

---

# SLEPc for Python

*Release 3.5.1*

**Lisandro Dalcin**

March 29, 2015

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
1.1	Features . . . . .	2
1.2	Components . . . . .	3
<b>2</b>	<b>Tutorial</b>	<b>5</b>
2.1	Commented source of a simple example . . . . .	5
2.2	Example of command-line usage . . . . .	7
<b>3</b>	<b>Installation</b>	<b>9</b>
3.1	Requirements . . . . .	9
3.2	Using <b>pip</b> or <b>easy_install</b> . . . . .	9
3.3	Using <b>distutils</b> . . . . .	10

## Abstract

This document describes `slepc4py`, a `Python` port to the `SLEPc` libraries.

`SLEPc` is a software package for the parallel solution of large-scale eigenvalue problems. It can be used for computing eigenvalues and eigenvectors of large, sparse matrices, or matrix pairs, and also for computing singular values and vectors of a rectangular matrix.

`SLEPc` relies on `PETSc` for basic functionality such as the representation of matrices and vectors, and the solution of linear systems of equations. Thus, `slepc4py` must be used together with its companion `petsc4py`.

# 1 Overview

*SLEPc for Python* (`slepc4py`) is a `Python` package that provides convenient access to the functionality of `SLEPc`.

`SLEPc` <sup>1, 2</sup> implements algorithms and tools for the numerical solution of large, sparse eigenvalue problems on parallel computers. It can be used for linear eigenvalue problems in either standard or generalized form, with real or complex arithmetic. It can also be used for computing a partial SVD of a large, sparse, rectangular matrix, and to solve nonlinear eigenvalue problems (polynomial or general). Additionally, `SLEPc` provides solvers for the computation of the action of a matrix function on a vector.

`SLEPc` is intended for computing a subset of the spectrum of a matrix (or matrix pair). One can for instance approximate the largest magnitude eigenvalues, or the smallest ones, or even those eigenvalues located near a given region of the complex plane. Interior eigenvalues are harder to compute, so `SLEPc` provides different methodologies. One such method is to use a spectral transformation. Cheaper alternatives are also available.

## 1.1 Features

Currently, the following types of eigenproblems can be addressed:

- Standard eigenvalue problem,  $Ax=kx$ , either for Hermitian or non-Hermitian matrices.
- Generalized eigenvalue problem,  $Ax=kBx$ , either Hermitian positive-definite or not.
- Partial singular value decomposition of a rectangular matrix,  $Au=sv$ .
- Polynomial eigenvalue problem,  $P(k)x=0$ .

---

<sup>1</sup> J. E. Roman, C. Campos, E. Romero, A. Tomas. *SLEPc Users Manual*. DSIC-II/24/02 - Revision 3.5 D. Sistemas Informáticos y Computación, Universitat Politècnica de València. 2014.

<sup>2</sup> Vicente Hernandez, Jose E. Roman and Vicente Vidal. *SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems*, *ACM Trans. Math. Softw.* 31(3), pp. 351-362, 2005.

- Nonlinear eigenvalue problem,  $T(k)x=0$ .
- Computing the action of a matrix function on a vector,  $w=f(\alpha A)v$ .

For the linear eigenvalue problem, the following methods are available:

- Krylov eigensolvers, particularly Krylov-Schur, Arnoldi, and Lanczos.
- Davidson-type eigensolvers, including Generalized Davidson and Jacobi-Davidson.
- Subspace iteration and single vector iterations (inverse iteration, RQI).
- Conjugate gradient for the minimization of the Rayleigh quotient.
- A contour integral solver.

For singular value computations, the following alternatives can be used:

- Use an eigensolver via the cross-product matrix  $A'A$  or the cyclic matrix  $[0\ A; A'\ 0]$ .
- Explicitly restarted Lanczos bidiagonalization.
- Implicitly restarted Lanczos bidiagonalization (thick-restart Lanczos).

For polynomial eigenvalue problems, the following methods are available:

- Use an eigensolver to solve the generalized eigenvalue problem obtained after linearization.
- TOAR and Q-Arnoldi, memory efficient variants of Arnoldi for polynomial eigenproblems.

Computation of interior eigenvalues is supported by means of the following methodologies:

- Spectral transformations, such as shift-and-invert. This technique implicitly uses the inverse of the shifted matrix  $(A-tI)$  in order to compute eigenvalues closest to a given target value,  $t$ .
- Harmonic extraction, a cheap alternative to shift-and-invert that also tries to approximate eigenvalues closest to a target,  $t$ , but without requiring a matrix inversion.

