in section *Defining the Problem* (page 16)).

The first comment is that for such large values of nev, the rule of thumb suggested in section *Defining the Problem* (page 16) for selecting the value of ncv ($\texttt{ncv} \geq 2 \cdot \texttt{nev}$) may be inappropriate. For small values of nev, this rule of thumb is intended to provide the solver with a sufficiently large subspace. But for large values of nev, it may be enough setting ncv to be slightly larger than nev.

The second thing to take into account has to do with costs, both in terms of storage and in terms of computational effort. This issue is dependent on the particular eigensolver used, but generally speaking the user can simplify to the following points:

1. It is necessary to store a basis of the subspace, that is, ncv vectors of length n.
2. A considerable part of the computation is devoted to orthogonalization of the basis vectors, whose cost is roughly of order $\texttt{ncv}^2 \cdot \texttt{n}$.
3. Within the eigensolution process, a projected eigenproblem of order ncv is built. At least one dense matrix of this dimension has to be stored.
4. Solving the projected eigenproblem has a computational cost of order $\texttt{ncv}^3$. Typically, such problems need to be solved many times within the eigensolver iteration.

It is clear that a large value of ncv implies a high storage requirement (points 1 and 3, especially point 1), and a high computational cost (points 2 and 4, especially point 2). However, in a scenario of such big eigenproblems, it is customary to solve the problem in parallel with many processors. In that case, it turns out that the basis vectors are stored in a distributed way and the associated operations are parallelized, so that points 1 and 2 become benign as long as sufficient processors are used. Then points 3 and 4 become really critical since in the current SLEPc version the projected eigenproblem (and its associated operations) are not treated in parallel. In conclusion, the user must be aware that using a large ncv value introduces a serial step in the computation with high cost, that cannot be amortized by increasing the number of processors.

From SLEPc 3.0.0, another parameter mpd has been introduced to alleviate this problem. The name mpd stands for maximum projected dimension. The idea is to bound the size of the projected eigenproblem so that steps 3 and 4 work with a dimension of mpd at most, while steps 1 and 2 still work with a bigger dimension, up to ncv. Suppose we want to compute nev=5000. Setting ncv=10000 or even ncv=6000 would be prohibitively expensive, for the reasons explained above. But if we set e.g. mpd=600 then the overhead of steps 3 and 4 will be considerably diminished. Of course, this reduces the potential of approximation at each outer iteration of the algorithm, but with more iterations the same result should be obtained. The benefits will be specially noticeable in the setting of parallel computation with many processors.

Note that it is not necessary to set both ncv and mpd. For instance, one can do

```
$ ./program -eps_nev 5000 -eps_mpd 600
```

2.6.6 Computing Interior Eigenvalues with Harmonic Extraction

The standard Rayleigh-Ritz projection procedure described in section *Eigenvalue Problems* (page 13) is most appropriate for approximating eigenvalues located at the periphery of the spectrum, especially those of largest magnitude. Most eigensolvers in SLEPc are restarted, meaning that the projection is carried out repeatedly with increasingly good subspaces. An effective restarting mechanism, such as that implemented in Krylov-Schur, improves the subspace by realizing a filtering effect that tries to eliminate components in the direction of unwanted eigenvectors. In that way, it is possible to compute eigenvalues located anywhere in the spectrum, even in its interior.

Even though in theory eigensolvers could be able to approximate interior eigenvalues with a standard extraction technique, in practice convergence difficulties may arise that prevent success. The problem comes from the property that Ritz values (the approximate eigenvalues provided by the standard projection procedure) converge from the interior to the periphery of the spectrum. That is, the Ritz values that stabilize first are those in the periphery, so convergence of interior ones requires the previous convergence of all eigenvalues between them and the periphery. Furthermore, this convergence behavior usually implies that restarting is carried out with bad approximations, so the restart is ineffective and global convergence is severely damaged.

Harmonic projection is a variation that uses a target value, τ, around which the user wants to compute eigenvalues (see, e.g., [Morgan and Zeng, 2006]). The theory establishes that harmonic Ritz values converge in such a way

that eigenvalues closest to the target stabilize first, and also that no unconverged value is ever close to the target, so restarting is safe in this case. As a conclusion, eigensolvers with harmonic extraction may be effective in computing interior eigenvalues. Whether it works or not in practical cases depends on the particular distribution of the spectrum.

In order to use harmonic extraction in SLEPc, it is necessary to indicate it explicitly, and provide the target value as described in section *Defining the Problem* (page 16) (default is $\tau = 0$). The type of extraction can be set with:

```
EPSSetExtraction(EPS eps,EPSExtraction extr);
```

Available possibilities are `EPS_RITZ` for standard projection and `EPS_HARMONIC` for harmonic projection (other alternatives such as refined extraction are still experimental).

A command line example would be:

```
$ ./ex5 -m 45 -eps_harmonic -eps_target 0.8 -eps_ncv 60
```

The example computes the eigenvalue closest to $\tau = 0.8$ of a real non-symmetric matrix of order 1035. Note that `ncv` has been set to a larger value than would be necessary for computing the largest magnitude eigenvalues. In general, users should expect a much slower convergence when computing interior eigenvalues compared to extreme eigenvalues. Increasing the value of `ncv` may help.

Currently, harmonic extraction is available in the default `EPS` solver, that is, Krylov-Schur, and also in Arnoldi, GD, and JD.

2.6.7 Balancing for Non-Hermitian Problems

In problems where the matrix has a large norm, $\|A\|_2$, the roundoff errors introduced by the eigensolver may be large. The goal of balancing is to apply a simple similarity transformation, DAD^{-1}, that keeps the eigenvalues unaltered but reduces the matrix norm, thus enhancing the accuracy of the computed eigenpairs. Obviously, this makes sense only in the non-Hermitian case. The matrix D is chosen to be diagonal, so balancing amounts to scaling the matrix rows and columns appropriately.

In SLEPc, the matrix DAD^{-1} is not formed explicitly. Instead, the operators of table *Operators used in each spectral transformation mode* (page 37) are preceded by a multiplication by D^{-1} and followed by a multiplication by D. This allows for balancing in the case of problems with an implicit matrix.

A simple and effective Krylov balancing technique, described in [Chen and Demmel, 2000], is implemented in SLEPc. The user calls the following subroutine to activate it:

```
EPSSetBalance(EPS eps,EPSBalance bal,PetscInt its,PetscReal cutoff);
```

Two variants are available, one-sided and two-sided, and there is also the possibility for the user to provide a pre-computed D matrix.

2.6.8 Structured Eigenvalue Problems

Structured eigenvalue problems are those whose defining matrices are structured, i.e., their n^2 entries depend on less than n^2 parameters. Symmetry is the most obvious structure, and it is supported in SLEPc solvers via the `EPS_HEP` and `EPS_GHEP` problem types, see table *Problem types considered in EPS* (page 17). The last entries listed in that table address other types of structured eigenproblems, which are discussed in this subsection. Preserving the algebraic structure can help preserve physically relevant symmetries in the eigenvalues of the matrix and may improve the accuracy and efficiency of the eigensolver. For example, in quadratic eigenvalue problems arising from gyroscopic systems (see section *Quadratic Eigenvalue Problems* (page 61)), eigenvalues appear in quadruples $\{\lambda, -\lambda, \bar{\lambda}, -\bar{\lambda}\}$, i.e., the spectrum is symmetric with respect to both the real and imaginary axes. This problem can be linearized to a $2n \times 2n$ skew-Hamiltonian/Hamiltonian pencil with the same eigenvalues. A structure-preserving eigensolver will give a more accurate answer because it enforces the structure throughout the computation.

Unless otherwise stated, the structured eigenproblems discussed below are only supported in the default `EPS` solver, Krylov-Schur. The idea is that the user creates the structured matrix with a helper function such as `MatCreateBSE()` (see below), and then selects the appropriate problem type with `EPSSetProblemType()`, in this case `EPS_BSE`. This will instruct the solver to exploit the problem structure. Alternatively, one can solve the problem as `EPS_NHEP`, in which case the solver will neglect the structure.

Warning

Check below the section *Extracting Eigenvectors of Structured Eigenproblems* (page 32), since the trivial approach may give vectors with disordered entries.

Bethe-Salpeter

One structured eigenproblem that has raised interest recently is related to the Bethe–Salpeter equation, which is relevant in many state-of-the-art computational physics codes. For instance, in the Yambo software [Sangalli *et al.*, 2019] it is used to evaluate optical properties of materials. It is commonly formulated as an eigenvalue problem with a block-structured matrix,

$$H = \begin{bmatrix} R & C \\ -C^* & -R^T \end{bmatrix}, \tag{2.11}$$

where R is Hermitian and C is complex symmetric. For a good enough approximation of the optical absorption spectrum, it is sufficient to compute a few eigenvalues with a customized version of Krylov-Schur. In this problem, eigenvalues are real and come in pairs $\{\lambda, -\lambda\}$. The eigenvalues of interest are those with the smallest magnitude, which in this case lie in the middle of the spectrum. Usually, both right and left eigenvectors are required, but the left eigenvectors can be obtained inexpensively once the corresponding right ones are known.

The helper function to generate the matrix H of equation (2.11) from the blocks R and C is `MatCreateBSE()`, and the associated problem type is `EPS_BSE` (or `-eps_bse` from the command line). It is possible to select a few variants of the solver with the function `EPSKrylovSchurSetBSEType()`.

Further details about the implementation of the SLEPc solvers for the BSE can be found in [Alvarruiz *et al.*, 2025]. See also the section on *Linear Response* (page 31) below.

Hamiltonian

The Hamiltonian structure is relevant in many applications, particularly in control theory. A (complex) Hamiltonian matrix has the block structure

$$H = \begin{bmatrix} A & B \\ C & -A^* \end{bmatrix}, \tag{2.12}$$

where A, B and C are either real with $B = B^T$, $C = C^T$, or complex with $B = B^*$, $C = C^*$. In the real case, eigenvalues appear in pairs $\{\lambda, -\lambda\}$ and for complex eigenvalues in quadruples $\{\lambda, -\lambda, \bar{\lambda}, -\bar{\lambda}\}$. For a complex Hamiltonian matrix, if λ is an eigenvalue, then $-\bar{\lambda}$ is also an eigenvalue. A structure-preserving eigensolver has been implemented in `EPS` (in Krylov-Schur), which is activated by selecting the `EPS_HAMILT` problem type (or `-eps_hamiltonian` from the command line). Note that matrix H (2.12) must be created with the helper function `MatCreateHamiltonian()` in order to use this solver.

Warning

The structure-preserving eigensolver for Hamiltonian eigenvalue problems should be considered experimental. Depending on the problem, it may become numerically unstable after some iterations, in which case the solver will abort, returning less eigenvalues than requested.

Linear Response

The Linear Response Eigenvalue Problem (LREP) can be seen as a real version of the *Bethe-Salpeter* (page 31) eigenproblem, and appears in TDDFT simulations and many other computational physics applications. We can express the problem with two equivalent formulations. In the first one, the matrix is

$$H_1 = \begin{bmatrix} A & B \\ -B & -A \end{bmatrix}, \tag{2.13}$$

where both A and B are real symmetric. As in the BSE, the eigenvalues are real and come in pairs $\{\lambda, -\lambda\}$, and the eigenvalues of interest are those with the smallest magnitude. This problem can be addressed with SLEPc's functionality for the BSE, but we also provide specific solvers for the LREP.

The second formulation (that we will call *reduced*) is with matrix

$$H_2 = \begin{bmatrix} 0 & K \\ M & 0 \end{bmatrix}, \tag{2.14}$$

where $K = A - B$ and $M = A + B$. Note that, in applications, generally it is required that at least one of K or M is positive definite. The two formulations are equivalent in the sense that it is possible to switch from one to the other via a similarity transformation, i.e, the eigenvalues of both matrices are the same. The advantage of the second formulation is that it can also be used for product eigenvalue problems, that is, when one is interested in computing eigenvalues of KM or MK, with symmetric matrices not necessarily coming from a LREP.

The helper function `MatCreateLREP()` can be used to generate the matrices H_1 or H_2 from the individual blocks. The problem type must be set to `EPS_LREP` (or `-eps_lrep` from the command line). Note that internally the SLEPc eigensolver will work with the reduced form, and it will carry out the transformation transparently if the first form is provided by the user.

Extracting Eigenvectors of Structured Eigenproblems

In the case of structured eigenproblems, one has to create the eigenvectors in a specific way, before passing them to functions such as `EPSGetEigenvector()` or `EPSGetLeftEigenvector()`. Otherwise, the obtained vector may have disordered entries when run with several processes, compared to sequential runs.

To understand the problem, it is important to note that the block matrices (2.11) or (2.12) are represented as PETSc matrices of type MATNEST, which implies that the individual blocks are stored independently (in parallel). The vectors associated with those matrices have the same distribution. For instance, if we are using two MPI processes, then process 0 will store the first half of the top part of the vector, and the first half of the bottom part, while process 1 will store the second half of both parts. Hence, the entries assigned to the different processes are not consecutive, but interleaved, so if we access that vector as a standard Vec the order of entries will be different when we change the number of processes.

The solution is to work with the top and bottom subvectors separately. There are three possible approaches for this, as explained next.

The first solution is to create a vector of type VECNEST compatible with the MATNEST. For instance in BSE, (2.11), we can create it from the R and C blocks, as follows (assuming both matrices have the same local and global sizes):

```
MatCreateVecs(R,&x1,&x2);
aux[0] = x1;
aux[1] = x2;
VecCreateNest(PETSC_COMM_WORLD,2,NULL,aux,&x);
```

We would pass `x` to `EPSGetEigenvector()`, and then the individual blocks of the eigenvector can be easily extracted with VecNestGetSubVec().

The second alternative is to retrieve the index sets IS that define the splitting

```
MatNestGetISs(H,is,NULL);
```

and then, after calling `EPSGetEigenvector()` on a standard vector `x`, extract the subvectors using the index sets:

```
VecGetSubVector(x,is[0],&x1);
VecGetSubVector(x,is[1],&x2);
/* do something with x1 and x2 */
VecRestoreSubVector(x,is[0],&x1);
VecRestoreSubVector(x,is[1],&x2);
```

Finally, a third possibility is to modify the vector type of the nested matrix, so that it creates nested vectors. For instance:

```
MatCreateBSE(R,C,&H);
MatNestSetVecType(H,VECNEST);
```

Then, if we create `x` with MatCreateVecs(`H`,`&x`,`NULL`) then it will be a VECNEST as in the first solution, and we can use VecNestGetSubVec() to extract the individual blocks.

CHAPTER 3

ST: Spectral Transformation

The Spectral Transformation (ST) is the object that encapsulates the functionality required for acceleration techniques based on the transformation of the spectrum. Most eigensolvers in EPS work by applying an operator to a set of vectors and this operator can adopt different forms. The ST object handles all the different possibilities in a uniform way, so that the solver can proceed without knowing which transformation has been selected. The spectral transformation can be specified at run time, together with related options such as which linear solver to use.

Despite being a rather unrelated concept, ST is also used to handle the preconditioners and correction-equation solvers used in preconditioned eigensolvers such as GD and JD.

The description in this chapter focuses on the use of ST in the context of EPS. For usage within other solver classes, we will provide further details, e.g., shift-and-invert for polynomial eigenproblems in section *Spectral Transformation for PEP* (page 67).

3.1 General Description

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

- Compute internal eigenvalues. In some applications, the eigenpairs of interest are not the extreme ones (largest magnitude, 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, most eigensolvers contained in EPS are implemented in such a way that they are independent of which transformation has been selected by the user (the exception are preconditioned solvers, see below). That is, the solver algorithm has to work with a generic operator, whose actual form depends on the transformation used. After convergence, eigenvalues are transformed back appropriately.

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

Preconditioners: As explained in the previous chapter, EPS contains preconditioned eigensolvers such as GD or JD. These solvers either apply a preconditioner at a certain step of the computation, or need to solve a correction equation with a preconditioned linear solver. One of the main goals of these solvers is to achieve a similar effect as an inverse-based spectral transformation such as shift-and-invert, but with less computational cost. For this reason, a "preconditioner" spectral transformation has been included in the ST object. However, this is just a convenient way of organizing the functionality, since this fake spectral transform cannot be used with non-preconditioned eigensolvers, and conversely preconditioned eigensolvers cannot be used with conventional spectral transformations.

3.2 Basic Usage

The ST module is the analog of some PETSc modules such as PC. The user does not usually need to create a standalone 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 function 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 with the ST object. For example, to set the value of the shift, σ, the following function is available:

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

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

> **Warning**
>
> Usually, `STSetShift()` is never called from application code, as the value of the shift is taken from the target. In all transformations except `STSHIFT`, there is a direct connection between the target τ (described in section *Eigenvalues of Interest* (page 18)) and the shift σ, as will be discussed below. The normal usage is that the user sets the target and then σ is set to τ automatically (though it is still possible for the user to set a different value of the shift).

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

3.3 Available Transformations

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

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

The ST type can also be set with the command-line option `-st_type` followed by the name of the method (see table *Spectral transformations available in the ST package* (page 37)). The first five spectral transformations are described in detail in the rest of this section. The last possibility, `STSHELL`, uses a specific, application-provided spectral transformation. Section *Extending SLEPc* (page 97) in the final chapter describes how to implement one of these transformations.

Table 1: Spectral transformations available in the ST package

Spectral Transformation	STType	Options Name	Operator
Shift of Origin	STSHIFT	shift	$B^{-1}A - \sigma I$
Shift-and-invert	STSINVERT	sinvert	$(A - \sigma B)^{-1}B$
Generalized Cayley	STCAYLEY	cayley	$(A - \sigma B)^{-1}(A + \nu B)$
Preconditioner	STPRECOND	precond	$K^{-1} \approx (A - \sigma B)^{-1}$
Polynomial Filter	STFILTER	filter	$p(A)$
Shell Transformation	STSHELL	shell	*user-defined*

The last column of table *Spectral transformations available in the ST package* (page 37) shows a general form of the operator used in each case. This generic operator can adopt different particular forms depending on whether the eigenproblem is standard or generalized, or whether the value of the shift (σ) and anti-shift (ν) is zero or not. All the possible combinations are listed in table *Operators used in each spectral transformation mode* (page 37).

Table 2: Operators used in each spectral transformation mode

ST	Choice of σ, ν	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	$\sigma \neq 0, \nu = 0$	$(A - \sigma I)^{-1}A$	$(A - \sigma B)^{-1}A$
	$\sigma = 0, \nu \neq 0$	$I + \nu A^{-1}$	$I + \nu A^{-1}B$
	$\sigma \neq 0, \nu \neq 0$	$(A - \sigma I)^{-1}(A + \nu I)$	$(A - \sigma B)^{-1}(A + \nu B)$
precond	$\sigma = 0$	$K^{-1} \approx A^{-1}$	$K^{-1} \approx A^{-1}$
	$\sigma \neq 0$	$K^{-1} \approx (A - \sigma I)^{-1}$	$K^{-1} \approx (A - \sigma B)^{-1}$

The expressions shown in table *Operators used in each spectral transformation mode* (page 37) 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 system of linear equations that is managed by setting up an associated KSP object. The user can control the behavior of this object by adjusting the appropriate options, as will be illustrated with examples in section *Solution of Linear Systems* (page 41).

3.3.1 Shift of Origin

By default, no spectral transformation is performed. This is equivalent to a shift of origin (STSHIFT) with $\sigma = 0$, that is, the first line of table *Operators used in each spectral transformation mode* (page 37). The solver works with the original expressions of the eigenvalue problems,

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

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

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

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

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

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

for standard problems, and

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

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

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

This means that after the solution process, the value σ has to be added[1] to the computed eigenvalues, θ, in order to retrieve the solution of the original problem, λ. This is done by means of the function `STBackTransform()`, which does not need to be called directly by the user.

3.3.2 Shift-and-invert

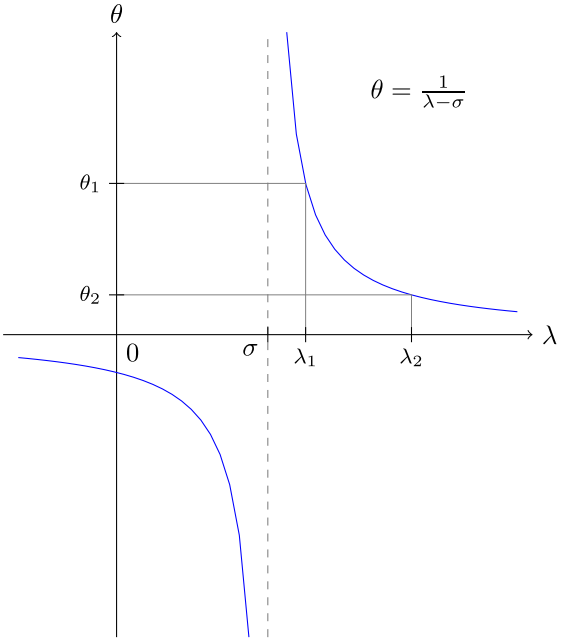


Fig. 1: The shift-and-invert spectral transformation

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

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

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

for standard and generalized problems, respectively. This transformation is effective for finding eigenvalues near σ since the eigenvalues θ of the operator that are largest in magnitude correspond to the eigenvalues λ of the original problem that are closest to the shift σ in absolute value, as illustrated in figure *The shift-and-invert spectral transformation* (page 38) for an example with real eigenvalues. Once the wanted eigenvalues have been found, they may be transformed back to eigenvalues of the original problem. Again, the eigenvectors remain unchanged. In this case, the relation between the eigenvalues of both problems is

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

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

This spectral transformation is used in the spectrum slicing technique, see section *Spectrum Slicing* (page 44).

[1] Note that the sign changed in SLEPc 3.5 with respect to previous versions.

3.3.3 Cayley

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

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

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

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

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

or in the command line with `-st_cayley_antishift`.

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

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

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

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

3.3.4 Preconditioner

As mentioned in the introduction of this chapter, the special type `STPRECOND` is used for handling preconditioners or preconditioned iterative linear solvers, which are used in the context of preconditioned eigensolvers for expanding the subspace. For instance, in the GD solver the so-called correction vector d_i to be added to the subspace in each iteration is computed as

$$d_i = K^{-1}P_i(A - \theta_i B)x_i, \tag{3.12}$$

where (θ_i, x_i) is the current approximation of the sought-after eigenpair, and P_i is a projector involving x_i and $K^{-1}x_i$. In the above expressions, K is a preconditioner matrix that is built from $A - \theta_i B$. However, since θ_i changes at each iteration, which would force recomputation of the preconditioner, we opt for using

$$K^{-1} \approx (A - \sigma B)^{-1}. \tag{3.13}$$

Similarly, in the JD eigensolver the expansion of the subspace is carried out by solving a correction equation similar to

$$(I - x_i x_i^*)(A - \theta_i B)(I - x_i x_i^*)d_i = -(A - \theta_i B)x_i, \tag{3.14}$$

where the system is solved approximately with a preconditioned iterative linear solver. For building the preconditioner of this linear system, the projectors $I - x_i x_i^*$ are ignored, and again it is not recomputed in each iteration. Therefore, the preconditioner is built as in equation (3.13) as well.

It should be clear from the previous discussion, that `STPRECOND` does not work in the same way as the rest of spectral transformations. In particular, it does not rely on `STBackTransform()`. It is rather a convenient mechanism for handling the preconditioner and linear solver (see examples in section *Solution of Linear Systems* (page 41)). The expressions shown in tables *Spectral transformations available in the ST package* (page 37) and *Operators used in each spectral transformation mode* (page 37) are just a reference to indicate from which matrix the preconditioner is built by default.