Other remarkable features include:

- High computational efficiency, by using NumPy and SLEPc under the hood.
- Data-structure neutral implementation, by using efficient sparse matrix storage provided by PETSc. Implicit matrix representation is also available by providing basic operations such as matrix-vector products as user-defined Python functions.
- Run-time flexibility, by specifying numerous settings at the command line.
- Ability to do the computation in parallel.

## 1.2 Components

SLEPc provides the following components, which are mirrored by slepc4py for its use from Python. The first five components are solvers for different classes of problems, while the rest can be considered auxiliary objects.

- EPS** The Eigenvalue Problem Solver is the component that provides all the functionality necessary to define and solve an eigenproblem. It provides mechanisms for completely specifying the problem: the problem type (e.g. standard symmetric), number of eigenvalues to compute, part of the spectrum of interest. Once the problem has been defined, a collection of solvers can be used to compute the required solutions. The behaviour of the solvers can be tuned by means of a few parameters, such as the maximum dimension of the subspace to be used during the computation.
- SVD** This component is the analog of EPS for the case of Singular Value Decompositions. The user provides a rectangular matrix and specifies how many singular values and vectors are to be computed, whether the largest or smallest ones, as well as some other parameters for fine tuning the computation. Different solvers are available, as in the case of EPS.
- PEP** This component is the analog of EPS for the case of Polynomial Eigenvalue Problems. The user provides the coefficient matrices of the polynomial. Several parameters can be specified, as in the case of EPS. It is also possible to indicate whether the problem belongs to a special type, e.g., symmetric or gyroscopic.
- NEP** This component covers the case of general nonlinear eigenproblems,  $T(\lambda)x=0$ .
- MFN** This component provides the functionality for computing the action of a matrix function on a vector. Given a matrix  $A$  and a vector  $b$ , the call `MFN-Solve(mfn,b,x)` computes  $x=f(A)b$ , where  $f$  is a function such as the exponential.
- ST** The Spectral Transformation is a component that provides convenient implementations of common spectral transformations. These are simple transformations that map eigenvalues to different positions, in such a way that convergence to wanted eigenvalues is enhanced. The most common spectral transformation is shift-and-invert, that allows for the computation of eigenvalues closest to a given target value.
- BV** This component encapsulates the concept of a set of Basis Vectors spanning a vector space. This component provides convenient access to common operations such as orthogonalization of vectors. The BV component is usually not required by end-users.
- DS** The Dense System (or Direct Solver) component, used internally to solve dense eigenproblems of small size that appear in the course of iterative eigensolvers.
- FN** A component used to define mathematical functions. This is required by the end-user for instance to define function  $T(\cdot)$  when solving nonlinear eigenproblems with NEP in split form.

## 2 Tutorial

This tutorial is intended for basic use of slepc4py. For more advanced use, the reader is referred to SLEPc tutorials as well as to slepc4py reference documentation.

### 2.1 Commented source of a simple example

In this section, we include the source code of example `demo/ex1.py` available in the slepc4py distribution, with comments inserted inline.

The first thing to do is initialize the libraries. This is normally not required, as it is done automatically at import time. However, if you want to gain access to the facilities for accessing command-line options, the following lines must be executed by the main script prior to any petsc4py or slepc4py calls:

```
import sys, slepc4py
slepc4py.init(sys.argv)
```

Next, we have to import the relevant modules. Normally, both PETSc and SLEPc modules have to be imported in all slepc4py programs. It may be useful to import NumPy as well:

```
from petsc4py import PETSc
from slepc4py import SLEPc
import numpy
```

At this point, we can use any petsc4py and slepc4py operations. For instance, the following lines allow the user to specify an integer command-line argument `n` with a default value of 30 (see the next section for example usage of command-line options):

```
opts = PETSc.Options()
n = opts.getInt('n', 30)
```

It is necessary to build a matrix to define an eigenproblem (or two in the case of generalized eigenproblems). The following fragment of code creates the matrix object and then fills the non-zero elements one by one. The matrix of this particular example is tridiagonal, with value 2 in the diagonal, and -1 in off-diagonal positions. See petsc4py documentation for details about matrix objects:

```
A = PETSc.Mat().create()
A.setSizes([n, n])
A.setFromOptions()
A.setUp()

rstart, rend = A.getOwnershipRange()

# first row
```

```

if rstart == 0:
    A[0, :2] = [2, -1]
    rstart += 1
# last row
if rend == n:
    A[n-1, -2:] = [-1, 2]
    rend -= 1
# other rows
for i in range(rstart, rend):
    A[i, i-1:i+2] = [-1, 2, -1]

```

```
A.assemble()
```

The solver object is created in a similar way as other objects in `petsc4py`:

```
E = SLEPc.EPS(); E.create()
```