There is the possibility that the user overrides the default behavior, that is, to explicitly supply a matrix from which the preconditioner is to be built, with:

```
STSetPreconditionerMat(ST st,Mat mat);
```

The above function can also be used in other spectral transformations such as shift-and-invert in case the user has a cheap approximation K of the coefficient matrix $A - \sigma B$. An alternative is to pass approximations of both A and B so that ST builds the preconditioner matrix internally, with:

```
STSetSplitPreconditioner(ST st,PetscInt n,Mat Psplit[],MatStructure strp);
```

Note that preconditioned eigensolvers in EPS select STPRECOND by default, so the user does not need to specify it explicitly.

3.3.5 Polynomial Filter

The type STFILTER is also special. It is used in the case of standard symmetric (or Hermitian) eigenvalue problems when the eigenvalues of interest are interior to the spectrum and we want to avoid the high cost associated with the matrix factorization of the shift-and-invert spectral transformation. The techniques generically known as *polynomial filtering* aim at this goal.

The polynomial filtering methods address the eigenvalue problem

$$p(A)x = \theta x, \tag{3.15}$$

where $p(\cdot)$ is a suitable high-degree polynomial. Once the polynomial is built, the eigensolver relies on STApply() to compute approximations of the eigenvalues θ of the transformed problem. These approximations must be processed in some way in order to recover the λ eigenvalues. Note that in this case there is no STBackTransform() operation. Details of the method can be found in [Fang and Saad, 2012].

Currently, SLEPc provides several types of polynomial filtering techniques, which can be selected via STFilterSetType(). Note that the external solver EPSEVSL also implements polynomial filters to compute all eigenvalues in an interval.

When computing interior eigenvalues with STFILTER the following considerations must be taken into account:

- This technique is an option when the wanted eigenvalues are interior but shift-and-invert (or *Spectrum Slicing* (page 44)) is not affordable because computing a factorization is too expensive or cannot be done (e.g., with a shell matrix).
- The user should call EPSSetInterval() and select EPS_ALL for the wanted eigenvalues. The interval information is passed to ST, so there is no need to call STFilterSetInterval(). STFILTER will build the filter using this interval and the spectral range, the interval containing all the eigenvalues, set with STFilterSetRange(). It is generally recommended to also omit the call to the latter and let SLEPc compute an estimate internally, because if the provided spectral range is not accurate the filter may not work correctly.
- The degree of the polynomial can be set with STFilterSetDegree(). Convergence may be slow, it depends on the distribution of eigenvalues and the shape of the filter, which in turn depends on the polynomial degree and the two intervals mentioned above. In difficult problems, it may be necessary to increase the degree to 2000 or even more. There is no good way to estimate an appropriate value of the degree, but if the solver returns 0 eigenvalues in the first iteration then it may indicate that the filter is not sharp enough, so the degree should be increased.
- If the interval is very narrow, the degree of the polynomial should generally be large. It may be better to use a larger interval with more eigenvalues, even if only a few eigenvalues are needed. In any case, the solver should be configured with a sufficiently large subspace dimension (ncv in EPSSetDimensions()) to hold as many eigenvectors as eigenvalues contained in the interval, and an even larger value will generally benefit convergence.

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 systems of linear equations. The user must be aware of what is

going on in each case, so that it is possible to guide the solution process in the most beneficial way. This section describes several advanced aspects that can have a considerable impact on efficiency.

3.4.1 Solution of Linear Systems

In many of the cases shown in table *Operators used in each spectral transformation mode* (page 37), the operator contains an inverted matrix, which means that a system of linear 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. Suppose you run a program that solves a generalized eigenproblem, with default options:

```
$ ./program
```

In this case, the ST object associated with the EPS solver creates a KSP object whose coefficient matrix is B. By default, this KSP object is set to use a direct solver, in particular an LU factorization. However, default settings can be changed, as illustrated below.

The following command-line is equivalent to the previous one:

```
$ ./program -st_ksp_type preonly -st_pc_type lu
```

The two options specify the type of the linear solver and preconditioner to be used. The `-st_` prefix indicates that the option corresponds to the linear solver within ST. The combination `preonly+lu` instructs to use a direct solver (LU factorization, see PETSc documentation for details), so this is the same as the default. Adding a new option changes the default behavior, for instance

```
$ ./program -st_ksp_type preonly -st_pc_type lu -st_pc_factor_mat_solver_type mumps
```

In this case, an external linear solver package is used (MUMPS, see PETSc documentation for other available packages). Note that an external package is required for computing a matrix factorization in parallel, since PETSc itself only provides sequential direct linear solvers.

Instead of a direct linear solver, it is possible to use an iterative solver. This may be necessary in some cases, specially for very large problems. However, the user is warned that using an iterative linear solver makes the overall solution process less robust (see also the discussion of preconditioned eigensolvers below). As an example, the command-line

```
$ ./program -st_ksp_type gmres -st_pc_type bjacobi -st_ksp_rtol 1e-9
```

selects the GMRES solver with block Jacobi preconditioning. In the case of iterative solvers, it is important to use an appropriate tolerance, usually slightly more stringent for the linear solves relative to the desired accuracy of the eigenvalue calculation (10^{-9} in the example, compared to 10^{-8} for the eigensolver).

Although the direct solver approach may seem too costly, note that 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. For instance, for the shift-and-invert technique with $\tau = 10$ using BiCGStab+Jacobi:

```
$ ./program -st_type sinvert -eps_target 10 -st_ksp_type bcgs -st_pc_type jacobi
```

In shift-and-invert and Cayley, unless $\sigma = 0$, the coefficient matrix is not a simple matrix but an expression that can be explicitly constructed or not, depending on the user's choice. This issue is examined in detail in section *Explicit Computation of the Coefficient Matrix* (page 42) below.

> **Note**
>
> By default the preconditioner is built from the matrix $A - \sigma I$ (or $A - \sigma B$ in generalized problems). In some special situations, the user may want to build the preconditioner from a different matrix, e.g., stemming from simplified versions of A and B. To do this, one should use the functions mentioned in section *Preconditioner* (page 39).

In many cases, especially if a shift-and-invert or Cayley transformation is being used, iterative methods may not be well suited for solving linear systems (because of the properties of the coefficient matrix that can be indefinite and ill-conditioned). When using an iterative linear solver, it may be helpful to run with the option `-st_ksp_converged_reason`, which will display the number of iterations required in each operator application. In extreme cases, the iterative solver fails, so `EPSSolve()` aborts with an error

```
[0]PETSC ERROR: KSP did not converge (reason=DIVERGED_ITS)!
```

If this happens, it is necessary to use a direct method for solving the linear systems, as explained above.

The Case of Preconditioned Eigensolvers

The KSP object contained internally in ST is also used for applying the preconditioner or solving the correction equation in preconditioned eigensolvers.

The GD eigensolver employs just a preconditioner. Therefore, by default it sets the KSP type to `preonly` (no other KSP is allowed) and the PC type to `jacobi`. The user may change the preconditioner, for example as

```
$ ./ex5 -eps_type gd -st_pc_type asm
```

The JD eigensolver uses both an iterative linear solver and a preconditioner, so both KSP and PC are meaningful in this case. It is important to note that, contrary to the ordinary spectral transformations where a direct linear solver is recommended, in JD using an iterative linear solver is usually better than a direct solver. Indeed, the best performance may be achieved with a few iterations of the linear solver (or a large tolerance). For instance, the next example uses JD with GMRES+Jacobi limiting to 10 the number of allowed iterations for the linear solver:

```
$ ./ex5 -eps_type jd -st_ksp_type gmres -st_pc_type jacobi -st_ksp_max_it 10
```

A discussion on the different options available for the Davidson solvers can be found in [Romero and Roman, 2014].

3.4.2 Explicit Computation of the Coefficient Matrix

Three possibilities can be distinguished regarding the form of the coefficient matrix of the systems of linear equations associated with 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 -eps_target 10 -st_ksp_type cg -st_pc_type jacobi -st_matmode shell
```

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

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

The user must consider which approach is the most appropriate for the particular application. The different options have advantages and drawbacks. The "shell" 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() might be 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.

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

3.4.3 Preserving the Symmetry in Generalized Eigenproblems

As mentioned in section *Defining the Problem* (page 16), some eigensolvers can exploit symmetry and compute a solution for Hermitian problems with less storage and/or computational cost than other methods. Also, symmetric solvers can be more accurate in some cases. However, in the case of generalized eigenvalue problems in which both A and B are symmetric, it happens that, due to the spectral transformation, symmetry is lost since none of the transformed operators $B^{-1}A - \sigma I$, $(A - \sigma B)^{-1}B$, etc. is symmetric (the same applies in the Hermitian case for complex matrices).

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

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

The second one can be used for preserving the symmetry in symmetric definite generalized problems, as described below. Note that $\langle x, y\rangle_B$ is a genuine inner product only if B is symmetric positive definite (for the case of symmetric positive semi-definite B see section *Purification of Eigenvectors* (page 44)).

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

Internally, SLEPc operates with the scheme illustrated in figure *Abstraction used by SLEPc solvers* (page 44). The operations indicated by dashed arrows are implemented as virtual functions. 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

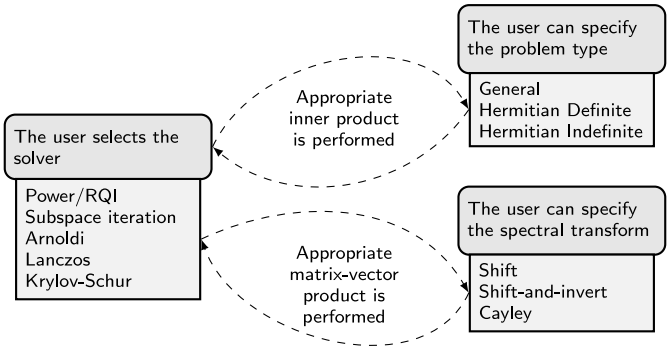


Fig. 2: Abstraction used by SLEPc solvers

with `EPSSetProblemType()` (see section *Defining the Problem* (page 16)). In the case of the Cayley transform, SLEPc is using $\langle x,y\rangle_{A+\nu B}$ as the inner product for preserving symmetry.

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

Note that the above discussion is not directly applicable to `STPRECOND` and the preconditioned eigensolvers, in the sense that the goal is not to recover the symmetry of the operator. Still, the B-inner product is also used in generalized symmetric-definite problems.

3.4.4 Purification of Eigenvectors

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

In shift-and-invert with operator matrix $T=(A-\sigma B)^{-1}B$, when B is singular all the eigenvectors that belong to finite eigenvalues are also eigenvectors of T and belong to the range of T, $\mathcal{R}(T)$. In this case, the bilinear form $\langle x,y\rangle_B$ introduced in section *Preserving the Symmetry in Generalized Eigenproblems* (page 43) is a semi-inner product and $\|x\|_B=\sqrt{\langle x,x\rangle_B}$ is a semi-norm. As before, T is self-adjoint with respect to this inner product since $BT=T^*B$. Also, $\langle x,y\rangle_B$ is a true inner product on $\mathcal{R}(T)$.

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

Although more elaborate purification strategies have been proposed (usually trying to reduce the computational effort, see [Meerbergen and Spence, 1997, Nour-Omid *et al.*, 1987]), the approach in SLEPc is simply to explicitly force the initial vector in the range of T, with $v_0 \leftarrow Tv_0$, as well as the computed eigenvectors at the end, $x_i \leftarrow Tx_i$. Since this computation can be costly, it can be deactivated if the user knows that B is non-singular, with:

```
EPSSetPurify(EPS eps,PetscBool purify);
```

A final comment is that eigenvector corruption happens also in the non-symmetric case. If A is non-symmetric but B is symmetric positive semi-definite, then the scheme presented above (B-inner product together with purification) can still be applied and is generally more successful than the straightforward approach with the standard inner product. For using this scheme in SLEPc, the user has to specify the special problem type `EPS_PGNHEP`, see table *Problem types considered in EPS* (page 17).

3.4.5 Spectrum Slicing

In the context of symmetric-definite generalized eigenvalue problems (`EPS_GHEP`) it is often required to compute all eigenvalues contained in a given interval $[a,b]$. This poses some difficulties, such as:

- The number of eigenvalues in the interval is not known a priori.

- There might be many eigenvalues, in some applications a significant percentage of the spectrum (20%, say).
- We must make certain that no eigenvalues are missed, and in particular all eigenvalues must be computed with their correct multiplicity.
- In some applications, the interval is open in one end, i.e., either a or b can be infinite.

One possible strategy to solve this problem is to sweep the interval from one end to the other, computing chunks of eigenvalues with a spectral transformation that updates the shift dynamically. This is generally referred to as *spectrum slicing*. The method implemented in SLEPc is similar to that proposed by Grimes *et al.* [1994], where inertia information is used to validate sub-intervals. Given a symmetric-indefinite triangular factorization

$$A - \sigma B = LDL^T, \tag{3.16}$$

by Sylvester's law of inertia we know that the number of eigenvalues on the left of σ is equal to the number of negative eigenvalues of D,

$$\nu(A - \sigma B) = \nu(D). \tag{3.17}$$

A detailed description of the method available in SLEPc can be found in [Campos and Roman, 2012]. The SLEPc interface hides all the complications of the algorithm. However, the user must be aware of all the restrictions for this technique to be employed:

- This is currently implemented only in Krylov-Schur.
- The method is based on shift-and-invert, so `STSINVERT` must be used. Furthermore, direct linear solvers are required.
- The direct linear solver must provide the matrix inertia (see PETSc's MatGetInertia()).

An example command-line that sets up all the required options is:

```
$ ./ex2 -n 50 -eps_interval 0.4,0.8 -st_type sinvert -st_ksp_type preonly -st_pc_type cholesky
```

Note that PETSc's Cholesky factorization is not parallel, so for doing spectrum slicing in parallel it is required to use an external solver that supports inertia. For example, with MUMPS (see section *Solution of Linear Systems* (page 41) on how to use external linear solvers) we would do:

```
$ ./ex2 -n 50 -eps_interval 0.4,0.8 -st_type sinvert -st_ksp_type preonly -st_pc_type cholesky -st_pc_
↪factor_mat_solver_type mumps -st_mat_mumps_icntl_13 1
```

The last option is required by MUMPS to compute the inertia. An alternative is to use SuperLU_DIST, in which case the required options would be:

```
$ ./ex2 -n 50 -eps_interval 0.4,0.8 -st_type sinvert -st_ksp_type preonly -st_pc_type cholesky -st_pc_
↪factor_mat_solver_type superlu_dist -st_mat_superlu_dist_rowperm NOROWPERM
```

In the latter example, MatSetOption() must be used in both matrices to explicitly state that they are symmetric (or Hermitian in the complex case).

Apart from the above recommendations, the following must be taken into account:

- The parameters `nev` and `ncv` from `EPSSetDimensions()` are determined internally (user-provided values are ignored, and set to the number of eigenvalues in the interval). It is possible for the user to specify the "local" `nev` and `ncv` parameters that will be used when computing eigenvalues around each shift, with `EPSKrylovSchurSetDimensions()`.
- The user can also tune the computation by setting the value of `max_it`.

Note

Usage with Complex Scalars: Some external packages that provide inertia information (MUMPS, Pardiso) do so only in real scalars, but not in the case of complex scalars. Hence, with complex scalars spectrum slicing is available only sequentially (with PETSc's Cholesky factorization) or via SuperLU_DIST (as in the last example

above). An alternative to spectrum slicing is to use the CISS solver with a region enclosing an interval on the real axis, see [Maeda *et al.*, 2016] for details.

Use of Multiple Communicators

Since spectrum slicing requires direct linear solves, parallel runs may suffer from bad scalability in the sense that increasing the number of MPI processes does not imply a performance gain. For this reason, SLEPc provides the option of using multiple communicators, that is, splitting the initial MPI communicator in several groups, each of them in charge of processing part of the interval.

The multi-communicator setting is activated with a value of `npart`>1 in the following function:

```
EPSKrylovSchurSetPartitions(EPS eps,PetscInt npart);
```

The interval $[a, b]$ is then divided in `npart` subintervals of equal size, and the problem of computing all eigenvalues in $[a, b]$ is divided in `npart` independent subproblems. Each subproblem is solved using only a subset of the initial p processes, with $\lceil p/\texttt{npart} \rceil$ processes at most. A final step will gather all computed solutions so that they are available in the whole `EPS` communicator.

The division of the interval in subintervals is done blindly, and this may result in load imbalance if some subintervals contain much more eigenvalues than others. This can be prevented by passing a list of subinterval boundaries, provided that the user has a priori information to roughly determine the eigenvalue distribution:

```
EPSKrylovSchurSetSubintervals(EPS eps,PetscReal subint[]);
```

An additional benefit of multi-communicator support is that it enables parallel spectrum slicing runs without the need to install a parallel direct solver (MUMPS), by setting the number of partitions equal to the number of MPI processes. The following command-line example uses sequential linear solves in 4 partitions, one process each:

```
$ mpiexec -n 4 ./ex25 -eps_interval 0.4,0.8 -eps_krylovschur_partitions 4 -st_type sinvert -st_ksp_type␣
↪preonly -st_pc_type cholesky
```

The analog example using MUMPS with 5 processes in each partition:

```
$ mpiexec -n 20 ./ex25 -eps_interval 0.4,0.8 -eps_krylovschur_partitions 4 -st_type sinvert -st_ksp_type␣
↪preonly -st_pc_type cholesky -st_pc_factor_mat_solver_type mumps -st_mat_mumps_icntl_13 1
```

3.4.6 Spectrum Folding

In SLEPc versions prior to 3.5, `ST` had another type intended to perform the spectrum folding technique described below. It is no longer available with `ST`, but it can be implemented directly in application code as illustrated in example `ex24.c`.

Spectrum folding involves squaring in addition to shifting. This makes sense for standard Hermitian eigenvalue problems, where the transformed problem to be addressed is

$$(A - \sigma I)^2 x = \theta x. \tag{3.18}$$

The following relation holds

$$\theta = (\lambda - \sigma)^2. \tag{3.19}$$

Note that the mapping between λ and θ is not injective, and hence this cannot be considered a true spectral transformation.

The effect is that the spectrum is folded around the value of σ. Thus, eigenvalues that are closest to the shift become the smallest eigenvalues in the folded spectrum, see figure *Illustration of the effect of spectrum folding* (page 47). For this reason, spectrum folding is commonly used in combination with eigensolvers that compute the smallest eigenvalues, for instance in the context of electronic structure calculations [Canning *et al.*, 2000]. This transformation can be an effective, low-cost alternative to shift-and-invert.

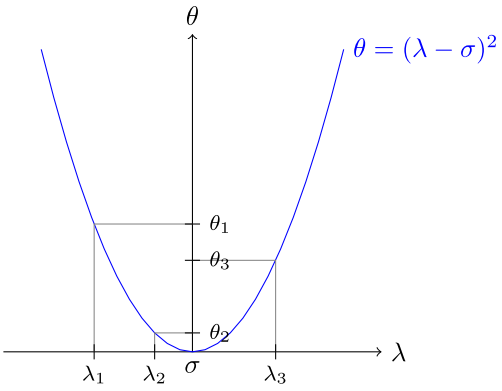


Fig. 3: Illustration of the effect of spectrum folding

CHAPTER 4

SVD: Singular Value Decomposition

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

4.1 Mathematical Background

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

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

4.1.1 The (Standard) Singular Value Decomposition (SVD)

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

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

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

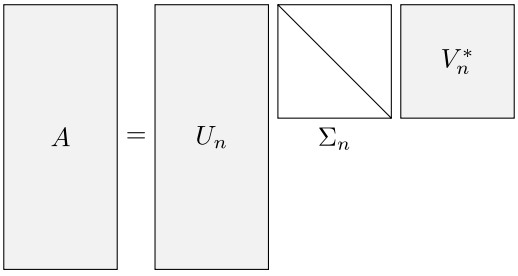


Fig. 1: Scheme of the thin SVD of a rectangular matrix

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

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

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

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

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

The last left singular vectors corresponding to equation (4.4) are often not computed, especially if $m \gg n$. In that case, the resulting factorization is sometimes called the *thin* SVD, $A = U_n\Sigma_n V_n^*$, and is depicted in figure *Scheme of the thin SVD of a rectangular matrix* (page 49). This factorization can also be written as

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

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

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

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

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

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

Equivalent Eigenvalue Problems

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

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

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

By pre-multiplying equation (4.2) by A^* and then using equation (4.3), the following relation results

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

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

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

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

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

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

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

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

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

4.1.2 The Generalized Singular Value Decomposition (GSVD)

An extension of the SVD to the case of two matrices is the generalized singular value decomposition (GSVD), which can be applied in constrained least squares problems, among others. An overview of the problem can be found in [Golub and van Loan, 1996, 8.7.3].

Consider two matrices, $A \in \mathbb{C}^{m\times n}$ with $m \geq n$ and $B \in \mathbb{C}^{p\times n}$. Note that both matrices must have the same column dimension. Then there exist two unitary matrices $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{p\times p}$ and an invertible matrix $X \in \mathbb{C}^{n\times n}$ such that

$$U^*AX = C, \qquad V^*BX = S, \tag{4.9}$$

where $C = \mathrm{diag}(c_1, \ldots, c_n)$ and $S = \mathrm{diag}(s_{n-q+1}, \ldots, s_n)$ with $q = \min(p, n)$. The values c_i and s_i are real and nonnegative, and the ratios define the generalized singular values,

$$\sigma(A, B) \equiv \{c_1/s_1, \ldots, c_q/s_q\}, \tag{4.10}$$

and if $p < n$ we can consider that the first $n - p$ generalized singular values are infinite, as if $s_1 = \cdots = s_{n-p} = 0$. Note that if B is the identity matrix, X can be taken to be unitary and then we recover the standard SVD, $\sigma(A, I) = \sigma(A)$, that is why equation (4.9) is considered a generalization of the SVD.

The diagonal matrices C and S satisfy $C^*C + S^*S = I$, and are related to the CS decomposition [Golub and van Loan, 1996, 2.6.4] associated with the orthogonal factor of the QR factorization of matrices A and B stacked, that is, if

$$Z := \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \tag{4.11}$$

then C and S can be obtained from the singular values of Q_1 and Q_2, respectively. The matrix Z is relevant for algorithms and is often built explicitly.

The above description assumes that matrix Z has full column rank, and the rank of B is also q. In the general case where these assumptions do not hold, the structure of matrices C and S is a bit more complicated. This includes also the case where $m < n$. A detailed description of those cases can be found in [Anderson *et al.*, 1999, 2.3.5.3].

As in the case of the SVD, one can consider thin, compact, truncated, and partial versions of the GSVD. A pictorial representation of the thin GSVD is shown in figure *Scheme of the thin GSVD of two matrices A and B, for the case p < n* (page 52).

The columns of X, x_i, are called the (right) generalized singular vectors. The left vectors in this case would correspond to the columns of U and V. In SLEPc, the user interface will provide these vectors stacked on top of each other, as a single $(m + p)$-vector $\begin{bmatrix} u_i \\ v_i \end{bmatrix}$.

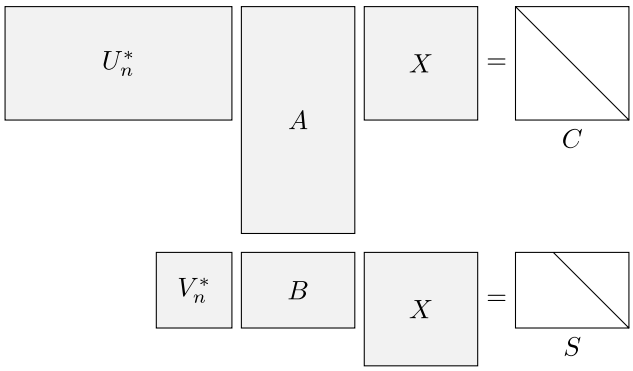


Fig. 2: Scheme of the thin GSVD of two matrices A and B, for the case $p < n$

Equivalent Eigenvalue Problems

In the GSVD it is also possible to formulate the problem as an eigenvalue problem, which opens the door to approach its solution via EPS. The columns of X satisfy

$$s_i^2 A^* A x_i = c_i^2 B^* B x_i, \tag{4.12}$$

and so if $s_i \neq 0$ then $A^* A x_i = \sigma_i^2 B^* B x_i$, a generalized eigenvalue problem for the matrix pencil (A^*A, B^*B). This is the analog of the cross product matrix eigenproblem of equation (4.6).

The formulation that is analog to the eigenproblem associated with the cyclic matrix equation (4.8) is to solve the generalized eigenvalue problem defined by any of the matrix pairs

$$\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}, \begin{bmatrix} I & 0 \\ 0 & B^*B \end{bmatrix} \right), \qquad \text{or} \qquad \left(\begin{bmatrix} 0 & B \\ B^* & 0 \end{bmatrix}, \begin{bmatrix} I & 0 \\ 0 & A^*A \end{bmatrix} \right). \tag{4.13}$$

4.1.3 The Hyperbolic Singular Value Decomposition (HSVD)

The hyperbolic singular value decomposition (HSVD) was introduced in [Onn *et al.*, 1991], motivated by some signal processing applications such as the so-called covariance differencing problem. The formulation of the HSVD is similar to that of the SVD, except that U is orthogonal with respect to a signature matrix,

$$A = U \Sigma V^*, \qquad U^* \Omega U = \tilde{\Omega}, \tag{4.14}$$

where $\Omega = \mathrm{diag}(\pm 1)$ is an $m \times m$ signature matrix provided by the user, while $\tilde{\Omega}$ is another signature matrix obtained as part of the solution. Sometimes U is said to be a hyperexchange matrix, or also an $(\Omega, \tilde{\Omega})$-orthogonal matrix. Note that in the problem definition normally found in the literature it is V that is $(\Omega, \tilde{\Omega})$-orthogonal and not U. We choose this definition for consistency with respect to the generalized HSVD of two matrices. If the user wants to compute the HSVD according to the alternative definition, then it suffices to (conjugate) transpose the input matrix A, using for instance MatHermitianTranspose().

As in the case of the SVD, the solution of the problem consists in singular triplets (σ_i, u_i, v_i), with σ_i real and nonnegative and sorted in nonincreasing order. Note that these quantities are different from those of section *The (Standard) Singular Value Decomposition (SVD)* (page 49), even though we use the same notation here. With each singular triplet, there is an associated sign $\tilde{\omega}_i$ (either 1 or -1), the corresponding diagonal element of $\tilde{\Omega}$. In SLEPc, this value is not returned by the user interface, but if required it can be easily computed as $u_i^* \Omega u_i$.

The relations between left and right singular vectors are slightly different from those of the standard SVD. We have $AV = U\Sigma$ and $A^*\Omega U = V\Sigma^*\tilde{\Omega}$, so for $m \geq n$ the following relations hold, together with equation (4.2):

$$A^* \Omega u_i = v_i \sigma_i \tilde{\omega}_i \,, \quad i = 1, \ldots, n, \tag{4.15}$$

$$A^* \Omega u_i = 0 \,, \quad i = n+1, \ldots, m. \tag{4.16}$$

In SLEPc we will compute a partial HSVD consisting of either the largest or smallest hyperbolic singular triplets. Note that the sign $\tilde{\omega}_i$ is not used when sorting for largest or smallest σ_i.

Equivalent Eigenvalue Problems

Once again, we can derive cross and cyclic schemes to compute the decomposition by solving an eigenvalue problem. The cross product matrix approach has two forms, to be selected depending on whether $m \geq n$ or not, as discussed in section *The (Standard) Singular Value Decomposition (SVD)* (page 49). The first form,

$$A^*\Omega A v_i = \sigma_i^2 \tilde{\omega}_i v_i, \tag{4.17}$$

is derived by pre-multiplying equation (4.2) by $A^*\Omega$ and then using equation (4.15). This eigenproblem can be solved as a HEP (cf. section *Defining the Problem* (page 16)) and may have both positive and negative eigenvalues, corresponding to $\tilde{\omega}_i = 1$ and $\tilde{\omega}_i = -1$, respectively. Once the right vector v_i has been computed, the corresponding left vector can be obtained using equation (4.2) with just a matrix-vector product, $u_i = \sigma_i^{-1} A v_i$.

The second form of cross computes the left vectors first, by pre-multiplying equation (4.15) by A and then using equation (4.2),

$$AA^*\Omega u_i = \sigma_i^2 \tilde{\omega}_i u_i. \tag{4.18}$$

In this case, the right singular vectors are obtained as $v_i = (\sigma_i \tilde{\omega}_i)^{-1} A^*\Omega u_i$. The coefficient matrix of (4.18) is non-Hermitian, so the eigenproblem has to be solved as non-Hermitian, or alternatively it can be formulated as a generalized eigenvalue problem of type `EPS_GHIEP` (cf. section *Defining the Problem* (page 16)) for the indefinite pencil (AA^*, Ω),

$$AA^*\hat{u}_i = \sigma_i^2 \tilde{\omega}_i \Omega \hat{u}_i, \tag{4.19}$$

with $\hat{u}_i = \Omega u_i$. The eigenvectors obtained from equation (4.18) or equation (4.19) must be normalized so that $U^*\Omega U = \tilde{\Omega}$ holds.

Finally, in the cyclic matrix approach for the HSVD we must solve a generalized eigenvalue problem defined by the matrices of order $(m+n)$

$$H(A) = \begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}, \qquad \hat{\Omega} = \begin{bmatrix} \Omega & 0 \\ 0 & I \end{bmatrix}. \tag{4.20}$$

As in the case of equation (4.19), this problem is Hermitian-indefinite and hence it may have complex eigenvalues. However, it can be shown that nonzero eigenvalues of the matrix pair $(H(A), \hat{\Omega})$ are either real (equal to $\pm\sigma_i$ for a certain hyperbolic singular value σ_i) or purely imaginary (equal to $\pm\sigma_i j$ for a certain σ_i with $j = \sqrt{-1}$). The associated eigenvectors are $\left[\begin{smallmatrix} \varsigma_i u_i \\ v_i \end{smallmatrix}\right]$, where ς_i is either ± 1 or $\pm j$.

4.2 Basic Usage

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

Listing 1: Example code for basic solution with `SVD`.

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

SVDCreate(PETSC_COMM_WORLD, &svd);
SVDSetOperators(svd, A, NULL);
SVDSetProblemType(svd, SVD_STANDARD);
SVDSetFromOptions(svd);
SVDSolve(svd);
SVDGetConverged(svd, &nconv);
for (j=0;j<nconv;j++) {
  SVDGetSingularTriplet(svd, j, &sigma, u, v);
  SVDComputeError(svd, j, SVD_ERROR_RELATIVE, &error);
}
SVDDestroy(&svd);
```

The basic steps for computing a partial SVD with SLEPc are illustrated in listing *Example code for basic solution with SVD.* (page 53). The steps are more or less the same as those described in chapter *EPS: Eigenvalue Problem Solver* (page 13) for the eigenvalue problem. First, the solver context is created with `SVDCreate()`. Then the problem matrices have to be specified with `SVDSetOperators()` and the type of problem can be selected via `SVDSetProblemType()`. Then, a call to `SVDSolve()` invokes the actual solver. After that, `SVDGetConverged()` is used to determine how many solutions have been computed, which are retrieved with `SVDGetSingularTriplet()`. Finally, `SVDDestroy()` gets rid of the object.

If one compares this example code with the `EPS` example in listing *Example code for basic solution with EPS* (page 15), the most outstanding differences are the following:

- The singular value is a PetscReal, not a PetscScalar.
- Each singular vector is defined with a single Vec object, not two as was the case for eigenvectors.

4.3 Defining the Problem

Defining the problem consists in specifying the problem matrix A (and also a second matrix B in case of the GSVD), the problem type, and the portion of the spectrum to be computed.

The problem matrices are provided with the following function:

```
SVDSetOperators(SVD svd,Mat A,Mat B);
```

where `A` can be any matrix, not necessarily square, stored in any allowed PETSc format including the matrix-free mechanism (see section *Supported Matrix Types* (page 96) for a detailed discussion), and `B` is generally set to `NULL` except in the case of computing the GSVD. If the problem is hyperbolic (cf. *The Hyperbolic Singular Value Decomposition (HSVD)* (page 52)) then in addition a signature matrix Ω must be provided with:

```
SVDSetSignature(SVD svd,Vec omega);
```

> **Note**
>
> The signature matrix Ω is represented as a vector containing values equal to 1 or -1.

The problem type can be specified with:

```
SVDSetProblemType(SVD svd,SVDProblemType type);
```

Note that in `SVD` it is currently not strictly required to call this function, since the problem type can be deduced from the information passed by the user: if only one matrix is passed to `SVDSetOperators()` the problem type will default to a standard SVD (or hyperbolic SVD if a signature has been provided), while if two matrices are passed then it defaults to GSVD. Still, it is recommended to always call `SVDSetProblemType()`. Table *Problem types considered in SVD* (page 54) lists the supported problem types.

Table 1: Problem types considered in `SVD`

Problem Type	`SVDProblemType`	Command line key
Standard SVD	`SVD_STANDARD`	`-svd_standard`
Generalized SVD (GSVD)	`SVD_GENERALIZED`	`-svd_generalized`
Hyperbolic SVD (HSVD)	`SVD_HYPERBOLIC`	`-svd_hyperbolic`

It is important to note that all SVD solvers in SLEPc make use of both A and A^*, as suggested by the description in section *The (Standard) Singular Value Decomposition (SVD)* (page 49). A^* is not explicitly passed as an argument to `SVDSetOperators()`, therefore it will have to stem from A. There are two possibilities for this: either A is transposed explicitly and A^* is created as a distinct matrix, or A^* is handled implicitly via MatMultTranspose() (or MatMultHermitianTranspose() in the complex case) operations whenever a matrix-vector product is required in the algorithm. The default is to build A^* explicitly, but this behavior can be changed with:

```
SVDSetImplicitTranspose(SVD svd,PetscBool impl);
```

In section *The (Standard) Singular Value Decomposition (SVD)* (page 49), it was mentioned that in SLEPc the cross product approach chooses the smallest of the two possible cases A^*A or AA^*, that is, A^*A is used if A is a tall, thin matrix ($m \geq n$), and AA^* is used if A is a fat, short matrix ($m < n$). In fact, what SLEPc does internally is that if $m < n$ the roles of A and A^* are reversed. This is equivalent to transposing all the SVD factorization, so left singular vectors become right singular vectors and vice versa. This is actually done in all singular value solvers, not only the cross product approach. The objective is to simplify the number of cases to be treated internally by SLEPc, as well as to reduce the computational cost in some situations. Note that this is done transparently and the user need not worry about transposing the matrix, only to indicate how the transpose has to be handled, as explained above.

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

```
SVDSetDimensions(SVD svd,PetscInt nsv,PetscInt ncv,PetscInt mpd);
```

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

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

requests 10 singular values and instructs to use 24 column vectors. Note that `ncv` must be at least equal to `nsv`, although in general it is recommended (depending on the method) to work with a larger subspace, for instance `ncv` $\geq 2 \cdot$ `nsv` or even more. As in the case of the `EPS` object, the last argument, `mpd`, can be used to limit the maximum dimension of the projected problem, as discussed in section *Computing a Large Portion of the Spectrum* (page 28). Using this parameter is especially important in the case that a large number of singular values are requested.

Table 2: Available possibilities for selection of the singular values of interest

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

For the selection of the portion of the spectrum of interest, there are only two possibilities in the case of SVD: largest and smallest singular values, see table *Available possibilities for selection of the singular values of interest* (page 55). The default is to compute the largest ones, but this can be changed with:

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

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

4.3.1 Using a threshold to specify wanted singular values

In some applications, the number of wanted singular values is not known a priori. For instance, suppose that matrix A is known to have low rank, and we want to approximate it by a truncated SVD containing the leading singular values. The goal is to have a good approximation, that is, discard only small singular values. The numerical rank k might be difficult to estimate, due to discretization error or other reasons. But since we assume that singular values decay abruptly around some unknown value k, we can configure SLEPc to detect this decay.

The threshold δ can be set with:

```
SVDSetThreshold(SVD svd,PetscReal thres,PetscBool rel);
```

When `rel` is PETSC_TRUE, the solver will compute k singular values, with $\sigma_j \geq \delta \cdot \sigma_1$, for $j = 1, \ldots, k-1$, where k is computed internally. In this case, the threshold is relative to the largest singular value σ_1, i.e., the matrix norm. It can be interpreted as a percentage, for instance $\delta = 0.8$ means compute singular values that are at least 80% of the matrix norm. Alternatively, an absolute threshold can be used by setting `rel` to PETSC_FALSE.

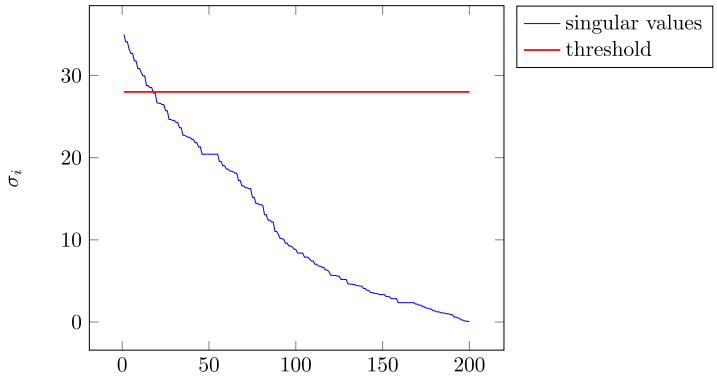


Fig. 3: Illustration of threshold usage with the singular values of the `rdb200` matrix; the red line represents a relative threshold of $\delta = 0.8$

In the low-rank example suggested above, the singular values will decay fast for a relatively small k, so a relative threshold $\delta = 0.5$ or less could do the job. But care must be taken in matrices whose singular values decay progressively, as in the example of figure *Illustration of threshold usage with the singular values of the `rdb200` matrix* (page 56), since a small δ would imply computing many singular triplets and hence a very high cost, both computationally and in memory. Since the number of computed singular values is not known a priori, the solver will need to reallocate the basis of vectors internally, to have enough room to accommodate all the singular vectors, so this option must be used with caution to avoid out-of-memory problems. The recommendation is to set the value of `ncv` to be larger than the estimated number of singular values, to minimize the number of reallocations. You can also use the `nsv` parameter in combination with the threshold to stop in case the number of computed singular triplets exceeds that value.

An absolute threshold is also available when computing smallest singular values, in which case the solver will compute the k smallest singular values, where $\sigma_j < \delta$, $j = n-k+1, \ldots, n$.

4.4 Selecting the SVD Solver

The available methods for computing the partial SVD are shown in table *List of solvers available in the SVD module* (page 57). These methods can be classified in the following categories:

- Solvers based on `EPS`. These solvers set up an `EPS` object internally, thus using the available eigensolvers for solving the SVD problem. The two possible approaches in this case are the cross product matrix and the cyclic matrix, as described in section *Mathematical Background* (page 49).
- Specific SVD solvers. These are typically eigensolvers that have been adapted algorithmically to exploit the structure of the SVD or GSVD problems. There are two Lanczos-type solvers in this category: Lanczos and thick-restart Lanczos, see [Hernandez *et al.*, 2007] for a detailed description of these methods. In this category, we could also add the randomized SVD (RSVD), a special solver that does not compute individual singular vectors accurately, but rather a low-rank approximation of A by means of randomization techniques.
- The LAPACK solver. This is an interface to some LAPACK routines, analog of those in the case of eigenproblems. These routines operate in dense mode with only one processor and therefore are suitable only for moderate size problems. This solver should be used only for debugging purposes.
- External packages: ScaLAPACK, KSVD and ELEMENTAL are dense packages and compute the complete SVD, while PRIMME offers Davidson-type methods to compute only a few singular triplets.

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

Table 3: List of solvers available in the SVD module; in the column of supported problems, 'G' means GSVD and 'H' means HSVD (the standard SVD is supported by all solvers)

Method	SVDType	Options Database	Supported	Default
Cross Product	SVDCROSS	cross	G,H	*
Cyclic Matrix	SVDCYCLIC	cyclic	G,H	
Lanczos	SVDLANCZOS	lanczos		
Thick-restart Lanczos	SVDTRLANCZOS	trlanczos	G,H	
Randomized SVD (RSVD)	SVDRANDOMIZED	randomized		
Wrapper to LAPACK	SVDLAPACK	lapack	G	
Wrapper to ScaLAPACK	SVDSCALAPACK	scalapack		
Wrapper to KSVD	SVDKSVD	ksvd		
Wrapper to ELEMENTAL	SVDELEMENTAL	elemental		
Wrapper to PRIMME	SVDPRIMME	primme		

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

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

while the user writes the options database command `-svd_type` followed by the name of the method (see table *List of solvers available in the SVD module* (page 57)).

Note

Not all solvers in the SVD module support non-standard problems (GSVD and HSVD). *List of solvers available in the SVD module* (page 57) indicates which solvers can be used for these.

The EPS-based solvers deserve some additional comments. These SVD solvers work by creating an EPS object internally and setting up an eigenproblem of type EPS_HEP (or EPS_GHEP in the case of the GSVD, or EPS_GHIEP in the case of the HSVD). These solvers implement the cross product matrix and the cyclic matrix approaches as described in section *Mathematical Background* (page 49). Therefore, the operator matrix associated with the EPS object will be A^*A in the case of the `cross` solver and $H(A)$ in the case of the `cyclic` solver (with variations in the case of non-standard problems).

In the case of the `cross` solver, the matrix A^*A is not built explicitly by default, since the resulting matrix may be much less sparse than the original matrix. By default, a shell matrix is created internally in the SVD object and passed to the EPS object. Still, the user may choose to force the computation of A^*A explicitly, by means of PETSc's sparse matrix-matrix product subroutines. This is set with:

```
SVDCrossSetExplicitMatrix(SVD svd,PetscBool explicit);
```

In the `cyclic` solver the user can also choose between handling $H(A)$ implicitly as a shell matrix (the default), or forming it explicitly, that is, storing its elements in a distinct matrix. The function for setting this option is:

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

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

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

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

```
$ ./program -svd_type cross -svd_cross_eps_type gd
```

At this point, one may consider changing also the options of the ST object associated with the EPS object in `cross` and `cyclic` SVD solvers, for example to compute singular values located at the interior of the spectrum via a shift-and-invert transformation. This is indeed possible, but some considerations have to be taken into account. When A^*A or $H(A)$ are managed as shell matrices, then the potential of the spectral transformation is limited seriously, because some of the required operations will not be defined if working with implicit matrices (this is discussed briefly in sections *Supported Matrix Types* (page 96) and *Explicit Computation of the Coefficient Matrix* (page 42)). The following example illustrates the computation of interior singular values with the `cyclic` solver with explicit $H(A)$ matrix:

```
$ ./program -svd_type cyclic -svd_cyclic_explicitmatrix -svd_cyclic_st_type sinvert -svd_cyclic_eps_target␣
↪12.0 -svd_cyclic_st_ksp_type preonly -svd_cyclic_st_pc_type lu
```

Similarly, in the case of GSVD the thick-restart Lanczos solver uses a KSP solver internally, that can be configured by accessing it with:

```
SVDTRLanczosGetKSP(SVD svd,KSP *ksp);
```

or with the corresponding command-line options prefixed with `-svd_trlanczos_`. This KSP object is used to solve a linear least squares problem at each Lanczos step with coefficient matrix Z equation (4.11), which by default is a shell matrix but the user can choose to create it explicitly with the function `SVDTRLanczosSetExplicitMatrix()`.

4.5 Retrieving the Solution

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

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

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

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

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

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

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

In some applications, it may be enough to compute only the right singular vectors. This is especially important in cases in which memory requirements are critical (remember that both U_k and V_k are dense matrices, and U_k may require much more storage than V_k, see figure *Scheme of the thin SVD of a rectangular matrix* (page 49)). In SLEPc, there is no general option for specifying this, but the default behavior of some solvers is to compute only right vectors and allocate/compute left vectors only in the case that the user requests them. This is done in the `cross` solver and in some special variants of other solvers such as one-sided Lanczos (consult [Hernandez *et al.*, 2007] for specific solver options).

In the case of the GSVD, the `sigma` argument of `SVDGetSingularTriplet()` contains $\sigma_i = c_i/s_i$ and the second Vec argument (`v`) contains the right singular vectors (x_i), while the first Vec argument (`u`) contains the other vectors of the decomposition stacked on top of each other, as a single $(m+p)$-vector: $\left[\begin{smallmatrix} u_i \\ v_i \end{smallmatrix}\right]$.

[1] The exception is in the HSVD, where u_j is normalized so that $U^*\Omega U = \tilde{\Omega}$.

Warning

In the GSVD, one has to be careful when dealing with the $(m+p)$-vector `u` in parallel runs, since the entries of the upper and lower part will get mixed unless we manage it as a VECNEST. One should follow a procedure similar to the one discussed in section *Extracting Eigenvectors of Structured Eigenproblems* (page 32).

4.5.1 Reliability of the Computed Solution

In SVD computations, a-posteriori error bounds are much the same as in the case of Hermitian eigenproblems, due to the equivalence discussed in section *The (Standard) Singular Value Decomposition (SVD)* (page 49). The residual vector is defined in terms of the cyclic matrix, $H(A)$, so its norm is

$$\|r_{\text{SVD}}\|_2 = \left(\|A\tilde{v} - \tilde{\sigma}\tilde{u}\|_2^2 + \|A^*\tilde{u} - \tilde{\sigma}\tilde{v}\|_2^2\right)^{\frac{1}{2}}, \tag{4.21}$$

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

Given the above definition, the following relation holds

$$|\sigma - \tilde{\sigma}| \leq \|r_{\text{SVD}}\|_2, \tag{4.22}$$

where σ is an exact singular value. The associated error can be obtained in terms of $\|r_{\text{SVD}}\|_2$ with the following function:

```
SVDComputeError(SVD svd,PetscInt j,SVDErrorType type,PetscReal *error);
```

In the case of the GSVD, the function `SVDComputeError()` will compute a residual norm based on the two relations (4.9),

$$\|r_{\text{GSVD}}\|_2 = \left(\|\tilde{s}^2A^*\tilde{u} - \tilde{c}B^*B\tilde{x}\|_2^2 + \|\tilde{c}^2B^*\tilde{v} - \tilde{s}A^*A\tilde{x}\|_2^2\right)^{\frac{1}{2}}, \tag{4.23}$$

where $\tilde{x}$, $\tilde{u}$, $\tilde{v}$ are the computed singular vectors corresponding to $\tilde{\sigma}$, and $\tilde{c}$, $\tilde{s}$ are obtained from $\tilde{\sigma}$ as $\tilde{s} = 1/\sqrt{1+\tilde{\sigma}^2}$ and $\tilde{c} = \tilde{\sigma}\tilde{s}$. See [Alvarruiz *et al.*, 2024] for details.

Similarly, in the HSVD we employ a modified residual

$$\|r_{\text{HSVD}}\|_2 = \left(\|A\tilde{v} - \tilde{\sigma}\tilde{u}\|_2^2 + \|A^*\Omega\tilde{u} - \tilde{\sigma}\tilde{\omega}\tilde{v}\|_2^2\right)^{\frac{1}{2}}, \tag{4.24}$$

where $\tilde{\omega}$ is the corresponding element of the signature $\tilde{\Omega}$ of the definition (4.14).

4.5.2 Controlling and Monitoring Convergence

Similarly to the case of eigensolvers, in `SVD` the number of iterations carried out by the solver can be determined with `SVDGetIterationNumber()`, and the tolerance and maximum number of iterations can be set with `SVDSetTolerances()`. Also, convergence can be monitored with command-line keys `-svd_monitor`, `-svd_monitor_all`, `-svd_monitor_conv`, or graphically with `-svd_monitor draw::draw_lg`, or alternatively with `-svd_monitor_all draw::draw_lg`. See section *Controlling and Monitoring Convergence* (page 23) for additional details.

Table 4: Available possibilities for the convergence criterion, with $Z = A$ in the standard SVD or as defined in equation (4.11) for the GSVD

Convergence criterion	SVDConv	Command line key	Error bound
Absolute	`SVD_CONV_ABS`	`-svd_conv_abs`	$\|r\|$
Relative to eigenvalue	`SVD_CONV_REL`	`-svd_conv_rel`	$\|r\|/\|\lambda\|$
Relative to matrix norms	`SVD_CONV_NORM`	`-svd_conv_norm`	$\|r\|/\|Z\|$
Fixed number of iterations	`SVD_CONV_MAXIT`	`-svd_conv_maxit`	∞
User-defined	`SVD_CONV_USER`	`-svd_conv_user`	*user function*

As in the case of eigensolvers, the user can choose different convergence tests, based on an error bound obtained from the computed residual norm, $\|r\|$. Table *Available possibilities for the convergence criterion* (page 59) lists

the available options. It is worth mentioning the SVD_CONV_MAXIT convergence criterion, which is a bit special. With this criterion, the solver will not compute any residual norms and will stop with a successful status when the maximum number of iterations is reached. This is intended for the SVDRANDOMIZED solver in cases when a low-rank approximation of a matrix needs to be computed instead of accurate singular vectors.

4.5.3 Viewing the Solution

There is support for different kinds of viewers for the solution, as in the case of eigensolvers. One can for instance use -svd_view_values, -svd_view_vectors, -svd_error_relative, or -svd_converged_reason. See the description in section *Viewing the Solution* (page 26).

CHAPTER 5

PEP: Polynomial Eigenvalue Problems

The Polynomial Eigenvalue Problem (PEP) solver object is intended for addressing polynomial eigenproblems of arbitrary degree, $P(\lambda)x = 0$. A particular instance is the quadratic eigenvalue problem (degree 2), which is the case more often encountered in practice. For this reason, part of the description of this chapter focuses specifically on quadratic eigenproblems.

Currently, most PEP solvers are based on linearization, either implicit or explicit. The case of explicit linearization allows the use of eigensolvers from EPS to solve the linearized problem.

5.1 Overview of Polynomial Eigenproblems

In this section, we review some basic properties of the polynomial eigenvalue problem. The main goal is to set up the notation as well as to describe the linearization approaches that will be employed for solving via an EPS object. To simplify, we initially restrict the description to the case of quadratic eigenproblems, and then extend it to the general case of arbitrary degree. For additional background material, the reader is referred to [Tisseur and Meerbergen, 2001]. More information can be found in a paper by Campos and Roman [2016], that focuses specifically on SLEPc implementation of methods based on Krylov iterations on the linearized problem.

5.1.1 Quadratic Eigenvalue Problems

In many applications, e.g., problems arising from second-order differential equations such as the analysis of damped vibrating systems, the eigenproblem to be solved is quadratic,

$$(K + \lambda C + \lambda^2 M)x = 0, \tag{5.1}$$

where $K, C, M \in \mathbb{C}^{n\times n}$ are the coefficients of a polynomial matrix of degree 2, $\lambda \in \mathbb{C}$ is the eigenvalue and $x \in \mathbb{C}^n$ is the eigenvector. As in the case of linear eigenproblems, the eigenvalues and eigenvectors can be complex even in the case that all three matrices are real.

It is important to point out some outstanding differences with respect to the linear eigenproblem. In the quadratic eigenproblem, the number of eigenvalues is $2n$, and the corresponding eigenvectors do not form a linearly independent set. If M is singular, some eigenvalues are infinite. Even when the three matrices are symmetric and positive definite, there is no guarantee that the eigenvalues are real, but still methods can exploit symmetry to some extent. Furthermore, numerical difficulties are more likely than in the linear case, so the computed solution can sometimes be untrustworthy.

If equation (5.1) is written as $P(\lambda)x = 0$, where P is the polynomial matrix, then multiplication by λ^{-2} results in $\mathrm{rev}\, P(\lambda^{-1})x = 0$, where $\mathrm{rev}\, P$ denotes the polynomial matrix with the coefficients of P in the reverse order. In other words, if a method is available for computing the largest eigenvalues, then reversing the roles of M and K results in the computation of the smallest eigenvalues. In general, it is also possible to formulate a spectral

transformation for computing eigenvalues closest to a given target, as discussed in section *Spectral Transformation for PEP* (page 67).

Problem Types

As in the case of linear eigenproblems, there are some particular properties of the coefficient matrices that confer a certain structure to the quadratic eigenproblem, e.g., symmetry of the spectrum with respect to the real or imaginary axes. These structures are important as long as the solvers are able to exploit them.

- Hermitian (symmetric) problems, when M, C, K are all Hermitian (symmetric). Eigenvalues are real or come in complex conjugate pairs. Furthermore, if $M > 0$ and $C, K \geq 0$ then the system is stable, i.e., $\mathrm{Re}(\lambda) \leq 0$.
- Hyperbolic problems, a particular class of Hermitian problems where $M > 0$ and $(x^*Cx)^2 > 4(x^*Mx)(x^*Kx)$ for all nonzero $x \in \mathbb{C}^n$. All eigenvalues are real, and form two separate groups of n eigenvalues, each of them having linearly independent eigenvectors. A particular subset of hyperbolic problems is the class of overdamped problems, where $C > 0$ and $K \geq 0$, in which case all eigenvalues are non-positive.
- Gyroscopic problems, when M, K are Hermitian, $M > 0$, and C is skew-Hermitian, $C = -C^*$. The spectrum is symmetric with respect to the imaginary axis, and in the real case, it has a Hamiltonian structure, i.e., eigenvalues come in quadruples $(\lambda, \bar{\lambda}, -\lambda, -\bar{\lambda})$.

> **Note**
>
> Currently, the problem type is not exploited by PEP solvers, except for a few exceptions. In the future, we may add more support for structure-preserving solvers.

Linearization

It is possible to transform the quadratic eigenvalue problem to a linear generalized eigenproblem $L_0 y = \lambda L_1 y$ by doubling the order of the system, i.e., $L_0, L_1 \in \mathbb{C}^{2n\times 2n}$. There are many ways of doing this. For instance, consider the following two pencils $L(\lambda) = L_0 - \lambda L_1$,

$$\begin{bmatrix} 0 & I \\ -K & -C \end{bmatrix} - \lambda \begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix}, \tag{5.2}$$

$$\begin{bmatrix} -K & 0 \\ 0 & I \end{bmatrix} - \lambda \begin{bmatrix} C & M \\ I & 0 \end{bmatrix}. \tag{5.3}$$

Both of them have the same eigenvalues as the quadratic eigenproblem, and the corresponding eigenvectors can be expressed as

$$y = \begin{bmatrix} x \\ x\lambda \end{bmatrix}, \tag{5.4}$$

where x is the eigenvector of the quadratic eigenproblem.

Other **non-symmetric** linearizations can be obtained by a linear combination of equations (5.2) and (5.3),

$$\begin{bmatrix} -\beta K & \alpha I \\ -\alpha K & -\alpha C + \beta I \end{bmatrix} - \lambda \begin{bmatrix} \alpha I + \beta C & \beta M \\ \beta I & \alpha M \end{bmatrix}. \tag{5.5}$$

for any $\alpha, \beta \in \mathbb{R}$. The linearizations (5.2) and (5.3) are particular cases of equation (5.5) taking $(\alpha, \beta) = (1, 0)$ and $(0, 1)$, respectively.

Symmetric linearizations are useful for the case that M, C, and K are all symmetric (Hermitian), because the resulting matrix pencil is symmetric (Hermitian), although indefinite:

$$\begin{bmatrix} \beta K & \alpha K \\ \alpha K & \alpha C - \beta M \end{bmatrix} - \lambda \begin{bmatrix} \alpha K - \beta C & -\beta M \\ -\beta M & -\alpha M \end{bmatrix}, \tag{5.6}$$

And for gyroscopic problems, we can consider **Hamiltonian** linearizations,

$$\begin{bmatrix} \alpha K & -\beta K \\ \alpha C + \beta M & \alpha K \end{bmatrix} - \lambda \begin{bmatrix} \beta M & \alpha K + \beta C \\ -\alpha M & \beta M \end{bmatrix}, \tag{5.7}$$

where one of the matrices is Hamiltonian and the other one is skew-Hamiltonian if (α, β) is $(1, 0)$ or $(0, 1)$.

In SLEPc, the `PEPLINEAR` solver is based on using one of the above linearizations for solving the quadratic eigenproblem. This solver makes use of linear eigensolvers from the `EPS` package.

We could also consider the *reversed* forms, e.g., the reversed form of equation (5.3) is

$$\begin{bmatrix} -C & -M \\ I & 0 \end{bmatrix} - \frac{1}{\lambda} \begin{bmatrix} K & 0 \\ 0 & I \end{bmatrix}, \tag{5.8}$$

which is equivalent to the form (5.2) for the problem $\mathrm{rev}\, P(\lambda^{-1})x = 0$. These reversed forms are not implemented in SLEPc, but the user can use them simply by reversing the roles of M and K, and considering the reciprocals of the computed eigenvalues. Alternatively, this can be viewed as a particular case of the spectral transformation (with $\sigma = 0$), see section *Spectral Transformation for PEP* (page 67).

5.1.2 Polynomials of Arbitrary Degree

In general, the polynomial eigenvalue problem can be formulated as

$$P(\lambda)x = 0, \tag{5.9}$$

where P is an $n \times n$ polynomial matrix of degree d. An n-vector $x \neq 0$ satisfying this equation is called an eigenvector associated with the corresponding eigenvalue λ.

We start by considering the case where P is expressed in terms of the monomial basis,

$$P(\lambda) = A_0 + A_1\lambda + A_2\lambda^2 + \cdots + A_d\lambda^d, \tag{5.10}$$

where $A_0, \ldots, A_d$ are the $n \times n$ coefficient matrices. As before, the problem can be solved via some kind of linearization. One of the most commonly used ones is the first companion form

$$L(\lambda) = L_0 - \lambda L_1, \tag{5.11}$$

where the related linear eigenproblem is $L(\lambda)y = 0$, with

$$L_0 = \begin{bmatrix} & I & & \\ & & \ddots & \\ & & & I \\ -A_0 & -A_1 & \cdots & -A_{d-1} \end{bmatrix}, \quad L_1 = \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & A_d \end{bmatrix}, \quad y = \begin{bmatrix} x \\ x\lambda \\ \vdots \\ x\lambda^{d-1} \end{bmatrix}. \tag{5.12}$$

This is the generalization of equation (5.2).

The definition of vector y above contains the successive powers of λ. For large polynomial degree, these values may produce overflow in finite precision computations, or at least lead to numerical instability of the algorithms due to the wide difference in magnitude of the eigenvector entries. For this reason, it is generally recommended to work with non-monomial polynomial bases whenever the degree is not small, e.g., for $d > 5$.

In the most general formulation of the polynomial eigenvalue problem, P is expressed as

$$P(\lambda) = A_0\phi_0(\lambda) + A_1\phi_1(\lambda) + \cdots + A_d\phi_d(\lambda), \tag{5.13}$$

where ϕ_i are the members of a given polynomial basis, for instance, some kind of orthogonal polynomials such as Chebyshev polynomials of the first kind. In that case, the expression of y in equation (5.12) contains $\phi_0(\lambda), \ldots, \phi_d(\lambda)$ instead of the powers of λ. Correspondingly, the form of L_0 and L_1 is different for each type of polynomial basis.

Avoiding the Linearization

An alternative to linearization is to directly perform a projection of the polynomial eigenproblem. These methods enforce a Galerkin condition on the polynomial residual, $P(\theta)u \perp \mathcal{K}$. Here, the subspace $\mathcal{K}$ can be built in various ways, for instance with the Jacobi-Davidson method. This family of methods need not worry about operating with vectors of dimension dn. The downside is that computing more than one eigenvalue is more difficult, since usual deflation strategies cannot be applied. For a detailed description of the polynomial Jacobi-Davidson method in SLEPc, see [Campos and Roman, 2020].

5.2 Basic Usage

The user interface of the PEP package is very similar to EPS. For basic usage, the most noteworthy difference is that all coefficient matrices A_i have to be supplied in the form of an array of Mat.

A basic example code for solving a polynomial eigenproblem with PEP is shown in listing *Example code for basic solution with PEP* (page 64), where the code for building matrices `A[0]`, `A[1]`, ... is omitted. The required steps are the same as those described in chapter *EPS: Eigenvalue Problem Solver* (page 13) for the linear eigenproblem. As always, the solver context is created with `PEPCreate()`. The coefficient matrices are provided with `PEPSetOperators()`, and the problem type is specified with `PEPSetProblemType()`. Calling `PEPSetFromOptions()` allows the user to set up various options through the command line. The call to `PEPSolve()` invokes the actual solver. Then, the solution is retrieved with `PEPGetConverged()` and `PEPGetEigenpair()`. Finally, `PEPDestroy()` destroys the object.

Listing 1: Example code for basic solution with PEP

```
#define NMAT 5
PEP         pep;       /*  eigensolver context  */
Mat         A[NMAT];   /*  coefficient matrices */
Vec         xr, xi;    /*  eigenvector, x       */
PetscScalar kr, ki;    /*  eigenvalue, k        */
PetscInt    j, nconv;
PetscReal   error;

PEPCreate(PETSC_COMM_WORLD, &pep);
PEPSetOperators(pep, NMAT, A);
PEPSetProblemType(pep, PEP_GENERAL);  /* optional */
PEPSetFromOptions(pep);
PEPSolve(pep);
PEPGetConverged(pep, &nconv);
for (j=0;j<nconv;j++) {
  PEPGetEigenpair(pep, j, &kr, &ki, xr, xi);
  PEPComputeError(pep, j, PEP_ERROR_BACKWARD, &error);
}
PEPDestroy(&pep);
```

5.3 Defining the Problem

Table 1: Polynomial bases available to represent the polynomial matrix in PEP

Polynomial Basis	PEPBasis	Options Database
Monomial	PEP_BASIS_MONOMIAL	monomial
Chebyshev (1st kind)	PEP_BASIS_CHEBYSHEV1	chebyshev1
Chebyshev (2nd kind)	PEP_BASIS_CHEBYSHEV2	chebyshev2
Legendre	PEP_BASIS_LEGENDRE	legendre
Laguerre	PEP_BASIS_LAGUERRE	laguerre
Hermite	PEP_BASIS_HERMITE	hermite

As explained in section *Polynomials of Arbitrary Degree* (page 63), the polynomial matrix $P(\lambda)$ can be expressed in terms of the monomials $1, \lambda, \lambda^2, \ldots$, or in a non-monomial basis as in equation (5.13). Hence, when defining the problem we must indicate which is the polynomial basis to be used as well as the coefficient matrices A_i in

that basis representation. By default, a monomial basis is used. Other possible bases are listed in table *Polynomial bases available to represent the polynomial matrix in PEP* (page 64), and can be set with:

```
PEPSetBasis(PEP pep,PEPBasis basis);
```

or with the command-line key `-pep_basis <name>`. The matrices are passed with:

```
PEPSetOperators(PEP pep,PetscInt nmat,Mat A[]);
```

Table 2: Problem types considered in PEP

Problem Type	PEPProblemType	Command line key
General	PEP_GENERAL	-pep_general
Hermitian	PEP_HERMITIAN	-pep_hermitian
Hyperbolic	PEP_HYPERBOLIC	-pep_hyperbolic
Gyroscopic	PEP_GYROSCOPIC	-pep_gyroscopic

As mentioned in section *Quadratic Eigenvalue Problems* (page 61), it is possible to distinguish among different problem types. The problem types currently supported for PEP are listed in table *Problem types considered in PEP* (page 65). The goal when choosing an appropriate problem type is to let the solver exploit the underlying structure, in order to possibly compute the solution more accurately with less floating-point operations. When in doubt, use the default problem type (PEP_GENERAL).

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

```
PEPSetProblemType(PEP pep,PEPProblemType type);
```

Note

Currently, the problem type is ignored in most solvers and it is taken into account only in some cases for the quadratic eigenproblem only.

Apart from the polynomial basis and the problem type, the definition of the problem is completed with the number and location of the eigenvalues to compute. This is done very much like in EPS, but with minor differences.

The number of eigenvalues (and eigenvectors) to compute, nev, is specified with the function:

```
PEPSetDimensions(PEP pep,PetscInt nev,PetscInt ncv,PetscInt mpd);
```

The default is to compute only one. This function also allows control over the dimension of the subspaces used internally. The second argument, ncv, is the number of column vectors to be used by the solution algorithm, that is, the largest dimension of the working subspace. The third argument, mpd, is the maximum projected dimension. These parameters can also be set from the command line with `-pep_nev`, `-pep_ncv` and `-pep_mpd`.

For the selection of the portion of the spectrum of interest, there are several alternatives listed in table *Available possibilities for selection of the eigenvalues of interest in PEP* (page 66), to be selected with:

```
PEPSetWhichEigenpairs(PEP pep,PEPWhich which);
```

The default is to compute the largest magnitude eigenvalues. For the sorting criteria relative to a target value, the scalar τ must be specified with:

```
PEPSetTarget(PEP pep,PetscScalar target);
```

or in the command-line with `-pep_target`. As in EPS, complex values of τ are allowed only in complex scalar SLEPc builds. The criteria relative to a target must be used in combination with a spectral transformation as explained in section *Spectral Transformation for PEP* (page 67).

There is also support for spectrum slicing, that is, computing all eigenvalues in a given interval, see section *Spectrum Slicing for PEP* (page 68). For this, the user has to specify the computational interval with:

```
PEPSetInterval(PEP pep,PetscReal a,PetscReal b);
```

or equivalently with `-pep_interval a,b`.

Finally, we mention that the use of regions for filtering is also available in PEP, see section *Specifying a Region for Filtering* (page 28).

Table 3: Available possibilities for selection of the eigenvalues of interest in PEP

PEPWhich	Command line key	Sorting criterion
PEP_LARGEST_MAGNITUDE	-pep_largest_magnitude	Largest $\|\lambda\|$
PEP_SMALLEST_MAGNITUDE	-pep_smallest_magnitude	Smallest $\|\lambda\|$
PEP_LARGEST_REAL	-pep_largest_real	Largest $\mathrm{Re}(\lambda)$
PEP_SMALLEST_REAL	-pep_smallest_real	Smallest $\mathrm{Re}(\lambda)$
PEP_LARGEST_IMAGINARY	-pep_largest_imaginary	Largest $\mathrm{Im}(\lambda)$[1]
PEP_SMALLEST_IMAGINARY	-pep_smallest_imaginary	Smallest $\mathrm{Im}(\lambda)$[1]
PEP_TARGET_MAGNITUDE	-pep_target_magnitude	Smallest $\|\lambda - \tau\|$
PEP_TARGET_REAL	-pep_target_real	Smallest $\|\mathrm{Re}(\lambda - \tau)\|$
PEP_TARGET_IMAGINARY	-pep_target_imaginary	Smallest $\|\mathrm{Im}(\lambda - \tau)\|$
PEP_ALL	-pep_all	All $\lambda \in [a, b]$
PEP_WHICH_USER	-pep_which_user	*user-defined*

5.4 Selecting the Solver

The solution method can be specified procedurally with:

```
PEPSetType(PEP pep,PEPType method);
```

or via the options database command `-pep_type` followed by the name of the method. The methods currently available in PEP are listed in table *Polynomial eigenvalue solvers available in the PEP module* (page 66). All solvers except PEPJD are based on the linearization explained above, whereas PEPJD performs a projection on the polynomial problem (without linearizing).

The default solver is PEPTOAR. The Two-level Orthogonal Arnoldi (TOAR) method is a stable algorithm for building an Arnoldi factorization of the linearization (5.11) without explicitly creating matrices L_0, L_1, and represents the Krylov basis in a compact way. Symmetric TOAR (STOAR) is a variant of TOAR that exploits symmetry (requires PEP_HERMITIAN or PEP_HYPERBOLIC problem types). Quadratic Arnoldi (Q-Arnoldi) is related to TOAR and follows a similar approach.

Table 4: Polynomial eigenvalue solvers available in the PEP module

Method	PEPType	Options Database	Polynomial Degree	Polynomial Basis
TOAR	PEPTOAR	toar	Arbitrary	Any
Symmetric TOAR	PEPSTOAR	stoar	Quadratic	Monomial
Q-Arnoldi	PEPQARNOLDI	qarnoldi	Quadratic	Monomial
Linearization via EPS	PEPLINEAR	linear	Arbitrary	Any
Jacobi-Davidson	PEPJD	jd	Arbitrary	Monomial
Contour integral SS	PEPCISS	ciss	Arbitrary	Monomial

The PEPLINEAR method carries out an explicit linearization of the polynomial eigenproblem, as described in section *Overview of Polynomial Eigenproblems* (page 61), resulting in a generalized eigenvalue problem that is handled by an EPS object created internally. If required, this EPS object can be extracted with the operation:

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

```
PEPLinearGetEPS(PEP pep,EPS *eps);
```

This allows the application programmer to set any of the EPS options directly within the code. Also, it is possible to change the EPS options through the command-line, simply by prefixing the EPS options with -pep_linear_.

In PEPLINEAR, if the eigenproblem is quadratic, the expression used in the linearization is dictated by the problem type set with PEPProblemType, which chooses from non-symmetric (5.5), symmetric (5.6), and Hamiltonian (5.7) linearizations. The parameters (α, β) of these linearizations can be set with:

```
PEPLinearSetLinearization(PEP pep,PetscReal alpha,PetscReal beta);
```

For polynomial eigenproblems with degree $d > 2$ the linearization is the one described in section *Polynomials of Arbitrary Degree* (page 63).

Another option of the PEPLINEAR solver is whether the matrices of the linearized problem are created explicitly or not. This is set with:

```
PEPLinearSetExplicitMatrix(PEP pep,PetscBool explicit);
```

The explicit matrix option is available only for quadratic eigenproblems (higher degree polynomials are always handled implicitly). In the case of explicit creation, matrices L_0 and L_1 are created as true Mat's, with explicit storage, whereas the implicit option works with *shell* Mat's that operate only with the constituent blocks M, C and K (or A_i in the general case). The explicit case requires more memory but gives more flexibility, e.g., for choosing a preconditioner. Some examples of usage via the command line are shown at the end of next section.

5.5 Spectral Transformation for PEP

For computing eigenvalues in the interior of the spectrum (closest to a target τ), it is necessary to use a spectral transformation. In PEP solvers this is handled via an ST object as in the case of linear eigensolvers. It is possible to proceed with no spectral transformation (shift) or with shift-and-invert. Every PEP object has an ST object internally.

The spectral transformation can be applied either to the polynomial problem or its linearization. We illustrate it first for the quadratic case.