Once the object is created, the eigenvalue problem must be specified. At least one matrix must be provided. The problem type must be indicated as well, in this case it is HEP (Hermitian eigenvalue problem). Apart from these, other settings could be provided here (for instance, the tolerance for the computation). After all options have been set, the user should call the `setFromOptions()` operation, so that any options specified at run time in the command line are passed to the solver object:

```

E.setOperators(A)
E.setProblemType(SLEPc.EPS.ProblemType.HEP)
E.setFromOptions()

```

After that, the `solve()` method will run the selected eigensolver, keeping the solution stored internally:

```
E.solve()
```

Once the computation has finished, we are ready to print the results. First, some informative data can be retrieved from the solver object:

```
Print = PETSc.Sys.Print
```

```

Print()
Print("*****")
Print("*** SLEPc Solution Results ***")
Print("*****")
Print()

```

```

its = E.getIterationNumber()
Print("Number of iterations of the method: %d" % its)

```

```
eps_type = E.getType()
```

```

Print("Solution method: %s" % eps_type)

nev, ncv, mpd = E.getDimensions()
Print("Number of requested eigenvalues: %d" % nev)

tol, maxit = E.getTolerances()
Print("Stopping condition: tol=%.4g, maxit=%d" % (tol, maxit))

```

For retrieving the solution, it is necessary to find out how many eigenpairs have converged to the requested precision:

```

nconv = E.getConverged()
Print("Number of converged eigenpairs %d" % nconv)

```

For each of the `nconv` eigenpairs, we can retrieve the eigenvalue `k`, and the eigenvector, which is represented by means of two `petsc4py` vectors `vr` and `vi` (the real and imaginary part of the eigenvector, since for real matrices the eigenvalue and eigenvector may be complex). We also compute the corresponding relative errors in order to make sure that the computed solution is indeed correct:

```

if nconv > 0:
    # Create the results vectors
    vr, wr = A.getVecs()
    vi, wi = A.getVecs()
    #
    Print()
    Print("          k          ||Ax-kx||/||kx|| ")
    Print("-----")
    for i in range(nconv):
        k = E.getEigenpair(i, vr, vi)
        error = E.computeRelativeError(i)
        if k.imag != 0.0:
            Print(" %9f%+9f j %12g" % (k.real, k.imag, error))
        else:
            Print(" %12f          %12g" % (k.real, error))
    Print()

```

## 2.2 Example of command-line usage

Now we illustrate how to specify command-line options in order to extract the full potential of `slepc4py`.

A simple execution of the `demo/ex1.py` script will result in the following output:

```
$ python demo/ex1.py
```

```
*****
```

\*\*\* SLEPc Solution Results \*\*\*

\*\*\*\*\*

Number of iterations of the method: 4  
Solution method: krylovschur  
Number of requested eigenvalues: 1  
Stopping condition: tol=1e-07, maxit=100  
Number of converged eigenpairs 4

k	$  Ax-kx  /  kx  $
3.989739	5.76012e-09
3.959060	1.41957e-08
3.908279	6.74118e-08
3.837916	8.34269e-08

For specifying different setting for the solver parameters, we can use SLEPc command-line options with the `-eps` prefix. For instance, to change the number of requested eigenvalues and the tolerance:

```
$ python demo/ex1.py -eps_nev 10 -eps_tol 1e-11
```

The method used by the solver object can also be set at run time:

```
$ python demo/ex1.py -eps_type lanczos
```

All the above settings can also be change within the source code by making use of the appropriate `slepc4py` method. Since options can be set from within the code and the command-line, it is often useful to view the particular settings that are currently being used:

```
$ python demo/ex1.py -eps_view
```

EPS Object: 1 MPI processes

type: krylovschur

Krylov-Schur: 50% of basis vectors kept after restart

problem type: symmetric eigenvalue problem

selected portion of the spectrum: largest eigenvalues in magnitude

number of eigenvalues (nev): 1

number of column vectors (ncv): 16

maximum dimension of projected problem (mpd): 16

maximum number of iterations: 100

tolerance: 1e-08

convergence test: relative to the eigenvalue

BV Object: 1 MPI processes

type: svec

17 columns of global length 30

orthogonalization method: classical Gram-Schmidt

orthogonalization refinement: if needed (eta: 0.7071)



```
DS Object: 1 MPI processes
  type: hep
  solving the problem with: Implicit QR method (_steqr)
ST Object: 1 MPI processes
  type: shift
  shift: 0
  number of matrices: 1
```

Note that for computing eigenvalues of smallest magnitude we can use the option `-eps_smallest_magnitude`, but for interior eigenvalues things are not so straightforward. One possibility is to try with harmonic extraction, for instance to get the eigenvalues closest to 0.6:

```
$ python demo/ex1.py -eps_harmonic -eps_target 0.6
```