Given the quadratic eigenproblem in equation (5.1), it is possible to define the transformed problem

$$(K_\sigma + \theta C_\sigma + \theta^2 M_\sigma)x = 0, \tag{5.14}$$

where the coefficient matrices are

$$\begin{aligned} K_\sigma &= M, \\ C_\sigma &= C + 2\sigma M, \\ M_\sigma &= \sigma^2 M + \sigma C + K, \end{aligned} \tag{5.15}$$

and the relation between the eigenvalue of the original eigenproblem, λ, and the transformed one, θ, is $\theta = (\lambda - \sigma)^{-1}$ as in the case of the linear eigenvalue problem. See chapter *ST: Spectral Transformation* (page 35) for additional details.

The polynomial eigenvalue problem of equation (5.14) corresponds to the reversed form of the shifted polynomial, $\mathrm{rev}\, P(\theta)$. The extension to polynomial matrices of arbitrary degree is also possible, where the coefficients of $\mathrm{rev}\, P(\theta)$ have the general form

$$T_k = \sum_{j=0}^{d-k} \binom{j+k}{k} \sigma^j A_{j+k}, \qquad k = 0, \ldots, d. \tag{5.16}$$

The way this is implemented in SLEPc is that the ST object is in charge of computing the T_k matrices, so that the PEP solver operates with these matrices as it would with the original A_i matrices, without changing its behavior. We say that ST performs the transformation.

An alternative would be to apply the shift-and-invert spectral transformation to the linearization (5.11) in a smart way, making the polynomial eigensolver aware of this fact so that it can exploit the block structure of the linearization. Let $S_\sigma := (L_0 - \sigma L_1)^{-1} L_1$, then when the solver needs to extend the Arnoldi basis with an operation such as $z = S_\sigma w$, a linear solve is required with the form

$$\begin{bmatrix} -\sigma I & I & & & \\ & -\sigma I & \ddots & & \\ & & \ddots & I & \\ & & & -\sigma I & I \\ -A_0 & -A_1 & \cdots & -\tilde{A}_{d-2} & -\tilde{A}_{d-1} \end{bmatrix} \begin{bmatrix} z^0 \\ z^1 \\ \vdots \\ z^{d-2} \\ z^{d-1} \end{bmatrix} = \begin{bmatrix} w^0 \\ w^1 \\ \vdots \\ w^{d-2} \\ A_d w^{d-1} \end{bmatrix}, \tag{5.17}$$

with $\tilde{A}_{d-2} = A_{d-2} + \sigma I$ and $\tilde{A}_{d-1} = A_{d-1} + \sigma A_d$. From the block LU factorization, it is possible to derive a simple recurrence to compute z^i, with one of the steps involving a linear solve with $P(\sigma)$.

Implementing the latter approach is more difficult (especially if different polynomial bases must be supported), and requires an intimate relation with the PEP solver. That is why it is only available currently in the default solver (TOAR) and in PEPLINEAR without explicit matrix. In order to choose between the two approaches, the user can set a flag with:

```
STSetTransform(ST st,PetscBool flg);
```

(or in the command line `-st_transform`) to activate the first one (ST performs the transformation). Note that this flag belongs to ST, not PEP (use `PEPGetST()` to extract it).

In terms of overall computational cost, both approaches are roughly equivalent, but the advantage of the second one is not having to store the T_k matrices explicitly. It may also be slightly more accurate. Hence, the `STSetTransform()` flag is turned off by default.

A command line example would be:

```
$ ./ex16 -pep_nev 12 -pep_type toar -pep_target 0 -st_type sinvert
```

The example computes 12 eigenpairs closest to the origin with TOAR and shift-and-invert. The `-st_transform` could be added optionally to switch to ST being in charge of the transformation. The same example with Q-Arnoldi would be

```
$ ./ex16 -pep_nev 12 -pep_type qarnoldi -pep_target 0 -st_type sinvert -st_transform
```

where in this case `-st_transform` would be set as default if not specified.

As a complete example of how to solve a quadratic eigenproblem via explicit linearization with explicit construction of the L_0 and L_1 matrices, consider the following command line:

```
$ ./sleeper -pep_type linear -pep_target -10 -pep_linear_st_type sinvert -pep_linear_st_ksp_type preonly -
↪pep_linear_st_pc_type lu -pep_linear_st_pc_factor_mat_solver_type mumps -pep_linear_st_mat_mumps_icntl_
↪14 100 -pep_linear_explicitmatrix
```

This example uses MUMPS for solving the associated linear systems, see section *Solution of Linear Systems* (page 41) for details. The following command line example illustrates how to solve the same problem without explicitly forming the matrices. Note that in this case the ST options are not prefixed with `-pep_linear_` since now they do not refer to the ST within the PEPLINEAR solver but the general ST associated to PEP.

```
$ ./sleeper -pep_type linear -pep_target -10 -st_type sinvert -st_ksp_type preonly -st_pc_type lu -st_pc_
↪factor_mat_solver_type mumps -st_mat_mumps_icntl_14 100
```

5.5.1 Spectrum Slicing for PEP

Similarly to the spectrum slicing technique available in linear symmetric-definite eigenvalue problems (cf. *Spectrum Slicing* (page 44)), it is possible to compute all eigenvalues in a given interval $[a, b]$ for the case of hyperbolic quadratic eigenvalue problems (PEP_HYPERBOLIC). In more general symmetric (or Hermitian) quadratic eigenproblems (PEP_HERMITIAN), it may also be possible to do spectrum slicing provided that computing inertia is feasible, which essentially means that all eigenvalues in the interval must be real and of the same definite type.

This computation is available only in the PEPSTOAR solver. The spectrum slicing mechanism implemented in PEP is very similar to the one described in section *Spectrum Slicing* (page 44) for linear problems, except for the multi-communicator option which is not implemented yet.

A command line example is the following:

```
$ ./spring -n 300 -pep_hermitian -pep_interval -10.1,-9.5 -pep_type stoar -st_type sinvert -st_ksp_type␣
↪preonly -st_pc_type cholesky
```

In hyperbolic problems, where eigenvalues form two separate groups of n eigenvalues, it will be necessary to explicitly set the problem type to `-pep_hyperbolic` if the interval $[a, b]$ includes eigenvalues from both groups.

Additional details can be found in [Campos and Roman, 2020].

5.6 Retrieving the Solution

After the call to `PEPSolve()` has finished, the computed results are stored internally. The procedure for retrieving the computed solution is exactly the same as in the case of EPS. The user has to call `PEPGetConverged()` first, to obtain the number of converged solutions, then call `PEPGetEigenpair()` repeatedly within a loop, once per each eigenvalue-eigenvector pair. The same considerations relative to complex eigenvalues apply, see section *Retrieving the Solution* (page 21) for additional details.

Controlling and Monitoring Convergence: As in the case of EPS, in PEP the number of iterations carried out by the solver can be determined with `PEPGetIterationNumber()`, and the tolerance and maximum number of iterations can be set with `PEPSetTolerances()`. Also, convergence can be monitored with command-line keys `-pep_monitor`, `-pep_monitor_all`, `-pep_monitor_conv`, `-pep_monitor draw::draw_lg`, or `-pep_monitor_all draw::draw_lg`. See section *Controlling and Monitoring Convergence* (page 23) for additional details.

Viewing the Solution: Likewise to linear eigensolvers, there is support for various kinds of viewers for the solution. One can for instance use `-pep_view_values`, `-pep_view_vectors`, `-pep_error_relative`, or `-pep_converged_reason`. See description in section *Viewing the Solution* (page 26).

5.6.1 Reliability of the Computed Solution

Table 5: Possible expressions for computing error bounds

Error type	PEPErrorType	Command line key	Error bound
Absolute error	PEP_ERROR_ABSOLUTE	-pep_error_absolute	$\|r\|$
Relative error	PEP_ERROR_RELATIVE	-pep_error_relative	$\|r\|/\|\lambda\|$
Backward error	PEP_ERROR_BACKWARD	-pep_error_backward	$\|r\|/(\sum_j \|A_j\|\|\lambda_i\|^j)$

As in the case of linear problems, the following function:

```
PEPComputeError(PEP pep,PetscInt j,PEPErrorType type,PetscReal *error);
```

is available to assess the accuracy of the computed solutions. This error is based on the computation of the 2-norm of the residual vector, defined as

$$r = P(\tilde{\lambda})\tilde{x}, \tag{5.18}$$

where $\tilde{\lambda}$ and $\tilde{x}$ represent any of the nconv computed eigenpairs delivered by `PEPGetEigenpair()`. From the residual norm, the error bound can be computed in different ways, see table *Possible expressions for computing error bounds* (page 69). It is usually recommended to assess the accuracy of the solution using the backward error, defined as

$$\eta(\tilde{\lambda}, \tilde{x}) = \frac{\|r\|}{\sum_{j=0}^{d} \|A_j\| |\tilde{\lambda}|^j \|\tilde{x}\|}, \tag{5.19}$$

where d is the degree of the polynomial. Note that the eigenvector is always assumed to have unit norm.

Similar expressions can be used in the convergence criterion used to accept converged eigenpairs internally by the solver. The convergence test can be set via the corresponding command-line switch (see table *Available possibilities for the convergence criterion* (page 70)) or with `PEPSetConvergenceTest()`.

Table 6: Available possibilities for the convergence criterion

Convergence criterion	PEPConv	Command line key	Error bound
Absolute	PEP_CONV_ABS	-pep_conv_abs	$\|r\|$
Relative to eigenvalue	PEP_CONV_REL	-pep_conv_rel	$\|r\|/\|\lambda\|$
Relative to matrix norms	PEP_CONV_NORM	-pep_conv_norm	$\|r\|/(\sum_j \|A_j\|\|\lambda\|^j)$
User-defined	PEP_CONV_USER	-pep_conv_user	*user function*

5.6.2 Scaling

When solving a quadratic eigenproblem via linearization, an accurate solution of the generalized eigenproblem does not necessarily imply a similar level of accuracy for the quadratic problem. Tisseur [2000] shows that in the case of the linearization (5.2), a small backward error in the generalized eigenproblem guarantees a small backward error in the quadratic eigenproblem. However, this holds only if M, C and K have a similar norm.

When the norm of M, C and K vary widely, Tisseur [2000] recommends to solve the scaled problem, defined as

$$(\mu^2 M_\alpha + \mu C_\alpha + K)x = 0, \tag{5.20}$$

with $\mu = \lambda/\alpha$, $M_\alpha = \alpha^2 M$ and $C_\alpha = \alpha C$, where α is a scaling factor. Ideally, α should be chosen in such a way that the norms of M_α, C_α and K have similar magnitude. A tentative value would be $\alpha = \sqrt{\frac{\|K\|_\infty}{\|M\|_\infty}}$.

In the general case of polynomials of arbitrary degree, a similar scheme is also possible, but it is not clear how to choose α to achieve the same goal. Betcke [2008] proposes such a scaling scheme as well as more general diagonal scalings $D_\ell P(\lambda) D_r$. In SLEPc, we provide these types of scalings, whose settings can be tuned with:

```
PEPSetScale(PEP pep,PEPScale scale,PetscReal alpha,Vec Dl,Vec Dr,PetscInt its,PetscReal lambda);
```

See the manual page for details and the description in [Campos and Roman, 2016].

5.6.3 Extraction

Some of the eigensolvers provided in the PEP package are based on solving the linearized eigenproblem of equation (5.12). From the eigenvector y of the linearization, it is possible to extract the eigenvector x of the polynomial eigenproblem. The most straightforward way is to take the first block of y, but there are other, more elaborate extraction strategies. For instance, one may compute the norm of the residual (5.18) for every block of y, and take the one that gives the smallest residual. The different extraction techniques may be selected with:

```
PEPSetExtract(PEP pep,PEPExtract extract);
```

For additional information, see [Campos and Roman, 2016].

5.6.4 Iterative Refinement

As mentioned above, scaling can sometimes improve the accuracy of the computed solution considerably, in the case that the coefficient matrices A_i are very different in norm. Still, even when the matrix norms are well balanced the accuracy can sometimes be unacceptably low. The reason is that methods based on linearization are not always backward stable, that is, even if the computation of the eigenpairs of the linearization is done in a stable way, there is no guarantee that the extracted polynomial eigenpairs satisfy the given tolerance.

If good accuracy is required, one possibility is to perform a few steps of iterative refinement on the solution computed by the polynomial eigensolver algorithm. Iterative refinement can be seen as the Newton method applied to a set of nonlinear equations related to the polynomial eigenvalue problem [Betcke and Kressner, 2011]. It is well known that global convergence of Newton's iteration is guaranteed only if the initial guess is close enough to the exact solution, so we still need an eigensolver such as TOAR to compute this initial guess.

Iterative refinement can be very costly (sometimes a single refinement step is more expensive than the whole iteration to compute the initial guess with TOAR), that is why in SLEPc it is disabled by default. When the user activates it, the computation of Newton iterations will take place within `PEPSolve()` as a final stage (identified as `PEPRefine()` in the `-log_view` report).

```
PEPSetRefine(PEP pep,PEPRefine refine,PetscInt npart,PetscReal tol,PetscInt its,PEPRefineScheme scheme);
```

There are two types of refinement, identified as *simple* and *multiple*. The first one performs refinement on each eigenpair individually, while the second one considers the computed invariant pair as a whole. This latter approach is more costly but it is expected to be more robust in the presence of multiple eigenvalues.

In `PEPSetRefine()`, the argument `npart` indicates the number of partitions in which the communicator must be split. This can sometimes improve the scalability when refining many eigenpairs.

Additional details can be found in [Campos and Roman, 2016].

CHAPTER 6

NEP: Nonlinear Eigenvalue Problems

The Nonlinear Eigenvalue Problem (NEP) solver object covers the general case where the eigenproblem is nonlinear with respect to the eigenvalue, but it cannot be expressed in terms of a polynomial. We will write the problem as $T(\lambda)x = 0$, where T is a matrix-valued function of the eigenvalue λ. Note that NEP does not cover the even more general case of having a nonlinear dependence on the eigenvector x.

In terms of the user interface, NEP is quite similar to previously discussed solvers. The main difference is how to represent the function T. We will show different alternatives for this.

The NEP module of SLEPc has been explained with more detail in [Campos and Roman, 2021], including an algorithmic description of the implemented solvers.

6.1 General Nonlinear Eigenproblems

As in previous chapters, we first set up the notation and briefly review basic properties of the eigenvalue problems to be addressed. In this case, we focus on general nonlinear eigenproblems, that is, those that cannot be expressed in a simpler form such as a polynomial eigenproblem. These problems arise in many applications, such as the discretization of delay differential equations. Some of the methods designed to solve such problems are based on Newton-type iterations, so in some ways NEP has similarities to PETSc's nonlinear solvers SNES. For background material on the nonlinear eigenproblem, the reader is referred to [Güttel and Tisseur, 2017, Mehrmann and Voss, 2004].

We consider nonlinear eigenvalue problems of the form

$$T(\lambda)x = 0, \qquad x \neq 0, \tag{6.1}$$

where $T : \Omega \to \mathbb{C}^{n\times n}$ is a matrix-valued function that is analytic on an open set of the complex plane $\Omega \subseteq \mathbb{C}$. Assuming that the problem is regular, that is, $\det T(\lambda)$ does not vanish identically, any pair (λ, x) satisfying equation (6.1) is an eigenpair, where $\lambda \in \Omega$ is the eigenvalue and $x \in \mathbb{C}^n$ is the eigenvector. Linear and polynomial eigenproblems are particular cases of equation (6.1).

An example application is the rational eigenvalue problem

$$-Kx + \lambda Mx + \sum_{j=1}^{k} \frac{\lambda}{\sigma_j - \lambda} C_j x = 0, \tag{6.2}$$

arising in the study of free vibration of plates with elastically attached masses. Here, all matrices are symmetric, K and M are positive-definite and C_j have small rank. Another example comes from the discretization of parabolic partial differential equations with time delay τ, resulting in

$$(-\lambda I + A + e^{-\tau\lambda}B)x = 0. \tag{6.3}$$

Split Form: Equation (6.1) can always be rewritten as

$$\big(A_0 f_0(\lambda) + A_1 f_1(\lambda) + \cdots + A_{\ell-1} f_{\ell-1}(\lambda)\big)x = \left(\sum_{i=0}^{\ell-1} A_i f_i(\lambda)\right) x = 0, \tag{6.4}$$

where A_i are $n \times n$ matrices and $f_i : \Omega \to \mathbb{C}$ are analytic functions. We will call equation (6.4) the split form of the nonlinear eigenvalue problem. Often, the formulation arising from applications already has this form, as illustrated by the examples above. Also, a polynomial eigenvalue problem fits this form, where in this case the f_i functions are the polynomial bases of degree i, either monomial or non-monomial.

6.2 Defining the Problem

The user interface of the NEP package is quite similar to EPS and PEP. As mentioned above, the main difference is the way in which the eigenproblem is defined. In equation *Using Callback Functions* (page 74), we focus on the case where the problem is defined as in PETSc's nonlinear solvers SNES, that is, providing user-defined callback functions to compute the nonlinear function matrix, $T(\lambda)$, and its derivative, $T'(\lambda)$. We defer the discussion of using the split form of the nonlinear eigenproblem to section *Expressing the NEP in Split Form* (page 76).

6.2.1 Using Callback Functions

A sample code for solving a nonlinear eigenproblem with NEP is shown in listing *Example code for basic solution with NEP using callbacks* (page 74). The usual steps are performed, starting with the creation of the solver context with `NEPCreate()`. Then the problem matrices are defined, see discussion below. The call to `NEPSetFromOptions()` captures relevant options specified in the command line. The actual solver is invoked with `NEPSolve()`. Then, the solution is retrieved with `NEPGetConverged()` and `NEPGetEigenpair()`. Finally, `NEPDestroy()` destroys the object.

Listing 1: Example code for basic solution with NEP using callbacks

```
NEP         nep;       /*  eigensolver context */
Mat         F, J;      /*  Function and Jacobian matrices  */
Vec         xr, xi;    /*  eigenvector, x       */
PetscScalar kr, ki;    /*  eigenvalue, k        */
ApplCtx     ctx;       /*  user-defined context */
PetscInt    j, nconv;
PetscReal   error;

NEPCreate(PETSC_COMM_WORLD, &nep);
/* create and preallocate F and J matrices */
NEPSetFunction(nep, F, F, FormFunction, &ctx);
NEPSetJacobian(nep, J, FormJacobian, &ctx);
NEPSetFromOptions(nep);
NEPSolve(nep);
NEPGetConverged(nep, &nconv);
for (j=0;j<nconv;j++) {
  NEPGetEigenpair(nep, j, &kr, &ki, xr, xi);
  NEPComputeError(nep, j, NEP_ERROR_RELATIVE, &error);
}
NEPDestroy(&nep);
```

In SNES, the usual way to define a set of nonlinear equations $F(x) = 0$ is to provide two user-defined callback functions, one to compute the residual vector, $r = F(x)$ for a given x, and another one to evaluate the Jacobian matrix, $J(x) = F'(x)$. In the case of NEP there are some differences, since the function T depends on the parameter λ only. For a given value of λ and its associated vector x, the residual vector is defined as

$$r = T(\lambda)x. \tag{6.5}$$

We require the user to provide a callback function to evaluate $T(\lambda)$, rather than computing the residual r. Once $T(\lambda)$ has been built, NEP solvers can compute its action on any vector x. Regarding the derivative, in NEP we use $T'(\lambda)$, which will be referred to as the Jacobian matrix by analogy to SNES. This matrix must be computed with another callback function.

> **Note**
>
> The derivative $T'(\lambda)$ is optional in some cases, depending on the selected solver.

Hence, both callback functions must compute a matrix. The nonzero pattern of these matrices does not usually change, so they must be created at the beginning of the solution process. Then, these Mat objects are passed to the solver, together with the pointers to the callback functions, with:

```
NEPSetFunction(NEP nep,Mat F,Mat P,NEPFunctionFn *fun,void *ctx);
NEPSetJacobian(NEP nep,Mat J,NEPJacobianFn *jac,void *ctx)
```

The argument `ctx` is an optional user-defined context intended to contain application-specific parameters required to build $T(\lambda)$ or $T'(\lambda)$, and it is received as the last argument in the callback functions. The callback routines also get an argument containing the value of λ at which T or T' must be evaluated. Note that the `NEPSetFunction()` callback takes two Mat arguments instead of one. The rationale for this is that some `NEP` solvers require to perform linear solves with $T(\lambda)$ within the iteration (in SNES this is done with the Jacobian), so $T(\lambda)$ will be passed as the coefficient matrix to a KSP object. The second Mat argument `P` is the matrix from which the preconditioner is constructed (which is usually the same as `F`).

There is the possibility of solving the problem in a matrix-free fashion, that is, just implementing subroutines that compute the action of $T(\lambda)$ or $T'(\lambda)$ on a vector, instead of having to explicitly compute all nonzero entries of these two matrices. The example program ex21.c illustrates this, using the concept of *shell* matrices (see section *Supported Matrix Types* (page 96) for details).

Parameters for Problem Definition

Once T and T' have been set up, the definition of the problem is completed with the number and location of the eigenvalues to compute, in a similar way as eigensolvers discussed in previous chapters.

The number of requested eigenvalues (and eigenvectors), `nev`, is established with:

```
NEPSetDimensions(NEP nep,PetscInt nev,PetscInt ncv,PetscInt mpd);
```

By default, `nev=1`. The other two arguments control the dimension of the subspaces used internally (the number of column vectors, `ncv`, and the maximum projected dimension, `mpd`), although they are relevant only in eigensolvers based on subspace projection (basic algorithms ignore them). There are command-line keys for these parameters: `-nep_nev`, `-nep_ncv` and `-nep_mpd`.

Table 1: Available possibilities for selection of the eigenvalues of interest in `NEP`

NEPWhich	Command line key	Sorting criterion
NEP_LARGEST_MAGNITUDE	-nep_largest_magnitude	Largest $\|\lambda\|$
NEP_SMALLEST_MAGNITUDE	-nep_smallest_magnitude	Smallest $\|\lambda\|$
NEP_LARGEST_REAL	-nep_largest_real	Largest $\mathrm{Re}(\lambda)$
NEP_SMALLEST_REAL	-nep_smallest_real	Smallest $\mathrm{Re}(\lambda)$
NEP_LARGEST_IMAGINARY	-nep_largest_imaginary	Largest $\mathrm{Im}(\lambda)$
NEP_SMALLEST_IMAGINARY	-nep_smallest_imaginary	Smallest $\mathrm{Im}(\lambda)$
NEP_TARGET_MAGNITUDE	-nep_target_magnitude	Smallest $\|\lambda-\tau\|$
NEP_TARGET_REAL	-nep_target_real	Smallest $\|\mathrm{Re}(\lambda-\tau)\|$
NEP_TARGET_IMAGINARY	-nep_target_imaginary	Smallest $\|\mathrm{Im}(\lambda-\tau)\|$
NEP_ALL	-nep_all	All $\lambda\in\Omega$
NEP_WHICH_USER	-nep_which_user	*user-defined*

For the selection of the portion of the spectrum of interest, there are several alternatives listed in table *Available possibilities for selection of the eigenvalues of interest in NEP* (page 75), to be selected with the function:

```
NEPSetWhichEigenpairs(NEP nep,NEPWhich which);
```

The default is to compute the largest magnitude eigenvalues. For the sorting criteria relative to a target value, τ must be specified with `NEPSetTarget()` or in the command-line with `-nep_target`.

`NEP` solvers can also work with a region of the complex plane (`RG`), as discussed in section *Specifying a Region for Filtering* (page 28) for linear problems. Some eigensolvers (NLEIGS) use the definition of the region to compute `nev` eigenvalues in its interior. If *all* eigenvalues inside the region are required, then a contour-integral method is required, see discussion in [Maeda *et al.*, 2016].

Left Eigenvectors

As in the case of linear eigensolvers, some `NEP` solvers have two-sided variants to compute also left eigenvectors. In the case of `NEP`, left eigenvectors are defined as

$$y^*T(\lambda) = 0^*, \qquad y \neq 0. \tag{6.6}$$

Two-sided variants can be selected with:

```
NEPSetTwoSided(NEP eps,PetscBool twosided);
```

6.2.2 Expressing the NEP in Split Form

Instead of implementing callback functions for $T(\lambda)$ and $T'(\lambda)$, a usually simpler alternative is to use the split form of the nonlinear eigenproblem, equation (6.4). Note that in split form, we have $T'(\lambda) = \sum_{i=0}^{\ell-1} A_i f_i'(\lambda)$, so the derivatives of $f_i(\lambda)$ are also required. As described below, we will represent each of the analytic functions f_i by means of an auxiliary object `FN` that holds both the function and its derivative.

Listing 2: Example code for defining the `NEP` eigenproblem in split form

```
FNCreate(PETSC_COMM_WORLD,&f1);  /* f1 = -lambda */
FNSetType(f1,FNRATIONAL);
coeffs[0] = -1.0; coeffs[1] = 0.0;
FNRationalSetNumerator(f1,2,coeffs);

FNCreate(PETSC_COMM_WORLD,&f2);  /* f2 = 1 */
FNSetType(f2,FNRATIONAL);
coeffs[0] = 1.0;
FNRationalSetNumerator(f2,1,coeffs);

FNCreate(PETSC_COMM_WORLD,&f3);  /* f3 = exp(-tau*lambda) */
FNSetType(f3,FNEXP);
FNSetScale(f3,-tau,1.0);

mats[0] = A;  funs[0] = f2;
mats[1] = Id; funs[1] = f1;
mats[2] = B;  funs[2] = f3;
NEPSetSplitOperator(nep,3,mats,funs,SUBSET_NONZERO_PATTERN);
```

Hence, for the split form representation we must provide ℓ matrices A_i and the corresponding functions $f_i(\lambda)$, by means of:

```
NEPSetSplitOperator(NEP nep,PetscInt l,Mat A[],FN f[],MatStructure str);
```

Here, the MatStructure flag is used to indicate whether all matrices have the same (or subset) nonzero pattern with respect to the first one. Listing *Example code for defining the NEP eigenproblem in split form* (page 76) illustrates this usage with the problem of equation (6.3), where $\ell = 3$ and the matrices are I, A and B (note that in the code we have changed the order for efficiency reasons, since the nonzero pattern of I and B is a subset of A's in this case). Two of the associated functions are polynomials ($-\lambda$ and 1) and the other one is the exponential $e^{-\tau\lambda}$.

Details of how to define the f_i functions by using the `FN` class are provided in section *FN: Mathematical Functions* (page 89).

Note

Using the split form is required in order to be able to use some eigensolvers, in particular, those that project the nonlinear eigenproblem onto a low dimensional subspace and then use a dense nonlinear solver for the projected problem.

Table 2: Problem types considered in NEP

Problem Type	NEPProblemType	Command line key
General	NEP_GENERAL	-nep_general
Rational	NEP_RATIONAL	-nep_rational

When defining the problem in split form, it may also be useful to specify a problem type. For example, if the user knows that all f_i functions are rational, as in equation (6.2), then setting the problem type to NEP_RATIONAL gives a hint to the solver that may simplify the solution process. The problem types currently supported for NEP are listed in table *Problem types considered in NEP* (page 77). When in doubt, use the default problem type (NEP_GENERAL).

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

```
NEPSetProblemType(NEP nep,NEPProblemType type);
```

Currently, the problem type is ignored in most solvers and it is taken into account only in NLEIGS for determining singularities automatically.

6.3 Selecting the Solver

The solution method can be specified procedurally with:

```
NEPSetType(NEP nep,NEPType method);
```

or via the options database command -nep_type followed by the name of the method (see table *Nonlinear eigenvalue solvers available in the NEP module* (page 78)). The methods currently available in NEP are the following:

- Residual inverse iteration (RII), where in each iteration the eigenvector correction is computed as $T(\sigma)^{-1}$ times the residual r.
- Successive linear problems (SLP), where in each iteration a linear eigenvalue problem $T(\tilde{\lambda})\tilde{x} = \mu T'(\tilde{\lambda})\tilde{x}$ is solved for the eigenvalue correction μ.
- Nonlinear Arnoldi, which builds an orthogonal basis V_j of a subspace expanded with the vectors generated by RII, then chooses the approximate eigenpair $(\tilde{\lambda}, \tilde{x})$ such that $\tilde{x} = V_j y$ and $V_j^* T(\tilde{\lambda}) V_j y = 0$.
- NLEIGS, which is based on a (rational) Krylov iteration operating on a companion-type linearization of a rational interpolant of the nonlinear function.
- CISS, a contour-integral solver that allows computing all eigenvalues in a given region.
- Polynomial interpolation, where a polynomial matrix $P(\lambda)$ is built by evaluating $T(\cdot)$ at a few points, then PEP is used to solve the resulting polynomial eigenproblem.

Table 3: Nonlinear eigenvalue solvers available in the NEP module

Method	NEPType	Options Database	Need $T'(\cdot)$	Two-sided
Residual inverse iteration	NEPRII	rii	no	
Successive linear problems	NEPSLP	slp	yes	yes
Nonlinear Arnoldi	NEPNARNOLDI	narnoldi	no	
Rational Krylov (NLEIGS)	NEPNLEIGS	nleigs	no	yes
Contour integral SS	NEPCISS	ciss	yes	
Polynomial interpolation	NEPINTERPOL	interpol	no	

The NEPSLP method performs a linearization that results in a (linear) generalized eigenvalue problem. This is handled by an EPS object created internally. If required, this EPS object can be extracted with the operation:

```
NEPSLPGetEPS(NEP nep,EPS *eps);
```

This allows the application programmer to set any of the EPS options directly within the code. These options can also be set through the command-line, simply by prefixing the EPS options with -nep_slp_.

Similarly, NEPINTERPOL works with a PEP object internally, that can be retrieved by NEPInterpolGetPEP(). Another relevant option of this solver is the degree of the interpolation polynomial, that can be set with:

```
NEPInterpolSetInterpolation(NEP nep,PetscReal tol,PetscInt deg);
```

The polynomial interpolation solver currently uses Chebyshev polynomials of the 1st kind and requires the user to specify an interval of the real line where the eigenvalues must be computed, e.g.

```
$ ./ex22 -nep_type interpol -rg_interval_endpoints 0.1,14.0,-0.1,0.1 -nep_nev 2 -nep_interpol_
↪interpolation_degree 15 -nep_target 1.0
```

Note that the target τ must lie inside the region. For details about specifying a region, see section *RG: Region* (page 90).

Some solvers such as NEPRII and NEPNARNOLDI need a KSP object to handle the solution of linear systems of equations. This KSP and can be retrieved, e.g., with:

```
NEPRIIGetKSP(NEP nep,KSP *ksp);
```

This KSP object is typically used to compute the action of $T(\sigma)^{-1}$ on a given vector. In principle, σ is an approximation of an eigenvalue, but it is usually more efficient to keep this value constant, otherwise the factorization or preconditioner must be recomputed every time since eigensolvers update eigenvalue approximations in each iteration. This behavior can be changed with:

```
NEPRIISetLagPreconditioner(NEP nep,PetscInt lag);
```

Recomputing the preconditioner every 2 iterations, say, will introduce a considerable overhead, but may reduce the number of iterations significantly. Another related comment is that, when using an iterative linear solver, the requested accuracy is adapted as the outer iteration progresses, being the tolerance larger in the first solves. Again, the user can modify this behavior with NEPRIISetConstCorrectionTol(). Both options can also be changed at run time. As an example, consider the following command line:

```
$ ./ex22 -nep_type rii -nep_rii_lag_preconditioner 2 -nep_rii_ksp_type bcgs -nep_rii_pc_type ilu -nep_rii_
↪const_correction_tol 1 -nep_rii_ksp_rtol 1e-3
```

The example uses RII with BiCGStab plus ILU, where the preconditioner is updated every two outer iterations and linear systems are solved up to a tolerance of 10^{-3}.

The NLEIGS solver is most appropriate for problems where $T(\cdot)$ is singular at some known parts of the complex plane, for instance the case that $T(\cdot)$ contains $\sqrt{\lambda}$. To treat this case effectively, the NLEIGS solver requires a discretization of the singularity set, which can be provided by the user in the form of a callback function:

```
NEPNLEIGSSetSingularitiesFunction(NEP nep,NEPNLEIGSSingularitiesFn *fun,void *ctx);
```

Alternatively, if the problem is known to be a rational eigenvalue problem, the user can avoid the computation of singularities by just specifying the problem type with `NEPSetProblemType()`, as explained at the end of the previous section. If none of the above functions is invoked by the user, then the NLEIGS solver attempts to determine the singularities automatically.

6.4 Retrieving the Solution

The procedure for obtaining the computed eigenpairs is similar to previously discussed eigensolvers. After the call to `NEPSolve()`, the computed results are stored internally and a call to `NEPGetConverged()` must be issued to obtain the number of converged solutions. Then calling `NEPGetEigenpair()` repeatedly will retrieve each eigenvalue-eigenvector pair.

```
NEPGetEigenpair(NEP nep,PetscInt j,PetscScalar *kr,PetscScalar *ki,Vec xr,Vec xi);
```

In two-sided solvers (see last column of table *Nonlinear eigenvalue solvers available in the NEP module* (page 78)), it is also possible to retrieve left eigenvectors with:

```
NEPGetLeftEigenvector(NEP nep,PetscInt j,Vec yr,Vec yi);
```

> **Warning**
>
> **Real vs complex scalar versions**. The interface makes provision for returning a complex eigenvalue (or eigenvector) when doing the computation in a PETSc/SLEPc version built with real scalars, as is done in other eigensolvers such as `EPS`. However, in some cases this will not be possible. In particular, when callback functions are used and a complex eigenvalue approximation is hit, the solver will fail unless configured with complex scalars. The reason is that the definitions of callback functions only have a single PetscScalar `lambda` argument and hence cannot handle complex arguments in real arithmetic.

The following function:

```
NEPComputeError(NEP nep,PetscInt j,NEPErrorType type,PetscReal *error);
```

can be used to assess the accuracy of the computed solutions. The error is based on the 2-norm of the residual vector r defined in equation (6.5).

As in the case of `EPS`, in `NEP` the number of iterations carried out by the solver can be determined with `NEPGetIterationNumber()`, and the tolerance and maximum number of iterations can be set with `NEPSetTolerances()`. Also, convergence can be monitored with either textual monitors `-nep_monitor`, `-nep_monitor_all`, `-nep_monitor_conv`, or graphical monitors `-nep_monitor draw::draw_lg`, `-nep_monitor_all draw::draw_lg`. See section *Controlling and Monitoring Convergence* (page 23) for additional details. Similarly, there is support for viewing the computed solution as explained in section *Viewing the Solution* (page 26).

The `NEP` class also provides some kind of iterative refinement, similar to the one available in `PEP`, see section *Iterative Refinement* (page 70). The parameters can be set with:

```
NEPSetRefine(NEP nep,NEPRefine refine,PetscInt npart,PetscReal tol,PetscInt its,NEPRefineScheme scheme);
```

Chapter 7

MFN: Matrix Function

The Matrix Function (`MFN`) solver object provides algorithms that compute the action of a matrix function on a given vector, without evaluating the matrix function itself. This is not an eigenvalue problem, but there is a connection between matrix functions and some strategies for computing eigenvalues, and that is why we have this functionality in SLEPc.

7.1 The Problem $f(A)v$

The need to evaluate a function $f(A) \in \mathbb{C}^{n\times n}$ of a matrix $A \in \mathbb{C}^{n\times n}$ arises in many applications. There are many methods to compute matrix functions, see for instance the survey by Higham and Al-Mohy [2010]. Here, we focus on the case that A is large and sparse, or is available only as a matrix-vector product subroutine. In such cases, it is the action of $f(A)$ on a vector, $f(A)v$, that is required and not $f(A)$. For this, it is possible to adapt some of the methods used to approximate eigenvalues, such as those based on Krylov subspaces or on the concept of contour integral. The description below will be restricted to the case of Krylov methods.

In the sequel, we concentrate on the exponential function, which is one of the most demanded in applications, although the concepts are easily generalizable to other functions as well. Using the Taylor series expansion of e^A, we have

$$y = e^A v = v + \frac{A}{1!}v + \frac{A^2}{2!}v + \cdots , \tag{7.1}$$

so, in principle, the vector y can be approximated by an element of the Krylov subspace $\mathcal{K}_m(A, v)$ defined in equation (2.5). This is the basis of the method implemented in Expokit [Sidje, 1998]. Let $AV_m = V_{m+1}\underline{H}_m$ be an Arnoldi decomposition, where the columns of V_m form an orthogonal basis of the Krylov subspace, then the approximation can be computed as

$$\tilde{y} = \beta V_m \exp(H_m)e_1, \tag{7.2}$$

where $\beta = \|v\|_2$ and e_1 is the first coordinate vector. Hence, the problem of computing the exponential of a large matrix A of order n is reduced to computing the exponential of a small matrix H_m of order m. For the latter task, we employ algorithms implemented in the `FN` auxiliary class, see section *FN: Mathematical Functions* (page 89).

7.2 Basic Usage

The user interface of the `MFN` package is simpler than the interface of eigensolvers. In some ways, it is more similar to KSP, in the sense that the solver maps a vector v to a vector y.

Listing 1: Example code for basic solution with MFN

```
MFN         mfn;       /*  MFN solver context                  */
Mat         A;         /*  problem matrix                      */
FN          f;         /*  the function, exp() in this example */
PetscScalar alpha;     /*  to compute exp(alpha*A)             */
Vec         v, y;      /*  right vector and solution           */

MFNCreate(PETSC_COMM_WORLD, &mfn);
MFNSetOperator(mfn, A);
MFNGetFN(mfn, &f);
FNSetType(f, FNEXP);
FNSetScale(f, alpha, 1.0);
MFNSetFromOptions(mfn);
MFNSolve(mfn, v, y);
MFNDestroy(&mfn);
```

Listing *Example code for basic solution with MFN* (page 82) shows a simple example with the basic steps for computing $y = \exp(\alpha A)v$. After creating the solver context with `MFNCreate()`, the problem matrix has to be passed with `MFNSetOperator()` and the function to compute $f(\cdot)$ must be specified with the aid of the auxiliary class `FN`, see details in section *FN: Mathematical Functions* (page 89). Then, a call to `MFNSolve()` runs the solver on a given vector v, returning the computed result y. Finally, `MFNDestroy()` is used to reclaim memory. We give a few more details below.

7.2.1 Defining the Problem

Defining the problem consists in specifying the matrix, A, and the function to compute, $f(\cdot)$. The problem matrix is provided with:

```
MFNSetOperator(MFN mfn,Mat A);
```

where `A` should be a square matrix, stored in any allowed PETSc format including the matrix-free mechanism (see section *Supported Matrix Types* (page 96)). The function $f(\cdot)$ is defined with an `FN` object. One possibility is to extract the `FN` object handled internally by `MFN`:

```
MFNGetFN(MFN mfn,FN *f);
```

An alternative would be to create a standalone `FN` object and pass it with `MFNSetFN()`. In any case, the function is defined via its type and the relevant parameters, see section *FN: Mathematical Functions* (page 89) for details. The scaling parameters can be used for instance for the exponential when used in the context of ODE integration, $y = e^{tA}v$, where t represents the elapsed time.

Note

Some `MFN` solvers may be restricted to only some types of `FN` functions.

In `MFN` it makes no sense to specify the number of eigenvalues. However, there is a related operation that allows the user to specify the size of the subspace that will be used internally by the solver (`ncv`, the number of column vectors of the basis):

```
MFNSetDimensions(EPS eps,PetscInt ncv);
```

This parameter can also be set at run time with the option `-mfn_ncv`.

7.2.2 Selecting the Solver

Table 1: List of solvers available in the MFN module.

Method	MFNType	Options Database	Supported Functions
Restarted Krylov solver	MFNKRYLOV	krylov	Any
Expokit algorithm	MFNEXPOKIT	expokit	Exponential

The methods available in MFN are shown in table *List of solvers available in the MFN module.* (page 83). The solution method can be specified procedurally with:

```
MFNSetType(MFN mfn,MFNType method);
```

or via the options database command -mfn_type followed by the method name (see table *List of solvers available in the MFN module.* (page 83)).

Currently implemented methods are:

- A Krylov method with restarts as proposed by Eiermann and Ernst [2006].
- The method implemented in Expokit [Sidje, 1998] for the matrix exponential.

7.2.3 Accuracy and Monitors

In the $f(A)v$ problem, there is no clear definition of residual, as opposed to the case of linear systems or eigenproblems. Still, the solvers have different ways of assessing the accuracy of the computed solution. The user can provide a tolerance and maximum number of iterations with MFNSetTolerances(), but there is no guarantee that an analog of the residual is below the tolerance.

After the solver has finished, the number of performed (outer) iterations can be obtained with MFNGetIterationNumber(). There are also monitors that display the error estimate, which can be activated with command-line keys -mfn_monitor, or -mfn_monitor draw::draw_lg. See section *Controlling and Monitoring Convergence* (page 23) for additional details.

Chapter 8

LME: Linear Matrix Equation

The Linear Matrix Equation (`LME`) solver object encapsulates functionality intended for the solution of linear matrix equations such as Lyapunov, Sylvester, and their various generalizations. A matrix equation is an equation where the unknown and the coefficients are all matrices, and it is linear if there are no nonlinear terms involving the unknown. This excludes some types of equations such as the Algebraic Riccati Equation or other quadratic matrix equations, which are not considered here.

The general form of a linear matrix equation is

$$AXE + DXB = C, \tag{8.1}$$

where X is the solution.

Since SLEPc is mainly concerned with iterative methods, so is the `LME` module. This implies that `LME` can address only the case where the solution X has low rank. In many situations, X does not have low rank, which means that the `LME` solver, if it converges, will truncate the solution to an artificially small rank, with a large approximation error.

> **Warning**
>
> The `LME` module should be considered experimental. As explained below, the currently implemented functionality is quite limited, and may be extended in the future.

8.1 Current Functionality

The user interface of the `LME` module is prepared for different types of equations, see a list in table *Problem types considered in LME* (page 85). However, currently there is only basic support for continuous-time Lyapunov equations, and the rest are just a wish list.

Table 1: Problem types considered in `LME`

Problem Type	Equation	`LMEProblemType`	A	B	D	E
Continuous-Time Lyapunov	$AX + XA^* = -C$	`LME_LYAPUNOV`	yes	A^*	-	-
Sylvester	$AX + XB = C$	`LME_SYLVESTER`	yes	yes	-	-
Generalized Lyapunov	$AXD^* + DXA^* = -C$	`LME_GEN_LYAPUNOV`	yes	A^*	yes	D^*
Generalized Sylvester	$AXE + DXB = C$	`LME_GEN_SYLVESTER`	yes	yes	yes	yes
Discrete-Time Lyapunov	$AXA^* - X = -C$	`LME_DT_LYAPUNOV`	yes	-	-	A^*
Stein	$AXE - X = -C$	`LME_STEIN`	yes	-	-	yes

8.1.1 Continuous-Time Lyapunov Equation

Given two matrices $A, C \in \mathbb{C}^{n\times n}$, with C Hermitian positive-definite, the continuous-time Lyapunov equation, assuming that the right-hand side matrix C has low rank, can be expressed as

$$AX + XA^* = -C_1C_1^*, \tag{8.2}$$

where we have written the right-hand side in factored form, $C = C_1C_1^*$, $C_1 \in \mathbb{C}^{n\times r}$ with $r = \text{rank}(C)$. In general, X will not have low rank, but our intention is to compute a low-rank approximation $\tilde{X} = X_1X_1^*$, $X_1 \in \mathbb{C}^{n\times p}$ for a given value p. Note that the equation has a unique solution if and only if A is stable, i.e., all of its eigenvalues have strictly negative real part.

The reason why SLEPc provides a solver for equation (8.2) is that it is required in the eigensolver `EPSLYAPII`, which implements the Lyapunov inverse iteration [Elman and Wu, 2013, Meerbergen and Spence, 2010].

> **Note**
>
> The `LME` solvers currently implemented in SLEPc are very basic. Users interested in applying `LME` to matrix equations appearing in their applications are encouraged to contact SLEPc developers describing their use case.

8.1.2 Krylov Solver

Currently, the solver for the continuous-time Lyapunov equation available in SLEPc is based on the following procedure for each column c of C_1:

1. Build an *Arnoldi factorization* for the Krylov subspace $\mathcal{K}_m(A, c)$, $AV_m = V_mH_m + h_{m+1,m}v_{m+1}e_m^*$.
2. Solve the *compressed Lyapunov equation*, $H_mY + YH_m^* = -\tilde{c}\,\tilde{c}^* \quad \text{with} \quad \tilde{c} = V_m^*c$.
3. Set the approximate solution to $X_m = V_mYV_m^*$.

Furthermore, our Krylov solver incorporates an Eiermann-Ernst-type restart as proposed by Kressner [2008]. The restart will discard the Arnoldi basis V_m but keep H_m, then continue the Arnoldi recurrence from v_{m+1}. The upper Hessenberg matrices generated at each restart are glued together. This implies that at each restart the size of the compressed Lyapunov equation grows. After convergence, once the full Y is available, we perform a second pass to reconstruct the successive V_m bases.

The restarted algorithm is as follows:

for $k = 1, 2, \ldots$

- Run Arnoldi $AV_m^{(k)} = V_m^{(k)}H_m^{(k)} + h_{m+1,m}^{(k)}v_{m+1}^{(k)}e_m^*$.
- Set $H_{km} = \begin{bmatrix} H_{(k-1)m} & 0 \\ h_{m+1,m}^{(k-1)}e_1e_{(k-1)m}^* & H_m^{(k)} \end{bmatrix}$.
- Solve compressed equation $H_{km}Y + YH_{km}^* = -\|c\|_2^2\, e_1e_1^*$.
- Compute residual norm $\|R_k\|_2 = h_{m+1,m}^{(k)}\|e_{km}^*Y\|_2$.

end

The residual norm is used for the stopping criterion.

8.1.3 Projected Problem

As mentioned in the previous section, the projected problem is a smaller Lyapunov equation, which must be solved in each outer iteration of the method (restart). The solution of this small dense matrix equation is also implemented within the `LME` module (there is no auxiliary class for this).

If A is stable, then X is symmetric positive semidefinite. Hence, looking at step 3 of the algorithm in the previous section, it is better to compute the Cholesky factor $Y = LL^*$ to obtain the low rank factor $X_1 = V_mL$. The method of Hammarling will compute L directly, without computing Y first.

If SLEPc has been configured with the option `--download-slicot` (see *Wrappers to External Libraries* (page 99)), it will be possible to use SLICOT subroutines to apply Hammarling's method, otherwise a more basic algorithm will be used.

> **Note**
>
> Most of the SLICOT subroutines are implemented for real arithmetic only, so it is not possible to use them in a complex build of PETSc/SLEPc.

In the restarted algorithm described above, in the second pass to recalculate the successive V_m's, the update is based on the truncated SVD of the Cholesky factor

$$L = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} Z_1 & Z_2 \end{bmatrix}^* \quad \text{with} \quad \|\Sigma_2\|_2 < \varepsilon,$$

so that the final X_1 has the appropriate rank.

8.2 Basic Usage

Once the solver object has been created with `LMECreate()`, the user has to define the matrix equation to be solved. First, the type of equation must be set with `LMESetProblemType()`, choosing one from table *Problem types considered in LME* (page 85).