Depending on the problem, harmonic extraction may fail to converge. In those cases, it is necessary to specify a spectral transformation other than the default. In the command-line, this is indicated with the `-st_` prefix. For example, shift-and-invert with a value of the shift equal to 0.6 would be:

```
$ python demo/ex1.py -st_type sinvert -eps_target 0.6
```

## 3 Installation

### 3.1 Requirements

You need to have the following software properly installed in order to build *SLEPc for Python*:

- Any [MPI](#) implementation <sup>3</sup> (e.g., [MPICH](#) or [Open MPI](#)), built with shared libraries.
- A matching version of [PETSc](#) built with shared libraries.
- A matching version of [SLEPc](#) built with shared libraries.
- [Python](#) 2.4 to 2.7 or 3.1 to 3.4 <sup>4</sup>.
- [NumPy](#) package.
- [petsc4py](#) package.

### 3.2 Using pip or easy\_install

You can use **pip** to install `slepc4py` and its dependencies (`mpi4py` is optional but highly recommended):

---

<sup>3</sup> Unless you have appropriately configured and built SLEPc and PETSc without MPI (configure option `--with-mpi=0`).

<sup>4</sup> You may need to use a parallelized version of the Python interpreter with some MPI-1 implementations (e.g. MPICH1).

```
$ pip install [--user] numpy mpi4py
$ pip install [--user] petsc petsc4py
$ pip install [--user] slepc slepc4py
```

Alternatively, you can use **easy\_install** (deprecated):

```
$ [sudo] easy_install [--user] slepc4py
```

If you already have working PETSc and SLEPc builds, set environment variables `SLEPC_DIR` and `PETSC_DIR` (and perhaps `PETSC_ARCH` for prefix installs) to appropriate values and next use **pip**:

```
$ export SLEPC_DIR=/path/to/slepc
$ export PETSC_DIR=/path/to/petsc
$ export PETSC_ARCH=arch-linux2-c-opt
$ pip install [--user] petsc4py slepc4py
```

## 3.3 Using distutils

### Downloading

The *SLEPc for Python* package is available for download at the project website generously hosted by Bitbucket. You can use **curl** or **wget** to get a release tarball.

- Using **curl**:

```
$ curl -O https://bitbucket.org/slepc/slepc4py/downloads/slepc4py-X.Y.tar.gz
```

- Using **wget**:

```
$ wget https://bitbucket.org/slepc/slepc4py/downloads/slepc4py-X.Y.tar.gz
```

### Building

After unpacking the release tarball:

```
$ tar -zxvf slepc4py-X.Y.tar.gz
$ cd slepc4py-X.Y
```

the distribution is ready for building.

---

**Note:** Mac OS X users employing a Python distribution built with **universal binaries** may need to set the environment variables `MACOSX_DEPLOYMENT_TARGET`, `SDKROOT`, and `ARCHFLAGS` to appropriate values. As an example, assume your Mac is running **Snow Leopard** on a **64-bit Intel** processor and you want to override the hard-wired cross-development SDK in Python configuration, your environment should be modified like this:

```
$ export MACOSX_DEPLOYMENT_TARGET=10.6
$ export SDKROOT=/
$ export ARCHFLAGS='-arch x86_64'
```

---

Some environment configuration is needed to inform the location of PETSc and SLEPc. You can set (using **setenv**, **export** or what applies to you shell or system) the environment variables `SLEPC_DIR`, `PETSC_DIR`, and `PETSC_ARCH` indicating where you have built/installed SLEPc and PETSc:

```
$ export SLEPC_DIR=/usr/local/slepc
$ export PETSC_DIR=/usr/local/petsc
$ export PETSC_ARCH=arch-linux2-c-opt
```

Alternatively, you can edit the file `setup.cfg` and provide the required information below the [config] section:

```
[config]
slepc_dir  = /usr/local/slepc
petsc_dir  = /usr/local/petsc
petsc_arch = arch-linux2-c-opt
...
```

Finally, you can build the distribution by typing:

```
$ python setup.py build
```

## Installing

After building, the distribution is ready for installation.

If you have root privileges (either by log-in as the root user or by using **sudo**) and you want to install *SLEPc for Python* in your system for all users, just do:

```
$ python setup.py install
```

The previous steps will install the `slepc4py` package at standard location `prefix/lib/pythonX.X/site-packages`.

If you do not have root privileges or you want to install *SLEPc for Python* for your private use, you have two options depending on the target Python version.

- For Python 2.6 and up:

```
$ python setup.py install --user
```

- For Python 2.5 and below (assuming your home directory is available through the `HOME` environment variable):

```
$ python setup.py install --home=$HOME
```

**Finally, add `$HOME/lib/python` or `$HOME/lib64/python` to your `PYTHONPATH` environment variable.**