The coefficient matrices A, B, D, E must be provided via `LMESetCoefficients()`, but some of them are optional depending on the matrix equation. For Lyapunov equations, only A must be set, which is normally a large, sparse matrix stored in MATAIJ format. Note that in table *Problem types considered in LME* (page 85), the notation A^* means that this matrix need not be passed, but the user may choose to pass an explicit transpose of matrix A (for improved efficiency). Also note that some of the equation types impose restrictions on the properties of the coefficient matrices (e.g., A stable in Lyapunov equations) and possibly on the right-hand side C.

To conclude the definition of the equation, we must pass the right-hand side C with `LMESetRHS()`. Note that C is expressed as the outer product of a tall-skinny matrix, $C = C_1 C_1^*$. This can be represented in PETSc using a special type of matrix, MATLRC, that can be created with MatCreateLRC passing a dense matrix C_1.

The call to `LMESolve()` will run the solver to compute the solution X. The rank of X may be prescribed by the user or selected dynamically by the solver. Next, we discuss these two options:

- To prescribe a fixed rank for X, we must preallocate it, creating a MATLRC matrix, and then pass it via `LMESetSolution()` *before* calling `LMESolve()`. In this way, the solver will be restricted to a rank equal to the number of columns provided in X_1.
- If we do not call `LMESetSolution()` beforehand, the solver will select the rank dynamically. Then after `LMESolve()` we must call `LMEGetSolution()` to retrieve a MATLRC matrix representing X.

Chapter 9

Auxiliary Classes

Apart from the main solver classes listed in table *SLEPc modules* (page i), SLEPc contains several auxiliary classes:

- `ST`: Spectral Transformation, fully described in chapter *ST: Spectral Transformation* (page 35).
- `FN`: Mathematical Function, required in application code to represent the constituent functions of the nonlinear operator in split form (chapter *NEP: Nonlinear Eigenvalue Problems* (page 73)), as well as the function to be used when computing the action of a matrix function on a vector (chapter *MFN: Matrix Function* (page 81)).
- `RG`: Region, a way to define a region of the complex plane.
- `DS`: Direct Solver (or Dense System), can be seen as a wrapper to LAPACK functions used within SLEPc.
- `BV`: Basis Vectors, provides the concept of a block of vectors that represent the basis of a subspace.

The first three classes, `ST`, `FN` and `RG`, are relevant for end users, while `DS` and `BV` are intended mainly for SLEPc developers. Below we provide a brief description of `FN`, `RG`, `DS` and `BV`.

9.1 FN: Mathematical Functions

The `FN` class provides a few predefined mathematical functions, including rational functions (of which polynomials are a particular case) and exponentials. Objects of this class are instantiated by providing the values of the relevant parameters. `FN` objects are created with `FNCreate()` and it is necessary to select the type of function (rational, exponential, etc.) with `FNSetType()`. Table *Mathematical functions available as FN objects* (page 89) lists available functions.

Table 1: Mathematical functions available as `FN` objects

Function	`FNType`	Expression
Polynomial and rational	`FNRATIONAL`	$p(x)/q(x)$
Exponential	`FNEXP`	e^x
Logarithm	`FNLOG`	$\log x$
φ-functions	`FNPHI`	$\varphi_0(x)$, $\varphi_1(x)$, ...
Square root	`FNSQRT`	$\sqrt{x}$
Inverse square root	`FNINVSQRT`	$x^{-\frac{1}{2}}$
Combine two functions	`FNCOMBINE`	See text

Parameters common to all `FN` types are the scaling factors, which are set with:

```
FNSetScale(FN fn,PetscScalar alpha,PetscScalar beta);
```

where `alpha` multiplies the argument and `beta` multiplies the result. With this, the actual function is $\beta \cdot f(\alpha \cdot x)$ for a given function $f(\cdot)$. For instance, an exponential function $f(x) = e^x$ will turn into

$$g(x) = \beta e^{\alpha x}. \tag{9.1}$$

In a rational function there are specific parameters, namely the coefficients of the numerator and denominator,

$$r(x) = \frac{p(x)}{q(x)} = \frac{\nu_{n-1}x^{n-1} + \cdots + \nu_1 x + \nu_0}{\delta_{m-1}x^{m-1} + \cdots + \delta_1 x + \delta_0}. \tag{9.2}$$

These parameters are specified with:

```
FNRationalSetNumerator(FN fn,PetscInt np,PetscScalar pcoeff[]);
FNRationalSetDenominator(FN fn,PetscInt nq,PetscScalar qcoeff[]);
```

Here, polynomials are passed as an array with high order coefficients appearing in low indices.

The φ-functions are given by

$$\varphi_0(x) = e^x, \qquad \varphi_1(x) = \frac{e^x - 1}{x}, \qquad \varphi_k(x) = \frac{\varphi_{k-1}(x) - 1/(k-1)!}{x}, \tag{9.3}$$

where the index k must be specified with `FNPhiSetIndex()`.

Whenever the solvers need to compute $f(x)$ or $f'(x)$ on a given scalar x, the following functions are invoked:

```
FNEvaluateFunction(FN fn,PetscScalar x,PetscScalar *y)
FNEvaluateDerivative(FN fn,PetscScalar x,PetscScalar *y)
```

The function can also be evaluated as a matrix function, $B = f(A)$, where A, B are small, dense, square matrices. This is done with `FNEvaluateFunctionMat()`. Note that for a rational function, the corresponding expression would be $q(A)^{-1}p(A)$. For computing functions such as the exponential of a small matrix A, several methods are available. When the matrix A is symmetric, the default is to compute $f(A)$ using the eigendecomposition $A = Q\Lambda Q^*$, for instance the exponential would be computed as $\exp(A) = Q\,\mathrm{diag}(e^{\lambda_i})Q^*$. In the general case, it is necessary to have recourse to one of the methods discussed in, e.g., [Higham and Al-Mohy, 2010].

Finally, there is a mechanism to combine simple functions in order to create more complicated functions. For instance, the function

$$f(x) = (1 - x^2)\exp\left(\frac{-x}{1 + x^2}\right) \tag{9.4}$$

can be represented with an expression tree with three leaves (one exponential function and two rational functions) and two interior nodes (one of them is the root, $f(x)$). Interior nodes are simply `FN` objects of type `FNCOMBINE` that specify how the two children must be combined (with either addition, multiplication, division or function composition):

```
FNCombineSetChildren(FN fn,FNCombineType comb,FN f1,FN f2)
```

The combination of f_1 and f_2 with division will result in $f_1(x)/f_2(x)$ and $f_2(A)^{-1}f_1(A)$ in the case of matrices.

9.2 RG: Region

The `RG` object defines a region of the complex plane, that can be used to specify where eigenvalues must be sought. Currently, the following types of regions are available:

- A (generalized) interval, defined as $[a, b] \times [c, d]$, where the four parameters can be set with `RGIntervalSetEndpoints()`. This covers the particular cases of an interval on the real axis (setting $c = d = 0$), the left halfplane $[-\infty, 0] \times [-\infty, +\infty]$, a quadrant, etc. See figure *Interval region defined via de RG class* (page 91).

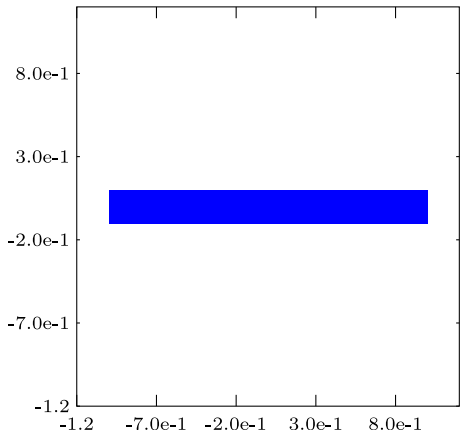


Fig. 1: Interval region defined via de RG class

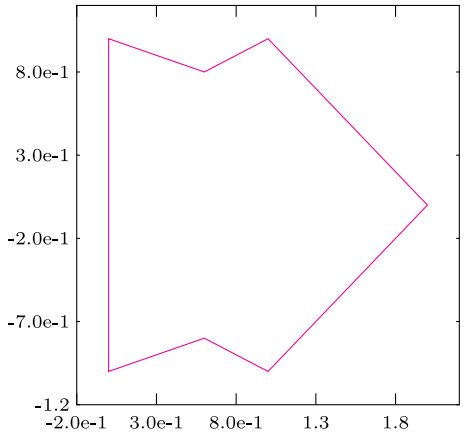


Fig. 2: Polygon region defined via de RG class

- A polygon defined by its vertices, given via `RGPolygonSetVertices()`. See figure *Polygon region defined via de RG class* (page 91).
- An ellipse defined by its center, radius and vertical scale (1 by default), specified with `RGEllipseSetParameters()`. See figure *Ellipse region defined via de RG class* (page 92).

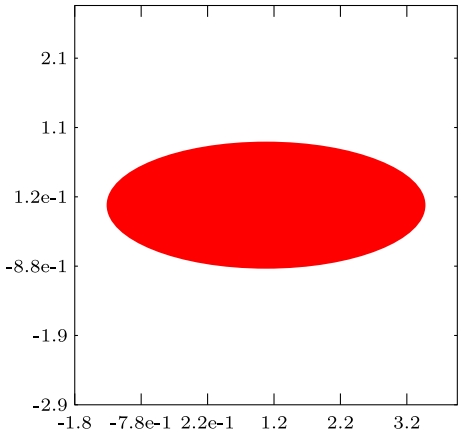


Fig. 3: Ellipse region defined via de RG class

- A ring region similar to an ellipse but consisting of a thin stripe along the ellipse with optional start and end angles. See figure *Ring region defined via de RG class* (page 92). The parameters are set with `RGRingSetParameters()`.

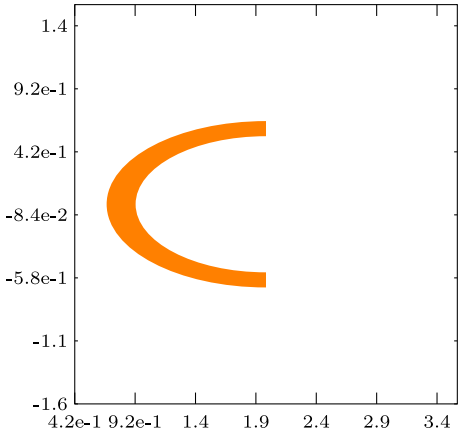


Fig. 4: Ring region defined via de RG class

Check table *Regions available as RG objects* (page 92) for the names that should be used in each case.

Table 2: Regions available as RG objects

Region Type	RGType	Options Database
(Generalized) Interval	RGINTERVAL	interval
Polygon	RGPOLYGON	polygon
Ellipse	RGELLIPSE	ellipse
Ring	RGRING	ring

Sometimes it is useful to specify the complement of a certain region, e.g., the part of the complex plane outside an ellipse. This can be achieved with:

```
RGSetComplement(RG rg,PetscBool flg)
```

or in the command line with `-rg_complement`.

By default, a newly created RG object that is not set a type nor parameters must represent the whole complex plane (the same as RGINTERVAL with values $[-\infty,+\infty]\times[-\infty,+\infty]$). We call this the *trivial* region, and provide a function to test this situation:

```
RGIsTrivial(RG rg,PetscBool *trivial)
```

Another useful operation is to check whether a given point of the complex plane is inside the region or not:

```
RGCheckInside(RG rg,PetscInt n,PetscScalar ar[],PetscScalar ai[],PetscInt *inside)
```

Note that the point is represented as two PetscScalar's, similarly to eigenvalues in SLEPc.

9.3 DS: Direct Solver (or Dense System)

The DS class can be seen as a wrapper to LAPACK functions [Anderson *et al.*, 1999] used within SLEPc. It is mostly an internal object that need not be called by end users.

Many of the iterative eigensolvers implemented in SLEPc, such as Krylov-Schur or LOBPCG, perform a projection onto a low-dimensional subspace, e.g., $M = V^*AV$, where M is a dense matrix of small dimension (compared to the size of the original problem) and may or may not have a special structure such as Hessenberg or tridiagonal. Computing the full solution of an eigenproblem associated with matrix M is necessary to continue the outer iterative eigensolver, and this is typically done by calling a LAPACK function.

In case of parallel runs with several MPI processes, the result of the projection, M, is available in all processes and all of them solve the dense eigenproblem redundantly. Although this behavior can be changed slightly (see DSSetParallel()), the computation done in DS should be considered a sequential step of the overall algorithm, because we assume the size of M is small. That is why it is relevant for parallel efficiency to keep the size of M bounded, see discussion in *Computing a Large Portion of the Spectrum* (page 28).

Due to the wide range of eigenproblems covered by SLEPc, the projected eigenproblem also has many variants, represented by the DSType.

> **Note**
>
> The projected problem associated with MFN solvers is a matrix function calculation, which is implemented in FN and not DS.

Not all DS types are covered by LAPACK subroutines, particularly DSGHIEP, DSHSVD, DSPEP, and DSNEP. Hence DS provides implementations for solving those problems. In many cases, several methods are available, and this includes the case where LAPACK provides several subroutines for the same problem. The user can select a solution method with an integer set via DSSetMethod().

The user interface is organized in a way that accommodates to the needs of iterative eigensolvers. The computation is split in three stages, DSSolve(), DSSort() and DSVectors(), which are typically called at different times. There are also convenience functions for truncating a previously computed solution, DSTruncate(), or to process the additional row present in M in case of Krylov methods, see DSUpdateExtraRow(), which results in the typical arrowhead form when restarting Lanczos, for instance.

To manipulate matrix M and other matrices associated with the projected problem, the DS interface provides a list of dense matrices, see DSMatType, and operations to work with them, such as DSGetMat() or DSGetArray().

9.4 BV: Basis Vectors

The BV class may be useful for advanced users, so we briefly describe it here for completeness. BV is a convenient way of handling a collection of vectors that often operate together, rather than working with an array of Vec. It can be seen as a generalization of Vec to a tall-skinny matrix with several columns.

Table 3: Operations available for BV objects

Operation	Block version	Column version	Vector version
$Y = X$	BVCopy()	BVCopyColumn()	BVCopyVec()
$Y = \beta Y + \alpha XQ$	BVMult()	BVMultColumn()	BVMultVec()
$M = Y^*AX$	BVMatProject()	–	–
$M = Y^*X$	BVDot()	BVDotColumn()	BVDotVec()
$Y = \alpha Y$	BVScale()	BVScaleColumn()	–
$r = \|X\|_{type}$	BVNorm()	BVNormColumn()	BVNormVec()
Set to random values	BVSetRandom()	BVSetRandomColumn()	–
Orthogonalize	BVOrthogonalize()	BVOrthogonalizeColumn()	BVOrthogonalizeVec()

Table *Operations available for BV objects* (page 94) shows a summary of the operations offered by the BV class, with variants that operate on the whole BV, on a single column, or on an external Vec object. Missing variants can be achieved simply with Vec and Mat operations. Other available variants not shown in the table are BVMultInPlace(), BVMultInPlaceHermitianTranspose() and BVOrthogonalizeSomeColumn().

Most SLEPc solvers use a BV object to represent the working subspace basis. In particular, orthogonalization operations are mostly confined within BV. Hence, BV provides options for specifying the method of orthogonalization of vectors (Gram-Schmidt) as well as the method of block orthogonalization, see BVSetOrthogonalization().

CHAPTER 10

Additional Information

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

10.1 Supported PETSc Features

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

SLEPc mainly contains high level objects, as depicted in figure *Numerical components of PETSc and SLEPc* (page 3). These object classes have been designed and implemented following the philosophy of other high level objects in PETSc. In this way, SLEPc benefits from a number of PETSc's good properties such as the following (see PETSc Users Guide for details):

- Portability and scalability in a wide range of platforms. Different architecture builds can coexist in the same installation. Where available, shared libraries are used to reduce disk space of executable files.
- Support for profiling of programs:
  - Display performance statistics with `-log_view`, including also SLEPc's objects. The collected data are *flops*, memory usage and execution times as well as information about parallel performance, for individual subroutines and the possibility of user-defined stages.
  - Event logging, including user-defined events.
  - Direct wall-clock timing with PetscTime().
  - Display detailed profile information and trace of events.
- Convergence monitoring, both textual and graphical.
- Support for debugging of programs:
  - Debugger startup and attachment of parallel processes.
  - Automatic generation of back-traces of the call stack.
  - Detection of memory leaks.
- A number of viewers for visualization of data, including built-in graphics capabilities that allow for sparse pattern visualization, graphic convergence monitoring, operator's spectrum visualization and display of regions of the complex plane.

- Easy handling of runtime options.
- Support for Fortran programming using Fortran modules. See section *Fortran Interface* (page 102) for details.

10.2 Supported Matrix Types

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

The above statement must be reconsidered when using `EPS` in combination with `ST`. As explained in chapter *ST: Spectral Transformation* (page 35), in many cases the operator associated with 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 *Operators used in each spectral transformation mode* (page 37)). In this case, the limitation is that there must be support for the requested operation for the selected matrix representation.

10.2.1 Shell Matrices

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

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

$$\begin{bmatrix} A & C \\ C^* & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} = \lambda \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} \quad , \tag{10.1}$$

where x and p denote the velocity and pressure fields. Similar formulations also appear in many other situations.

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

All these cases can be easily handled in SLEPc by means of shell matrices. These are matrices that do not require explicit storage of the matrix entries. 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 function MatCreateShell(). Then, the function MatShellSetOperation() is used to provide any user-defined shell matrix operations (see the PETSc Users Guide for additional details). Several examples are available in SLEPc that illustrate how to solve a matrix-free eigenvalue problem.

In the simplest case, defining matrix-vector product operations (`MATOP_MULT`) is enough for using `EPS` with shell matrices. However, in the case of generalized problems, if matrix B is also a shell matrix then it may be necessary to define other operations in order to be able to solve the linear system successfully, for example `MATOP_GET_DIAGONAL` to use an iterative linear solver with 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 *Explicit Computation of the Coefficient Matrix* (page 42) for discussion).

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

10.3 GPU Computing

Support for graphics processing unit (GPU) computing is included in SLEPc. This is related to section *Supported Matrix Types* (page 96) because GPU support in PETSc is based on using special types of Mat and Vec. GPU support in SLEPc has been tested in all solver classes and most solvers should work, although the performance

gain to be expected depends on the particular algorithm. Regarding PETSc, all iterative linear solvers are prepared to run on the GPU, but this is not the case for direct solvers and preconditioners. The user must not expect a spectacular performance boost, but in general moderate gains can be achieved by running the eigensolver on the GPU instead of the CPU (in some cases a 10-fold improvement).

SLEPc currently provides support for NVIDIA GPUs using CUDA[1] as well as AMD GPUs using HIP and ROCm[2]. GPUs can also be used indirectly via Kokkos, see below.

CUDA provides a C/C++ compiler with CUDA extensions as well as the cuBLAS and cuSPARSE libraries that implement dense and sparse linear algebra operations. For instance, to configure PETSc with GPU support in single precision arithmetic use the following options:

```
$ ./configure --with-precision=single --with-cuda
```

VECCUDA and MATAIJCUSPARSE are currently the mechanism in PETSc to run a computation on the GPU. VECCUDA is a special type of Vec whose array is mirrored in the GPU (and similarly for MATAIJCUSPARSE). PETSc takes care of keeping memory coherence between the two copies of the array, and performs the computation on the GPU when possible, trying to avoid unnecessary copies between the host and the device. For maximum efficiency, the user has to make sure that all vectors and matrices are of these types. If they are created in the standard way (VecCreate() plus VecSetFromOptions()) then it is sufficient to run the SLEPc program with

```
$ ./program -vec_type cuda -mat_type aijcusparse
```

Note that the first option is unnecessary if no Vec is created in the main program, or if all vectors are created via MatCreateVecs() from a MATAIJCUSPARSE.

For AMD GPUs the procedure is very similar, with HIP providing the compiler and ROCm providing the analogue libraries hipBLAS and hipSPARSE. To configure PETSc with HIP do:

```
$ ./configure --with-precision=single --with-hip
```

Then the equivalent vector and matrix types are VECHIP and MATAIJHIPSPARSE, which can be used in the command line with

```
$ ./program -vec_type hip -mat_type aijhipsparse
```

An alternative is to enable Kokkos and Kokkos Kernels during configuration of PETSc, and use VECKOKKOS and MATAIJKOKKOS, or in the command line with

```
$ ./program -vec_type kokkos -mat_type aijkokkos
```

This will select the CUDA or HIP backends, if enabled during configuration of PETSc, or use the CPU otherwise.

10.4 Extending SLEPc

Shell matrices, presented in section *Supported Matrix Types* (page 96), are a simple mechanism of extensibility, in the sense that the package is extended with new user-defined matrix objects. Once the new matrix has been defined, it can be used by SLEPc in the same way as the rest of the matrices as long as the required operations are provided.

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

The following function:

[1] https://developer.nvidia.com/cuda-zone
[2] https://rocm.docs.amd.com

```
STShellSetApply(ST st,STShellApplyFn *apply)
```

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 this function:

```
STShellSetBackTransform(ST st,STShellBackTransformFn *backtr)
```

can be used optionally to specify the routine for the back-transformation of eigenvalues. The two functions provided by the user can make use of any required user-defined information via a context that can be retrieved with `STShellGetContext()`. The example program ex10.c illustrates the use of shell transformations.

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

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

After this call, the new solver could be used in the same way as the rest of SLEPc solvers, e.g. with `-eps_type newsubspace` in the command line. A similar mechanism is available for registering new types of the other classes.

10.5 Directory Structure

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

- `lib/slepc/conf` - Directory containing the base SLEPc makefile, to be included in application makefiles.
- `config` - SLEPc configuration scripts.
- `doc` - Directory containing the source from which all documentation of SLEPc is generated, including the manual and the website.
- `include` - All include files for SLEPc. The following subdirectories exist:
  - `slepc/finclude` - include files for Fortran programmers.
  - `slepc/private` - include files containing implementation details, for developer use only.
- `share/slepc` - Common files, including:
  - `datafiles` - data files used by some examples.
- `src` - The source code for all SLEPc components, which currently includes:
  - `sys` - system-related routines and auxiliary classes `bv`, `ds`, `fn`, `rg`, `st`.
  - `eps` - eigenvalue problem solver.
  - `svd` - singular value decomposition solver.
  - `pep` - polynomial eigenvalue problem solver.
  - `nep` - nonlinear eigenvalue problem solver.
  - `mfn` - matrix function.
  - `lme` - linear matrix equations.
  - `binding` - source code of slepc4py
- `$PETSC_ARCH` - For each value of `PETSC_ARCH`, a directory exists containing files generated during installation of that particular configuration. Among others, it includes the following subdirectories:
  - `lib` - all the generated libraries.
  - `lib/slepc/conf` - configuration parameters and log files.

- `include` - automatically generated include files, such as Fortran `*.mod` files.

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 the different implementations.
- `tutorials`: Example programs intended for learning to use SLEPc.
- `tests`: Example programs used by testing scripts.

10.6 Wrappers to External Libraries

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

In order to use SLEPc together with an external library such as ARPACK, one needs to do the following.

1. Install the external software, with the same compilers and MPI that will be used for PETSc/SLEPc.
2. Enable the utilization of the external software from SLEPc by specifying configure options as explained in section *Configuration Options* (page 4).
3. Build the SLEPc libraries.
4. Use the runtime option `-eps_type <type>` to select the solver.

Exceptions to the above rule are LAPACK, which should be enabled during PETSc's configuration, and BLOPEX, that must be installed with `--download-blopex` in SLEPc's configure. Other packages also support the download option.

10.6.1 List of External Libraries

> **Warning**
>
> This list might be incomplete. Check the output of `./configure --help` for other libraries not listed here.

LAPACK

https://www.netlib.org/lapack

LAPACK [Anderson *et al.*, 1999] is a software package for the solution of many different dense linear algebra problems, including various types of eigenvalue problems and singular value decompositions. SLEPc explicitly creates the operator matrix in dense form and then the appropriate LAPACK driver routine is invoked. Therefore, this interface should be used only for testing and validation purposes and not in a production code. The operator matrix is created by applying the operator to the columns of the identity matrix.

Installation: PETSc already depends on LAPACK. The SLEPc interface to LAPACK can be used directly. If SLEPc's `configure` script complains about missing LAPACK functions, then reconfigure PETSc with option `--download-f2cblaslapack`.

ARPACK

https://www.arpack.org

https://github.com/opencollab/arpack-ng

ARPACK [Lehoucq *et al.*, 1998, Maschhoff and Sorensen, 1996] 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 matrices. 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: To install ARPACK we recommend using the *arpack-ng* distribution, available in `github.com`, that supports `configure+make` for installation. Also, SLEPc's `configure` allows to download this version automatically via the `--download-arpack` option. It is also possible to configure SLEPc with the serial version of ARPACK. For this, you have to configure PETSc with the option `--with-mpi=0`.

PRIMME

https://www.cs.wm.edu/~andreas/software

PRIMME [Stathopoulos and McCombs, 2010] is a C library for finding a number of eigenvalues and their corresponding eigenvectors of a real symmetric (or complex Hermitian) matrix. This library provides a multimethod eigensolver, based on Davidson/Jacobi-Davidson. Particular methods include GD+1, JDQMR, and LOBPCG. It supports preconditioning as well as the computation of interior eigenvalues.

Installation: Type `make lib` after customizing the file `Make_flags` appropriately. Alternatively, the `--download-primme` option is also available in SLEPc's `configure`.

Specific options: Since PRIMME contains preconditioned solvers, the SLEPc interface uses `STPRECOND`, as described in section *Preconditioner* (page 39). The SLEPc interface to this package allows the user to specify the maximum allowed block size with the function `EPSPRIMMESetBlockSize()` or at run time with the option `-eps_primme_blocksize <size>`. For changing the particular algorithm within PRIMME, use the function `EPSPRIMMESetMethod()`. PRIMME also provides a solver for the singular value decomposition that is interfaced in SLEPc's `SVD`, see `SVDPRIMMESetMethod()`.

EVSL

https://www-users.cse.umn.edu/~saad/software/EVSL/

EVSL [Li *et al.*, 2019] is a sequential library that implements methods for computing all eigenvalues located in a given interval for real symmetric (standard or generalized) eigenvalue problems. Currently SLEPc only supports standard problems.

Installation: The option `--download-evsl` is available in SLEPc's `configure` for easy installation. Alternatively, one can use an already installed version.

BLOPEX

https://github.com/lobpcg/blopex

BLOPEX [Knyazev *et al.*, 2007] is a package that implements the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method for computing several extreme eigenpairs of symmetric positive generalized eigenproblems. Numerical comparisons suggest that this method is a genuine analog for eigenproblems of the standard preconditioned conjugate gradient method for symmetric linear systems.

Installation: In order to use BLOPEX from SLEPc, it necessary to install it during SLEPc's configuration: `./configure --download-blopex`.

Specific options: Since BLOPEX contains preconditioned solvers, the SLEPc interface uses `STPRECOND`, as described in section *Preconditioner* (page 39).

ScaLAPACK

https://www.netlib.org/scalapack

ScaLAPACK [Blackford *et al.*, 1997] is a library of high-performance linear algebra routines for parallel distributed memory machines. It contains eigensolvers for dense Hermitian eigenvalue problems, as well as solvers for the (dense) SVD.

Installation: For using ScaLAPACK from SLEPc it is necessary to select it during configuration of PETSc.

ELPA

https://elpa.mpcdf.mpg.de/

ELPA [Auckenthaler *et al.*, 2011] is a high-performance library for the parallel solution of dense symmetric (or Hermitian) eigenvalue problems on distributed memory computers. It uses a ScaLAPACK-compatible matrix distribution.

Installation: The SLEPc wrapper to ELPA can be activated at configure time with the option `--download_elpa`, in which case ScaLAPACK support must have been enabled during the configuration of PETSc.

KSVD

https://github.com/ecrc/ksvd/

KSVD [Sukkari *et al.*, 2019] is a high performance software framework for computing a dense SVD on distributed-memory manycore systems. The KSVD solver relies on the polar decomposition (PD) based on the QR Dynamically-Weighted Halley (QDWH) and ZOLO-PD algorithms.

Installation: The option `--download-ksvd` is available in SLEPc's `configure` for easy installation, which in turn requires adding `--download-polar` and `--download-elpa`.

ELEMENTAL

https://github.com/elemental/Elemental

ELEMENTAL [Poulson *et al.*, 2013] is a distributed-memory, dense and sparse-direct linear algebra package. It contains eigensolvers for dense Hermitian eigenvalue problems, as well as solvers for the SVD.

Installation: For using ELEMENTAL from SLEPc it is necessary to select it during configuration of PETSc.

FEAST

https://feast-solver.org/

FEAST [Polizzi, 2009] is a numerical library for solving the standard or generalized symmetric eigenvalue problem, and obtaining all the eigenvalues and eigenvectors within a given search interval. It is based on an innovative fast and stable numerical algorithm which deviates fundamentally from the traditional Krylov subspace based iterations or Davidson-Jacobi techniques. The FEAST algorithm takes its inspiration from the density-matrix representation and contour integration technique in quantum mechanics. Latest versions also support non-symmetric problems.

Installation: We only support the FEAST implementation included in Intel MKL. For using it from SLEPc it is necessary to configure PETSc with MKL by adding the corresponding option, e.g., `--with-blas-lapack-dir=$MKLROOT`.

Specific options: The SLEPc interface to FEAST allows the user to specify the number of contour integration points with the function `EPSFEASTSetNumPoints()` or at run time with the option `-eps_feast_num_points <n>`.

CHASE

https://github.com/ChASE-library/ChASE

CHASE [Winkelmann *et al.*, 2019] is a modern and scalable library based on subspace iteration with polynomial acceleration to solve dense Hermitian (symmetric) algebraic eigenvalue problems, especially solving dense Hermitian eigenproblems arranged in a sequence. Novel to ChASE is the computation of the spectral estimates that enter in the filter and an optimization of the polynomial degree that further reduces the necessary floating-point operations.

Installation: Currently, the CHASE interface in SLEPc is based on the MPI version with block-cyclic distribution, i.e., ScaLAPACK matrix storage, so it is necessary to enable ScaLAPACK during configuration of PETSc.

SLICOT

https://www.slicot.org

SLICOT provides Fortran 77 implementations of numerical algorithms for computations in systems and control theory. In SLEPc, they are used in the `LME` module only, for solving Lyapunov equations of small size.

Installation: The option `--download-slicot` is available in SLEPc's `configure` for easy installation.

10.7 Fortran Interface

SLEPc provides an interface for Fortran 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.

A detailed discussion can be found in the PETSc Users Guide.

The following is a Fortran example. It is the Fortran equivalent of the program given in section *Simple SLEPc Example* (page 8) and can be found in `${SLEPC_DIR}/src/eps/tutorials` (file `ex1f.F90`).

```
#include <slepc/finclude/slepceps.h>
program ex1f
  use slepceps
  implicit none

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!     Declarations
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
  Mat            :: A    ! operator matrix
  EPS            :: eps  ! eigenproblem solver context
  EPSType        :: tname
  PetscInt       :: n, i, Istart, Iend
  PetscInt       :: nev
  PetscInt       :: row(1), col(3)
  PetscMPIInt    :: rank
  PetscErrorCode :: ierr
  PetscBool      :: flg, terse
  PetscScalar    :: val(3)

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

  PetscCallA(SlepcInitialize(PETSC_NULL_CHARACTER, "ex1f test"//c_new_line, ierr))
  PetscCallMPIA(MPI_Comm_rank(PETSC_COMM_WORLD, rank, ierr))
  n = 30
  PetscCallA(PetscOptionsGetInt(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-n', n, flg, ierr))

  if (rank == 0) then
    write (*, '(/a,i4,a)') '1-D Laplacian Eigenproblem, n =', n, ' (Fortran)'
  end if

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

  PetscCallA(MatCreate(PETSC_COMM_WORLD, A, ierr))
  PetscCallA(MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, n, n, ierr))
  PetscCallA(MatSetFromOptions(A, ierr))

  PetscCallA(MatGetOwnershipRange(A, Istart, Iend, ierr))
  if (Istart == 0) then
    row(1) = 0
    col(1) = 0
    col(2) = 1
    val(1) = 2.0
    val(2) = -1.0
```

(continues on next page)

(continued from previous page)

```
    PetscCallA(MatSetValues(A, 1_PETSC_INT_KIND, row, 2_PETSC_INT_KIND, col, val, INSERT_VALUES, ierr))
    Istart = Istart + 1
  end if
  if (Iend == n) then
    row(1) = n - 1
    col(1) = n - 2
    col(2) = n - 1
    val(1) = -1.0
    val(2) = 2.0
    PetscCallA(MatSetValues(A, 1_PETSC_INT_KIND, row, 2_PETSC_INT_KIND, col, val, INSERT_VALUES, ierr))
    Iend = Iend - 1
  end if
  val(1) = -1.0
  val(2) = 2.0
  val(3) = -1.0
  do i = Istart, Iend - 1
    row(1) = i
    col(1) = i - 1
    col(2) = i
    col(3) = i + 1
    PetscCallA(MatSetValues(A, 1_PETSC_INT_KIND, row, 3_PETSC_INT_KIND, col, val, INSERT_VALUES, ierr))
  end do

  PetscCallA(MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY, ierr))
  PetscCallA(MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY, ierr))

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! Create the eigensolver and display info
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

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

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

! ** Set solver parameters at runtime
  PetscCallA(EPSSetFromOptions(eps, ierr))

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

  PetscCallA(EPSSolve(eps, ierr))

! ** Optional: Get some information from the solver and display it
  PetscCallA(EPSGetType(eps, tname, ierr))
  if (rank == 0) then
    write (*, '(a,a)') ' Solution method: ', tname
  end if
  PetscCallA(EPSGetDimensions(eps, nev, PETSC_NULL_INTEGER, PETSC_NULL_INTEGER, ierr))
  if (rank == 0) then
    write (*, '(a,i4)') ' Number of requested eigenvalues:', nev
  end if

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

! ** show detailed info unless -terse option is given by user
  PetscCallA(PetscOptionsHasName(PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, '-terse', terse, ierr))
  if (terse) then
    PetscCallA(EPSErrorView(eps, EPS_ERROR_RELATIVE, PETSC_NULL_VIEWER, ierr))
  else
    PetscCallA(PetscViewerPushFormat(PETSC_VIEWER_STDOUT_WORLD, PETSC_VIEWER_ASCII_INFO_DETAIL, ierr))
    PetscCallA(EPSConvergedReasonView(eps, PETSC_VIEWER_STDOUT_WORLD, ierr))
    PetscCallA(EPSErrorView(eps, EPS_ERROR_RELATIVE, PETSC_VIEWER_STDOUT_WORLD, ierr))
    PetscCallA(PetscViewerPopFormat(PETSC_VIEWER_STDOUT_WORLD, ierr))
  end if
  PetscCallA(EPSDestroy(eps, ierr))
```

(continues on next page)

(continued from previous page)

```
  PetscCallA(MatDestroy(A, ierr))

  PetscCallA(SlepcFinalize(ierr))
end program ex1f
```

Bibliography

[BalPU] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. Curfman McInnes, R. Tran Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, H. Suh, S. Zampini, H. Zhang, H. Zhang, and J. Zhang. *PETSc/TAO Users Manual*. 2026. doi:10.2172/2998643.

[Bal97] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, 163–202. Birkhaüser, 1997.

[Her09] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. A survey of software for sparse eigenvalue problems. Technical Report STR-6, Universitat Politècnica de València, 2009. URL: https://slepc.upv.es/documentation.

[MPI94] MPI Forum. MPI: a message-passing interface standard. *Int. J. Supercomp. Applic. High Perf. Comp.*, 8(3/4):159–416, 1994.

[Alv25] F. Alvarruiz, B. Mellado-Pinto, and J. E. Roman. Variants of thick-restart Lanczos for the Bethe-Salpeter eigenvalue problem. *arXiv:2503.20920 : retrieved 27 May 2025*, 2025. doi:10.48550/arXiv.2503.20920.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[Che00] T.-Y. Chen and J. W. Demmel. Balancing sparse matrices for computing eigenvalues. *Linear Algebra Appl.*, 309(1–3):261–287, 2000. doi:10.1016/S0024-3795(00)00014-8.

[Gol00] G. H. Golub and H. A. van der Vorst. Eigenvalue computation in the 20th century. *J. Comput. Appl. Math.*, 123(1-2):35–65, 2000. doi:10.1016/S0377-0427(00)00413-1.

[Her07a] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. Orthogonalization routines in SLEPc. Technical Report STR-1, Universitat Politècnica de València, 2007. URL: https://slepc.upv.es/documentation.

[Mae16] Y. Maeda, T. Sakurai, and J. E. Roman. Contour integral spectrum slicing method in SLEPc. Technical Report STR-11, Universitat Politècnica de València, 2016. URL: https://slepc.upv.es/documentation.

[Mor06] R. B. Morgan and M. Zeng. A harmonic restarted Arnoldi algorithm for calculating eigenvalues and determining multiplicity. *Linear Algebra Appl.*, 415(1):96–113, 2006. doi:10.1016/j.laa.2005.07.024.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, NJ, 1980. Reissued with revisions by SIAM, Philadelphia, 1998.

[Saa92] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. John Wiley and Sons, New York, 1992.

[San19] D. Sangalli, A. Ferretti, H. Miranda, C. Attaccalite, I. Marri, E. Cannuccia, P. Melo, M. Marsili, F. Paleari, A. Marrazzo, G. Prandini, P. Bonfà, M. O. Atambo, F. Affinito, M. Palummo, A. Molina-Sánchez, C. Hogan, M. Grüning, D. Varsano, and A. Marini. Many-body perturbation theory calculations using the yambo code. *J. Condens. Matter Phys.*, 31(32):325902, 2019. doi:10.1088/1361-648x/ab15d0.

[Ste01a] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[Cam12] C. Campos and J. E. Roman. Strategies for spectrum slicing based on restarted Lanczos methods. *Numer. Algorithms*, 60(2):279–295, 2012. doi:10.1007/s11075-012-9564-z.

[Can00] A. Canning, L. W. Wang, A. Williamson, and A. Zunger. Parallel empirical pseudopotential electronic structure calculations for million atom systems. *J. Comput. Phys.*, 160(1):29–41, 2000. doi:10.1006/jcph.2000.6440.

[Eri80] T. Ericsson and A. Ruhe. The spectral transformation Lanczos method for the numerical solution of large sparse generalized symmetric eigenvalue problems. *Math. Comp.*, 35(152):1251–1268, 1980. doi:10.1090/S0025-5718-1980-0583502-2.

[Fan12] H. Fang and Y. Saad. A filtered Lanczos procedure for extreme and interior eigenvalue problems. *SIAM J. Sci. Comput.*, 34(4):A2220–A2246, 2012. doi:10.1137/110836535.

[Gri94] R. G. Grimes, J. G. Lewis, and H. D. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl.*, 15(1):228–272, 1994. doi:10.1137/S0895479888151111.

[Leh01] R. B. Lehoucq and A. G. Salinger. Large-scale eigenvalue calculations for stability analysis of steady flows on massively parallel computers. *Int. J. Numer. Methods Fluids*, 36:309–327, 2001. doi:10.1002/fld.135.

[Mae16] Y. Maeda, T. Sakurai, and J. E. Roman. Contour integral spectrum slicing method in SLEPc. Technical Report STR-11, Universitat Politècnica de València, 2016. URL: https://slepc.upv.es/documentation.

[Mee97] K. Meerbergen and A. Spence. Implicitly restarted Arnoldi with purification for the shift-invert transformation. *Math. Comp.*, 66(218):667–689, 1997. doi:10.1090/S0025-5718-97-00844-2.

[Mee94] K. Meerbergen, A. Spence, and D. Roose. Shift-invert and Cayley transforms for detection of rightmost eigenvalues of nonsymmetric matrices. *BIT*, 34(3):409–423, 1994. doi:10.1007/BF01935650.

[Nou87] B. Nour-Omid, B. N. Parlett, T. Ericsson, and P. S. Jensen. How to implement the spectral transformation. *Math. Comp.*, 48(178):663–673, 1987. doi:10.1090/S0025-5718-1987-0878698-5.

[Rom14] E. Romero and J. E. Roman. A parallel implementation of Davidson methods for large-scale eigenvalue problems in SLEPc. *ACM Trans. Math. Software*, 40(2):13:1–13:29, 2014. doi:10.1145/2543696.

[Sco82] D. S. Scott. The advantages of inverted operators in Rayleigh-Ritz approximations. *SIAM J. Sci. Statist. Comput.*, 3(1):68–75, 1982. doi:10.1137/0903006.

[Alv24] F. Alvarruiz, C. Campos, and J. E. Roman. Thick-restarted joint Lanczos bidiagonalization for the GSVD. *J. Comput. Appl. Math.*, 440:115506, 2024. doi:10.1016/j.cam.2023.115506.

[And99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[Bjo96] Å. Björck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[Gol96] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[Han98] P. C. Hansen. *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[Her07c] V. Hernandez, J. E. Roman, and A. Tomas. Restarted Lanczos bidiagonalization for the SVD in SLEPc. Technical Report STR-8, Universitat Politècnica de València, 2007. URL: https://slepc.upv.es/documentation.

[Onn91] R. Onn, A. O. Steinhardt, and A. Bojanczyk. The hyperbolic singular value decomposition and applications. *IEEE Trans. Signal Proces.*, 39(7):1575–1588, 1991. doi:10.1109/78.134396.

[Bet08] T. Betcke. Optimal scaling of generalized and polynomial eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 30(4):1320–1338, 2008. doi:10.1137/070704769.

[Bet11] T. Betcke and D. Kressner. Perturbation, extraction and refinement of invariant pairs for matrix polynomials. *Linear Algebra Appl.*, 435(3):514–536, 2011. doi:10.1016/j.laa.2010.06.029.

[Cam16b] C. Campos and J. E. Roman. Parallel iterative refinement in polynomial eigenvalue problems. *Numer. Linear Algebra Appl.*, 23(4):730–745, 2016. doi:10.1002/nla.2052.

[Cam16a] C. Campos and J. E. Roman. Parallel Krylov solvers for the polynomial eigenvalue problem in SLEPc. *SIAM J. Sci. Comput.*, 38(5):S385–S411, 2016. doi:10.1137/15M1022458.

[Cam20a] C. Campos and J. E. Roman. A polynomial Jacobi-Davidson solver with support for non-monomial bases and deflation. *BIT*, 60(2):295–318, 2020. doi:10.1007/s10543-019-00778-z.

[Cam20b] C. Campos and J. E. Roman. Inertia-based spectrum slicing for symmetric quadratic eigenvalue problems. *Numer. Linear Algebra Appl.*, 27(4):e2293, 2020. doi:10.1002/nla.2293.

[Tis00] F. Tisseur. Backward error and condition of polynomial eigenvalue problems. *Linear Algebra Appl.*, 309(1–3):339–361, 2000. doi:10.1016/S0024-3795(99)00063-4.

[Tis01] F. Tisseur and K. Meerbergen. The quadratic eigenvalue problem. *SIAM Rev.*, 43(2):235–286, 2001. doi:10.1137/S0036144500381988.

[Cam21] C. Campos and J. E. Roman. NEP: a module for the parallel solution of nonlinear eigenvalue problems in SLEPc. *ACM Trans. Math. Software*, 47(3):23:1–23:29, 2021. doi:10.1145/3447544.

[Gut17] S. Güttel and F. Tisseur. The nonlinear eigenvalue problem. *Acta Numerica*, 26:1–94, 2017. doi:10.1017/S0962492917000034.

[Mae16] Y. Maeda, T. Sakurai, and J. E. Roman. Contour integral spectrum slicing method in SLEPc. Technical Report STR-11, Universitat Politècnica de València, 2016. URL: https://slepc.upv.es/documentation.

[Meh04] V. Mehrmann and H. Voss. Nonlinear eigenvalue problems: a challenge for modern eigenvalue methods. *GAMM Mitt.*, 27(2):121–152, 2004. doi:10.1002/gamm.201490007.

[Eie06] M. Eiermann and O. G. Ernst. A restarted Krylov subspace method for the evaluation of matrix functions. *SIAM J. Numer. Anal.*, 44(6):2481–2504, 2006. doi:10.1137/050633846.

[Hig10] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010. doi:10.1017/S0962492910000036.

[Sid98] R. B. Sidje. Expokit: a software package for computing matrix exponentials. *ACM Trans. Math. Software*, 24(1):130–156, 1998. doi:10.1145/285861.285868.

[Elm13] H. Elman and M. Wu. Lyapunov inverse iteration for computing a few rightmost eigenvalues of large generalized eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 34(4):1685–1707, 2013. doi:10.1137/120897468.

[Kre08] D. Kressner. Memory-efficient Krylov subspace techniques for solving large-scale Lyapunov equations. In *2008 IEEE International Conference on Computer-Aided Control Systems*, 613–618. IEEE, 2008. doi:10.1109/cacsd.2008.4627370.

[Mee10] K. Meerbergen and A. Spence. Inverse iteration for purely imaginary eigenvalues with application to the detection of Hopf bifurcations in large-scale problems. *SIAM J. Matrix Anal. Appl.*, 31(4):1982–1999, 2010. doi:10.1137/080742890.

[And99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[Hig10] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010. doi:10.1017/S0962492910000036.

[And99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[Auc11] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Comput.*, 37(12):783–794, 2011. doi:10.1016/j.parco.2011.05.002.

[Bla97] L. S. Blackford, 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. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[Kny07] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov. Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in HYPRE and PETSc. *SIAM J. Sci. Comput.*, 29(5):2224–2239, 2007. doi:10.1137/060661624.

[Leh98] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[Li19] R. Li, Y. Xi, L. Erlandson, and Y. Saad. The eigenvalues slicing library (EVSL): algorithms, implementation, and software. *SIAM J. Sci. Comput.*, 41(4):C393–C415, 2019. doi:10.1137/18M1170935.

[Mas96] K. J. Maschhoff and D. C. Sorensen. PARPACK: an efficient portable large scale eigenvalue package for distributed memory parallel architectures. *Lect. Notes Comp. Sci.*, 1184:478–486, 1996.

[Pol09] E. Polizzi. Density-matrix-based algorithm for solving eigenvalue problems. *Phys. Rev. B*, 79(11):115112, 2009. doi:10.1103/PhysRevB.79.115112.

[Pou13] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: a new framework for distributed memory dense matrix computations. *ACM Trans. Math. Software*, 2013. doi:10.1145/2427023.2427030.

[Sta10] A. Stathopoulos and J. R. McCombs. PRIMME: PReconditioned Iterative MultiMethod Eigensolver: methods and software description. *ACM Trans. Math. Software*, 37(2):21:1–21:30, 2010. doi:10.1145/1731022.1731031.

[Suk19] D. Sukkari, H. Ltaief, A. Esposito, and D. Keyes. A QDWH-based SVD software framework on distributed-memory manycore systems. *ACM Trans. Math. Software*, 2019. doi:10.1145/3309548.

[Win19] J. Winkelmann, P. Springer, and E. Di Napoli. ChASE: Chebyshev accelerated subspace iteration eigensolver for sequences of hermitian eigenvalue problems. *ACM Trans. Math. Software*, 2019. doi:10.1145/3313828.